

TOWARDS TRUSTWORTHY ON-DEVICE COMPUTATION

by

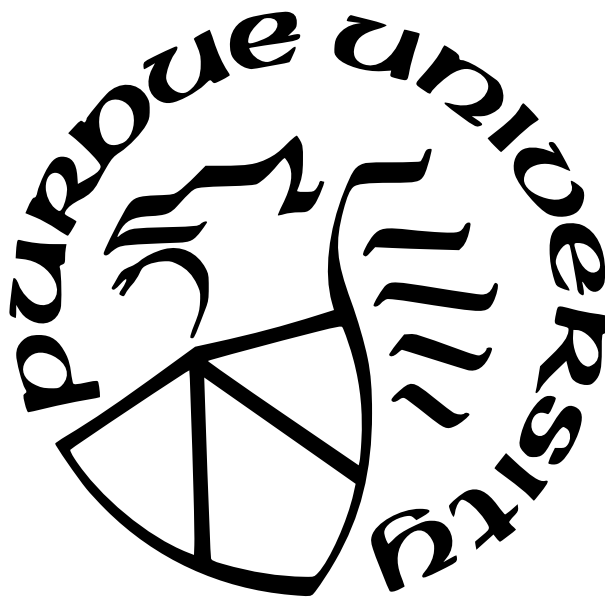
Heejin Park

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy



School of Electrical and Computer Engineering

West Lafayette, Indiana

May 2022

**THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL**

Dr. Felix Xiaozhu Lin, Chair

School of Electrical and Computer Engineering

Dr. Y. Charlie Hu

School of Electrical and Computer Engineering

Dr. Saurabh Bagchi

School of Electrical and Computer Engineering

Dr. Tim Rogers

School of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

ACKNOWLEDGMENTS

This long journey would not have been successfully completed without the support from many people in my life. First and foremost, I am most grateful to my advisor, Professor Felix Xiaozhu Lin, for his careful guidance during my time of Ph.D. study. He has been an exemplary advisor in conducting systems research, but also a great life mentor.

I would also like to thank the rest of my committee members, Professor Y. Charlie Hu, Professor Saurabh Bagchi, and Professor Tim Rogers, for their responsiveness and insightful comments on this work. Undoubtedly, their feedback have improved my idea and the dissertation.

It was my pleasure to be a member of system research group, XSEL. I would like to thank all my colleagues Hongyu Miao, Liwei Guo, Tiantu Xu, and Shuang Zhai, for useful discussion and companionship throughout this long journey. I shall not forget the great time we had together in XSEL and at Purdue.

Last but not least, I am grateful to all my family members, particularly my parents, for their unconditional love and support. I would like to dedicate my dissertation to my family.

TABLE OF CONTENTS

LIST OF TABLES	9
LIST OF FIGURES	10
ABSTRACT	12
1 INTRODUCTION	13
1.1 Motivation	13
1.2 Dissertation Statement	14
1.2.1 StreamBox-TZ: Secure Stream Analytics at the Edge with TrustZone	14
1.2.2 GPURip: A 50-KB GPU Stack for Client ML	15
1.2.3 GPU Acceleration in TrustZone via Safe and Practical Recording . .	15
1.3 Dissertation Organization	16
2 STREAMBOX-TZ: SECURE STREAM ANALYTICS AT THE EDGE WITH TRUST- ZONE	17
2.1 Introduction	17
2.2 Background & Motivation	20
2.2.1 ARM for Cloud Edge	20
2.2.2 Stream Analytics	21
2.2.3 Security Threats & Design Objectives	22
2.3 Security Approach Overview	23
2.3.1 Scope	23

2.3.2	Approach and Security Benefits	24
2.4	Design Overview	26
2.4.1	Challenges	26
2.4.2	StreamBox-TZ in a Nutshell	28
2.5	Trusted Primitives and Optimizations	30
2.6	TEE Memory Management	31
2.6.1	Unbounded Array	31
2.6.2	Placing uArrays in uGroups	32
2.7	Attestation for Correctness and Freshness	34
2.8	Implementation	37
2.9	Evaluation	38
2.9.1	TCB Analysis	38
2.9.2	Performance & Overhead	39
2.9.3	Validation of Key Design Features	45
2.10	Related Work	47
2.11	Conclusions	48
3	GPURIP: A 50-KB GPU STACK FOR CLIENT ML	49
3.1	Introduction	49
3.2	Motivations	52
3.2.1	The GPU stack and its problems	52

3.2.2	GPU Trends We Exploit	54
3.3	GPURip	55
3.3.1	Using GPURip	55
3.3.2	The GPU Model	58
3.4	Record	60
3.4.1	Interface Knowledge and Instrumentation	60
3.4.2	Register access	61
3.4.3	Dumping proprietary job binaries	61
3.4.4	Locating input and output for a recording	62
3.4.5	Pace replay actions	63
3.5	Replay	65
3.5.1	Verification of security properties	65
3.5.2	The nano GPU driver	66
3.5.3	GPU handoff and preemption	66
3.5.4	Handling replay failures	66
3.6	Implementations and Experiences	67
3.6.1	The recorder for Arm Mali	68
3.6.2	The recorder for Broadcom v3d	68
3.6.3	Replayers in various environments	68
3.6.4	Reusing recordings across GPU models	70

3.7	Evaluation	70
3.7.1	Analysis	71
3.7.2	Validation of replay correctness	72
3.7.3	Memory overheads	74
3.7.4	Replay speed	75
3.7.5	Validation of Key Designs	77
3.8	Related Work	79
3.9	Concluding Remarks	80
4	GPU ACCELERATION IN TRUSTZONE VIA SAFE AND PRACTICAL RECORD- ING	81
4.1	Introduction	81
4.2	Motivations	85
4.2.1	Mobile GPUs	85
4.2.2	Prior Approaches	86
4.2.3	GPURip in TrustZone	86
4.2.4	The Problem of Recording Environment	87
4.3	CoDry	89
4.3.1	The Approach	89
4.3.2	The CoDry architecture	91
4.3.3	Challenge: long network delays	91

4.4	Hiding Register Access Delays	92
4.4.1	Register Access Deferral	92
4.4.2	Speculation	96
4.4.3	Offloading polling loops	98
4.5	Memory Synchronization	100
4.6	Implementations	102
4.7	Evaluation	104
4.7.1	Security Analysis	104
4.7.2	Performance	105
4.7.3	Validation of key designs	107
4.7.4	Energy consumption	109
4.8	Related Work	110
4.9	Conclusions	111
5	CONCLUSIONS	113
	REFERENCES	115

LIST OF TABLES

2.1	Comparison to existing secure processing systems	29
2.2	Selected trusted primitives (23 in total) and operators they constitute. These operators cover most listed in the Spark Streaming documentation [93].	29
2.3	The test platform used in experiments	39
2.4	A breakdown of the StreamBox-TZ source, of which 5K SLoC are in TCB. Binary code sizes shown in parentheses	39
2.5	Engine versions for comparison (plots in Figure 2.8)	40
3.1	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	58
3.2	Replay actions in a recording	61
3.3	GPURip implementations. * = used in evaluation. See Table 3.5 for evaluated recordings	67
3.4	Codebase comparisons. Binaries are stripped.	70
3.5	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 . . .	73
4.1	Statistics of record runs, showing CoDry significantly reduces network round trips that block the recording and the memory synchronization traffic	107
4.2	Replay delays of CoDry (OursMDS) are similar to Native, which executes benchmarks on the GPU stack in the normal world of the same device	107

LIST OF FIGURES

2.1	An overview of StreamBox-TZ	19
2.2	Example stream data, operators, and a pipeline	21
2.3	SBT on an edge platform with ARM TrustZone. Bold arrows show the protected data path.	25
2.4	Among alternative architectures for secure stream analytics, StreamBox-TZ (c) leads to the smallest TCB and the most optimized data plane. Arrows indicate data flows.	27
2.5	The uArrays in one uGroup	33
2.6	Audit records: fields (top) and layout (bottom)	35
2.7	Sample audit records for the pipeline in Figure 2.2. Format is simplified. ts means processing timestamp.	36
2.8	StreamBox-TZ throughput (lines, left/right y-axes) as a function of CPU cores (x-axis) under given output delays (above each plot). Steady consumptions of TEE memory as columns with annotated values. See Table 2.5 for legends and explanations.	42
2.9	StreamBox-TZ achieves much higher throughput than commodity insecure engines [51], [57], [92] on HiKey. Benchmark: windowed aggregation; target output delay: 50ms.	43
2.10	Run time breakdown of operator GroupBy under different input batch sizes. The control plane runs 8 threads to execute GroupBy in parallel. Total execution time is normalized to 100%.	44
2.11	Without consumption hints, the allocator uses more TEE memory. Since memory usage fluctuates at run time, the error bars show two standard deviations below and above the average.	45
2.12	On-demand growth of uArrays vs. std::vector	46
2.13	Compression of audit records saves uplink bandwidth substantially.	47
3.1	The overview of GPURip	50
3.2	The software/hardware for an integrated GPU	53
3.3	Replaying NN execution with GPURip	55
3.4	The different ways to deploy a GPURip replayer	57

3.5	Intervals between CPU/GPU interactions, accumulated by GPU job. Intervals among earlier jobs are longer than later ones. Workload: AlexNet inference. ACL [148] on Mali G71. Excluded: GPU busy time; parameters loading IO	64
3.6	Startup delays prior to NN inference. The replayer (GR) takes much less time than the original GPU stack (OS).	74
3.7	NN inference delays. The replayer (GR) incurs similar delays as compared to the original GPU stack (OS).	75
3.8	NN training delays. Benchmark: MNIST training atop DeepCL + OpenCL on Mali G71 (OS: orig stack; GR: GPURip).	77
3.9	Mali G71 can replay recordings from other GPUs at full hardware speed. Benchmark: 16M elements vecadd	77
3.10	GPURip removes unnecessary intervals between replay actions. Benchmark: ACL NN inference atop Mali G71	78
3.11	NN inference delays (including startup) with various granularities. The count of recordings is annotated.	78
4.1	System overview. Shaded components belong to CoDry	82
4.2	A timeline for replaying NN inference	87
4.3	Numbers of mobile GPU models per year [241]	88
4.4	CoDry’s online recording. The cloud VM collaboratively runs GPU stack with the client GPU.	89
4.5	CoDry’s strategies for hiding long RTTs	93
4.6	Selective memory synchronization of GPU metastate but not program data .	100
4.7	Recording delays of CoDry(Ours MDS) are significantly lower than other versions	106
4.8	Breakdown of speculation; the actual number of commits for each segment are shown in parentheses.	108
4.9	Energy consumption for record and replay	109

ABSTRACT

Driven by breakthroughs in mobile and IoT devices, on-device computation becomes promising. Meanwhile, there is a growing concern over its security: it faces many threats in the wild, while not supervised by security experts; the computation is highly likely to touch users' privacy-sensitive information. Towards trustworthy on-device computation, we present novel system designs focusing on two key applications: stream analytics, and machine learning training and inference.

First, we introduce Streambox-TZ (SBT), a secure stream analytics engine for ARM-based edge platforms. SBT contributes a data plane that isolates only analytics' data and computation in a trusted execution environment (TEE). By design, SBT achieves a minimal trusted computing base (TCB) inside TEE, incurring modest security overhead.

Second, we design a minimal GPU software stack (50KB), called GPURip. GPURip allows developers to record GPU computation ahead of time, which will be replayed later on client devices. In doing so, GPURip excludes the original GPU stack from run time eliminating its wide attack surface and exploitable vulnerabilities.

Finally, we propose CoDry, a novel approach for TEE to record GPU computation remotely. CoDry provides an online GPU recording in a safe and practical way; it hosts GPU stacks in the cloud that collaboratively perform a dryrun with client GPU models. To overcome frequent interactions over a wireless connection, CoDry implements a suite of key optimizations.

1. INTRODUCTION

1.1 Motivation

A recent support of powerful processors expedites on-device computation in mobile/edge environments, which directly processes sensor data on low-end (client) devices not a cloud server. The key application is mobile/edge intelligence including stream analytics and machine learning. In such applications, system security including data and computation is crucial. (1) They are often parts of mission-critical systems [1]–[3], e.g. aircraft control, electricity grid and emergency communication systems. Hence, a corrupted computation may lead to a critical result on the entire system. (2) As being processed, the data potentially include confidential information [4]–[6], e.g. proprietary sensor data or video frames.

However, a security issues over on-device computation have been constantly reported [7]–[9], raising concerns of end users. Fundamentally, the on-device computation faces following security threats. (1) Unlike a cloud which is well supervised by security professionals [10], [11], a client device is managed by end users who are often non-experts. Hence, the device may suffer from a weak security configuration, a late security update, or a slow countermeasure to a new attacks. (2) The data flows through a set of sophisticated components that exposes a wide attack surface. For example, the client device should host a complex software stack for stream analytics or GPU stack for machine learning workloads, where exploitable vulnerabilities are not uncommon [12], [13]. (3) With data including privacy-sensitive information, e.g. user’s health activities, voice records, and photos, a client device is easily considered as a high-value target to adversaries.

For these reasons, we claim that a new system design is necessary to achieve trustworthy on-device computation with strong security guarantees. Once attackers compromise devices, they not only access confidential data but also may fabricate the result, threatening the integrity of an entire system.

1.2 Dissertation Statement

This dissertation aims at addressing security concerns over on-device computation. Towards trustworthy on-device computation, we draw new system designs taking application-specific security properties into account. Our first objective is to enable a secure stream analytics on an edge platform, which safeguards data confidentiality and integrity, supports verifiable results, and ensures high throughput with low output delay. Our second objective is to enable a secure GPU acceleration on client devices, eliminating a complex, vulnerable GPU stack at run time. Our last objective is to provide a cloud service for a secure GPU acceleration on client devices in safe and practical ways. To achieve the above goals, we consider the following approaches.

- Analyzing key applications and minimizing corresponding software stacks
- Leveraging a hardware-supported trusted execution environment, e.g. ARM TrustZone
- Involving a trustworthy cloud in the computation.

We demonstrate the efficacy of our approaches with the three main projects.

1.2.1 StreamBox-TZ: Secure Stream Analytics at the Edge with TrustZone

To achieve our first objective, we present StreamBox-TZ, a secure analytics engine at an edge platform. StreamBox-TZ isolates the data and its computation in a trusted execution environment (TEE) on the edge, shielding them from the remaining edge software stack which is deemed untrusted. StreamBox-TZ offers strong data security, verifiable results, and good performance. It contributes a data plane designed and optimized for a TEE based on ARM TrustZone. It supports continuous remote attestation for analytics correctness and result freshness while incurring low overhead. StreamBox-TZ only adds 42.5 KB executable to the TCB (16% of the entire TCB). On an octa core ARMv8 platform, it delivers the state-of-the-art performance by processing input events up to 140 MB/sec (12M events/sec) with sub-second delay. The overhead incurred by StreamBox-TZ’s security mechanism is less than 25%.

1.2.2 GPURip: A 50-KB GPU Stack for Client ML

We accomplish our second objective with GPURip, 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 GPURip 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 GPURip works with a variety of integrated GPU hardware, GPU APIs, ML frameworks, and 33 neural network (NN) implementations for inference or training.

1.2.3 GPU Acceleration in TrustZone via Safe and Practical Recording

We attain our final objective with CoDry, a holistic design for GPU-accelerated computation in TrustZone TEE. Without pulling the complex GPU software stack into the TEE, we follow a simple approach: record the CPU/GPU interactions ahead of time, and replay the interactions in the TEE at run time. This paper addresses the approach’s key missing piece – the recording environment, which needs both strong security and access to diverse mobile GPUs. To this end, we present a novel architecture called CoDry, in which a mobile device (which possesses the GPU hardware) and a trustworthy cloud service (which runs the GPU software) exercise the GPU hardware/software in a collaborative, distributed fashion. To overcome numerous network round trips and long delays, CoDry contributes optimizations specific to mobile GPUs: register access deferral, speculation, and metastate-only synchronization. With these optimizations, recording a compute workload takes only tens of seconds, which is up to 95% less than a naive approach; replay incurs 25% lower delays compared to insecure, native execution.

1.3 Dissertation Organization

This dissertation is organized as follows. Chapter 2 describes StreamBox-TZ, a secure stream analytics leveraging ARM TrustZone. Chapter 3 presents GPURip, which squeezes attack surfaces and removes vulnerabilities from GPU stack by record and replay GPU computation. Chapter 4 proposes CoDry, a cloud service for collaborative dryrun to record GPU computation with client TEE. Chapter 5 summarizes the dissertation.

2. STREAMBOX-TZ: SECURE STREAM ANALYTICS AT THE EDGE WITH TRUSTZONE

2.1 Introduction

Many key applications of Internet of Things (IoT) process a large influx of sensor¹ data, i.e. telemetry. Oil producers track tank status, pump pressure, and fluid temperatures to see if wells work at ideal operating points [14], [15]; an oil rig is reported to produce 1–2TB of data per day [16]. Smart grid aggregates power telemetry to detect supply/demand imbalance and power disturbances [17]; a power sensor is reported to produce up to 140 million samples per day [18], [19]. Manufacturers routinely monitor vibration and ultrasonic energy of industrial equipment to discover anomalies and do predictive maintenance; a monitored machine is reported to create PBs of data in a matter of days.

The large telemetry data streams must be processed in time. The high cost and long delay in transmitting the data necessitate edge processing [20], [21]: sensors send the data to nearby gateways dubbed “cloud edge”; the edge runs a pipeline of continuous computations to cleanse and summarize the telemetry data and reports the results to cloud servers for deeper analysis. The hardware at the edge is often optimized for cost and efficiency. According to a 2018 survey [22], modern ARM machines are typical choices for edge platforms. Such a platform often has 2–8 CPU cores and several GB DRAM.

Unfortunately, edge processing exposes the IoT data to significant security threats. i) Deployed in the wild, the edge suffers from common IoT weaknesses such as lack of professional supervision [23], [24], weak configurations [25], [26], and delayed security updates [23], [27]. ii) On the edge, the IoT data flows through a set of sophisticated components that expose a wide attack surface. These components include a commodity OS (e.g. Linux or Windows), a set of user libraries, and a runtime framework called *stream analytics engine* [28]–[30]. They reuse much code developed for servers and workstations. It is not uncommon for them to have exploitable misconfigurations [31] and vulnerabilities [32]–[34] iii) As data aggregated from various sources, the edge become a high-value target to adversaries. For these reasons,

¹↑Recognizing that IoT data sources range from small sensors to large equipment, we refer to them all as *sensors* for brevity.

edge is even *more vulnerable* than sensors, which run much simpler software with narrower attack surfaces. Once attackers compromise the edge, they can not only obtain confidential data but also remove or fabricate data transmitted to the cloud, threatening the integrity of an entire IoT deployment.

Towards secure stream analytics on an edge platform, our goal is to safeguard IoT data confidentiality and integrity, support verifiable results, and ensure high throughput with low output delay. Following the principle of least privilege [35], we protect the analytics data and computations in a trusted execution environment (TEE) and limit their interface; we leave out the remaining edge software stack which we deem untrusted. In doing so, the trusted computing base (TCB) is reduced to only the protected functionalities, the TEE, and the hardware. We hence significantly enhance data security.

We face three challenges: i) what functionalities should be secured in TEE and behind what interfaces? ii) how to execute stream analytics on a TEE’s low TCB and limited physical memory while still delivering high throughput and low delay? iii) as both trusted and untrusted edge components participate in stream analytics, how to verify the outcome?

Existing solutions are inadequate: pulling entire stream analytics engines to TEE [36]–[38] would result in a large TCB with a wide attack surface; the systems securing distributed operators [39]–[41] often lack stream semantics or optimizations for efficient execution in a single TEE, which are crucial to the edge; only attesting TEE integrity [42] or data lineages [39], [40], [43], [44] is insufficient to verify stream analytics. We will show more evidences in the paper.

Our response is SBT, a secure engine for analytics over telemetry data streams. As shown in Figure 2.1, SBT builds on ARM TrustZone [45] on an edge platform. SBT contributes the following notable designs:

(1) *Architecting a data plane for protection* SBT provides a data plane exposing narrow, shared-nothing interfaces to untrusted software. SBT’s data plane encloses i) all the analytics data; ii) a new library of low-level stream algorithms called *trusted primitives* as the only allowed computations on the data; iii) key runtime functions, including memory management and cache-coherent parallel execution of trusted primitives. SBT leaves thread scheduling and synchronization out of TEE.

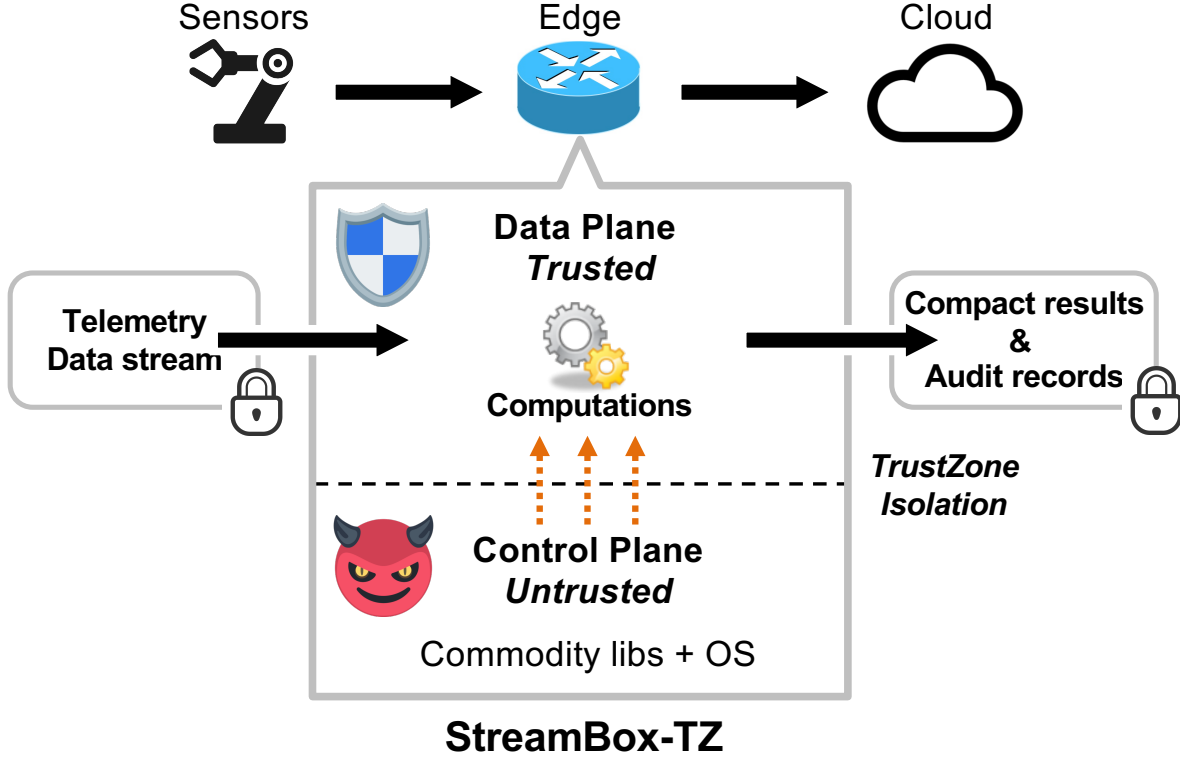


Figure 2.1. An overview of StreamBox-TZ

(2) *Optimizing data plane performance within a TEE* In contrast to many TEE-oblivious stream engines that operate numerous small objects, hash tables, and generic memory allocators [46]–[48], SBT embraces unconventional design decisions for its data plane. i) SBT implements trusted primitives with array-based algorithms and contributes new optimizations with handwritten ARMv8 vector instructions. ii) To process high-velocity data in TEE, SBT introduces a new abstraction called uArrays, which are contiguous, virtually unbounded buffers for encapsulating all the analytics data; SBT backs uArrays with on-demand paging in TEE and manages uArrays using a dedicated allocator. For the compact memory layout, the allocator leverages hints from untrusted software. iii) SBT takes advantage of TrustZone’s lesser-explored hardware features: ingesting data straightly through trusted IO without a detour through the untrusted OS; avoiding relocating streaming data by leveraging the large virtual address space dedicated to a TEE.

(3) *Verifying edge analytics execution* SBT provides cloud verifiers to attest analytics *correctness*, result *freshness*, and the untrusted hints received during execution. SBT captures

coarse-grained dataflows and generates audit records. A cloud verifier replays the audit records for attestation. To save the edge-cloud uplink bandwidth, SBT compresses the records with domain-specific encoding.

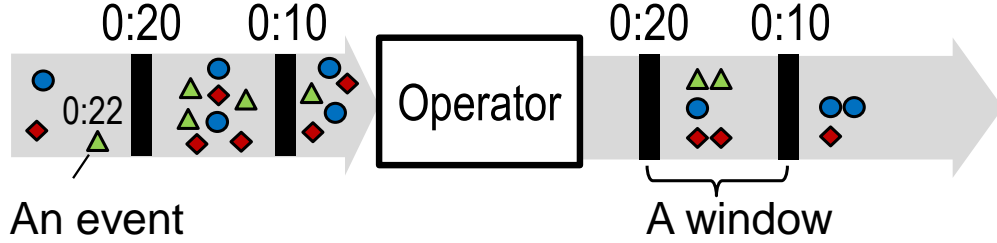
We implement SBT to offer a generic stream model [49] with a broad arsenal of stream operators. The TCB of SBT contains as little as 267.5 KB of executable code, of which SBT only constitutes 16%. On an octa core ARMv8 platform, SBT processes up to 12M events (144 MB) per second at sub-second output delays. Its throughput on this platform is an order of magnitude higher than an SGX-based secure stream engine running on a small x86 cluster with richer hardware resources [41]. SBT’s security mechanisms incur less than 25% throughput loss with the same output delay; decrypting ingress data, when required, results in 4%–35% throughput loss with the same output delay. In most benchmarks, SBT consumes up to 130 MB of physical memory while sustaining high throughput.

The key contributions of SBT are: i) a stream engine architecture with strongly isolated data and a lean TCB; ii) a data plane built from the ground up with computations and memory management optimized for a single TrustZone-based TEE; iii) remote attestation for stream analytics on the edge with domain-specific compression of audit records. To our knowledge, SBT is the first system designed and optimized for data-intensive, parallel computations inside ARM TrustZone. Beyond stream analytics, the SBT architecture should aid in secure other important analytics on the edge, e.g. machine learning inference. The SBT source can be found at <http://github.com/edgeflow-dev>.

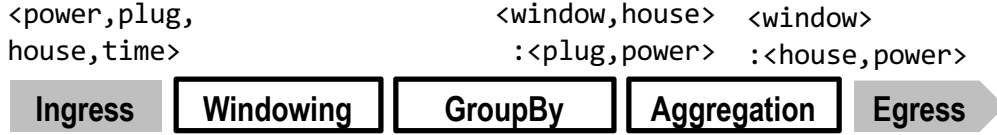
2.2 Background & Motivation

2.2.1 ARM for Cloud Edge

Recent ARM platforms are typically preferred as hardware for IoT gateways [22], because of competitive performance at low power, which suits edge as well. Most modern ARM cores are equipped with TrustZone [45], a security extension for TEE enforcement. TrustZone logically partitions a platform’s hardware resources, e.g. DRAM and IO, into a normal (insecure) and a secure world. CPU cores independently switch between two worlds. A



(a) A stream of events flowing through an operator.



(b) A simple analytics pipeline that predicts power grid loads

```

/* 1. Declare operators */
Ingress in(/* config info */);
Window w(1_SECOND); GroupBy<house> gb;
Aggregation<house, win> ag; Egress out;
/* 2. Create a pipeline. Connect operators */
Pipeline p; p.apply(in);
in.connect(w).connect(gb)
.connect(ag).connect(out);
/* 3. Execute the pipeline */
Runner r( /* config */ ); r.run(p);

```

(c) Simplified pseudo code declaring the above pipeline

Figure 2.2. Example stream data, operators, and a pipeline

TEE atop TrustZone owns dedicated channel called *trusted IO*, a unique feature that TEE technologies e.g. Intel SGX [50] lack.

2.2.2 Stream Analytics

Stream Model We focus on stream analytics over sensor data. A data stream comprises of sensor events that carry timestamps defined by event occurrence, as illustrated in Figure 2.2(a). Programmers define a pipeline of continuous computations called *operators*, e.g. Filter, Count, and GroupBy, that are widely used for telemetry analytics [51], [52]. As data arrives at the edge, a stream analytics engine ingests the data at the pipeline ingress, pushes the data through the pipeline, and externalizes the results at the pipeline egress.

We follow a generic stream model [46], [48], [53]–[55]. Operators execute on event-time scopes called *windows*. Data sources emit special events called *watermarks*. A watermark

ensures that no subsequent events in the stream will have event times earlier the watermark timestamp. A pipeline’s *output delay* is defined as the elapsed time starting from the moment the ingress receives the watermark signaling the completion of the current window to the moment the egress externalizes the window results [47]. A pipeline may maintain its internal states organized by windows at different operators. See prior work [56] for a formal stream model.

Analytics example: Power load prediction Figure 2.2(b-c) depicts an example derived from an IoT scenario [52]: it forecasts future household power loads based on the current loads reported by smart power plugs. The example pipeline ingests a stream of power samples and groups them by 1-second fixed windows and by houses. For each house in each window, it aggregates all the loads and predicts the next-window load as an exponentially weighted moving average over the recent windows. At the egress, the pipeline emits a stream of per-house load prediction for each window.

Stream analytics engines Stream pipelines are executed by a runtime framework called a stream analytics engine [28]–[30], [47], [51], [57]. A stream analytics engine consists of two types of function: *data functions* for data move and computations; *control functions* for resource management and computation orchestration, e.g. creating and scheduling tasks. The boundary between the two is often blurry. To amortize overheads, control functions often organize data in *batches* and invoke data functions to operate on the batches.

2.2.3 Security Threats & Design Objectives

The edge faces common security threats in IoT deployment. First, IT expertise is weak. Edge platforms are likely managed by field experts (such as farmers [27], firefighters [24], and petroleum engineers [23]) rather than IT experts. Such lack of professional supervision is known to result in weak configurations [25], [26]. Second, the infrastructure is weak. Deployed in the field (e.g. farms [27], fireground [24], and offshore oil rigs [23]), the edge often experiences slow uplinks to the cloud and hence much delayed software security updates. For cost saving, edge analytics may need to share OS and hardware with other high-risk, untrusted software [27] such as web browsers.

Apart from the common threats, conventional edge software stacks entrust IoT data with commodity operating systems, analytics engines, and language runtimes (e.g. JVM). However, such components are unable to provide strong security guarantees due to their complexity and wide interfaces. Each of them is likely to contain more than several hundreds of KSLoC [58]. A number of vulnerabilities are constantly reported [32], [33], [59]–[61], making these components untrusted in recent research [62]–[65]. By exploiting these vulnerabilities, a local adversary as an edge user program may compromise the kernel through the wide user/kernel interfaces [66], [67] or attack an analytics engine through IPC [68]; a remote adversary, through the edge’s network services, may compromise analytics engines [69] or the OS [70]. A successful adversary may expose IoT data, corrupt the data, or covertly manipulate the data. Taking the application in Figure 2.2(b) as an example, the adversary gains access to the smart plug readings, which may contain residents’ private information, and injects fabricated data.

Objectives We aim three objectives for stream analytics over telemetry data on an edge platform: i) confidentiality and integrity of IoT data, raw or derived; ii) verifiable correctness and freshness of the analytics results; iii) modest security overhead and good performance.

2.3 Security Approach Overview

2.3.1 Scope

IoT scenarios Our target is an edge platform that collects and analyzes telemetry data. We recognize the significance of mission-critical IoT with tight control loops, but do not target it. In our target scenario, there are source sensors, edge platforms, and a cloud server that we dub “cloud consumer”. All the raw IoT data and analytics results are owned by one party. The sensors produce trusted events, e.g. by using secure sensing techniques [71]–[73]. The cloud consumer is trustworthy; it defines analytics pipelines to the edge and consumes the results from the edge. We consider *untrusted* source-edge links (e.g. public networks) which requires data encryption by the source, as well as *trusted* source-edge links (e.g. direct IO bus or on-premise local networks), and will evaluate the corresponding designs (§2.9). We assume untrusted edge-cloud links, which require encryption of the uploaded data.

In-scope Threats We assume malicious adversaries interested in gaining access to IoT data, tampering with edge processing outcome, or obstructing processing progress. We consider powerful adversaries who can control the entire OS and all applications on the edge by exploiting weak configuration or vulnerabilities in the edge software.

Out-of-scope Threats We do not protect the confidentiality of stream pipelines, in the interest of including only low-level compute primitives in a lean TCB. We do not defend the following attacks.

- i) Attacks to non-edge components assumed trusted above, e.g. sensors [74].
- ii) Exploitation of TEE kernel bugs [75]–[77].
- iii) Side channel attacks: by observing hardware usage outside TEE, adversaries may learn the properties of protected data, e.g. key skew [78]. Note that controlled-channel attack [79] cannot be applied to ARM TrustZone as it has separate page management within a separate secure OS unlike Intel SGX.
- iv) Physical attacks, e.g. sniffing TEE’s DRAM access [80], [81]. Note that many of these attacks are mitigated by prior work [40], [82]–[84] orthogonal to SBT.

2.3.2 Approach and Security Benefits

As depicted in Figure 2.3, SBT protects its data functions in a trusted *data plane* in TEE. SBT runs its untrusted *control plane* in the normal world. The control plane invokes the data plane via narrow, shared-nothing interfaces. As a result, the engine’s TCB only contains the TEE (including the data plane) and the hardware.

Streaming data always flows in TEE where the data plane ingests the data through TrustZone’s trusted IO. After ingestion, the data plane returns *opaque references* of the data batches to the control plane. The opaque references are long, random integers. The control plane then requests computations over the protected data via invoking the data plane with the opaque references. The data plane keeps track of all active opaque references, validates incoming opaque references, and only accept those that genuine. At the pipeline egress, the data plane encrypts, signs, and delivers the result to the cloud.

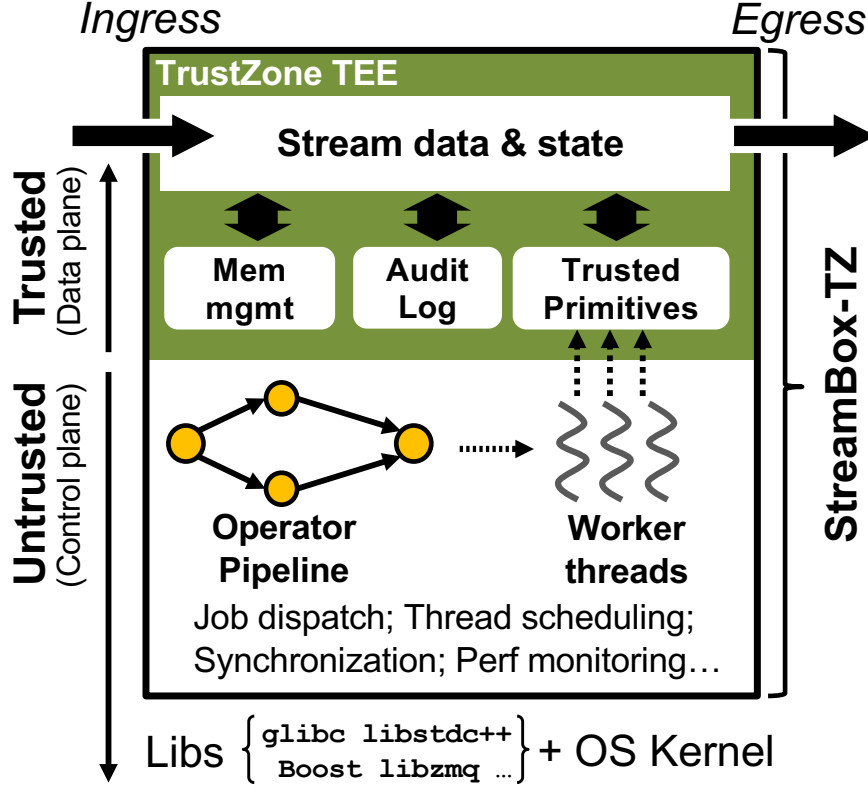


Figure 2.3. SBT on an edge platform with ARM TrustZone. Bold arrows show the protected data path.

The analytics execution is continuously attested. SBT collects comprehensive and deterministic dataflows of the stream analytics as well as execution timing; it sends periodic reports to the cloud server. The cloud server then verifies if all the ingested data correctly flow through the given pipeline (correctness,) and if the processing incurs low delay (freshness).

Thwarted attacks SBT defeats the following attacks.

i) ***Breaking IoT data confidentiality or integrity.*** Adversaries on the edge cannot touch, drop, or inject data since raw and derived data enters and leaves the dge TEE via trusted IOs. When data is sent off the edge over untrusted networks, it is encrypted and hence protected against network-lever adversaries.

ii) ***Breaking the data plane integrity.*** As all opaque references are validated before in use, any fabricated opaque reference passed to the data plane will be denied. An adversary

may exploit bugs in the data plane and compromise it through the data plane’s interface. However, SBT substantially reduces the data plane’s attack surface and potential exploitable bugs by minimizing the data plane codebase and hardening its interface.

iii) ***Breaking analytics correctness.*** A compromised control plane may request computations deviating from pipeline declarations or the stream model. It may, for example, invoke trusted computations on partial data, incorrect windows, or legitimate but undesirable opaque references. SBT defeats these attacks by remote attestation: the cloud verifier can detect such correctness violation and rejects the edge analytics results since it processes complete knowledge on ingested data and pipelines.

iv) ***Attacks on analytics performance or availability.*** A compromised control plane may pause or delay invocation of trusted computations, violating the freshness guarantee. However, the cloud verifier can detect such attacks by attesting the execution timing of trusted computation. When the attack is detected, it can choose to prompt further investigation.

v) ***Attempting to trigger data race or deadlock.*** By design, data race and deadlocks will never occur inside the data plane: the trusted computations do not share state concurrently and all locking takes place outside of the TEE.

2.4 Design Overview

2.4.1 Challenges

Our approach raises three challenges. i) ***Architecting the engine with a proper protection boundary.*** This is dependent on a key trade-off among TEE functional richness, overhead of TEE entry/exit, and TCB size. ii) ***Optimizing data functions within a TEE.*** To process high-velocity data in a TEE, simple algorithms and compact memory are significantly favored. Existing stream engines, on the other hand, often use a large number of short-lived objects indexed in hash tables or trees [46]–[48], [51], [55], e.g. for grouping events by key. To manage these objects, they use generic memory allocators [47] or garbage collectors [46], [85]. Such designs are unsuitable for a TEE’s small TCB and limited DRAM portion, e.g. typically tens of MB for a TrustZone TEE and up to 128 MB for

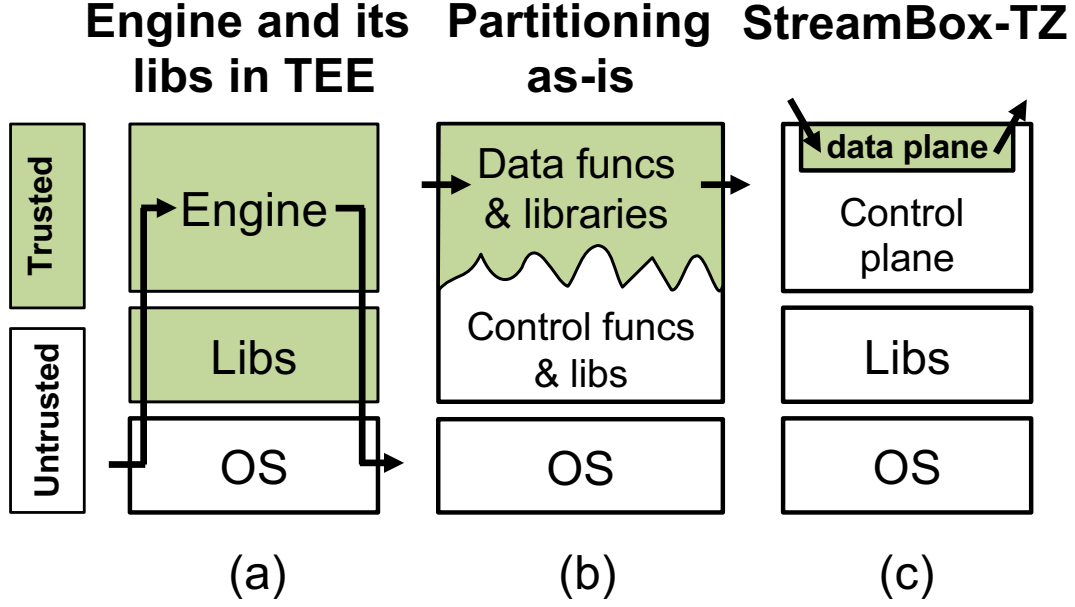


Figure 2.4. Among alternative architectures for secure stream analytics, StreamBox-TZ (c) leads to the smallest TCB and the most optimized data plane. Arrows indicate data flows.

an Intel SGX enclave [86]. iii) *Verifying stream analytics results*. This necessitates tracking unbounded data flows in stream pipelines, validating that operators respect the temporal properties, e.g. windows, and minimizing the resultant overhead in execution and communication.

Why are existing systems inadequate? First, many TEE-based systems [36]–[38] pull entire user applications and libraries to the TCB, as illustrated in Figure 2.4(a). A modern analytics engine and its libraries, on the other hand, are huge, complex, and potentially vulnerable as we discussed in Section 2.2.2. Second, partitioning applications to fit a TEE, as shown in Figure 2.4(b) [87]–[89], is unsuitable for existing stream engines: partitioning does not change their hash-based data structures and algorithms, which by design mismatch a TEE. Similarly, recent secure processing engines disfavor partitioning [90], [91]. Third, recent systems deploy TEE to protect data in analytics or network packet processing. As summarized in Table 2.1, they lack support for stream analytics, key computation optimizations, or specialized memory allocation, that we will demonstrate as vital to our objective.

To assert analytics correctness, attesting TEE integrity [42], [90] is insufficient. VC3 [39] and Opaque [40] check the history of compute results, i.e. their data lineage [43], [44], to verify correctness of *batch* analytics. Without tracking data as it is continually ingested and without a stream model, data lineages cannot assert whether *all* ingested data processed correctly according to pipeline declarations, temporal windows, and watermarks, which are crucial to stream analytics.

2.4.2 StreamBox-TZ in a Nutshell

SBT builds on TrustZone [45] because of ARM’s popularity for the edge and trusted IO, which benefits stream analytics (§2.2).

Programmability SBT provides a similar programmability like what commodity engines like Spark Streaming [46] and Flink [92] support. Analytics programmers build pipelines using declarative high-level operators as exemplified in Figure 2.2(c). These stream operators are commonly used for analytics over telemetry data [51], [52]. SBT supports the majority of the common operators supplied by commodity engines, as summarized in Table 2.2. These operators are commonly used for analytics over telemetry data [51], [52]. SBT also offers User Defined Functions (UDFs) certified by a trusted party, that is a common requirement in TEE-based systems [90].

SBT architecture SBT’s data plane incarnates as a TrustZone module, as shown in Figure 2.3, SBT runs its control plane as a parallel runtime in the normal world. The control plane invokes the data plane through a narrow interface (details in Section 2.9). The control plane orchestrates the execution of analytics pipelines. It generates abundant parallelism among and within operators, that is elastically mapped to a pool of threads it maintains. At a given moment, all threads may work concurrently on one operator as well as different operators over different data.

Data plane & design choices SBT’s data plane consists of only the trusted primitives and a runtime for them.

i) Trusted primitives are stateless, single-threaded functions that are oblivious to synchronization. In the data plan, we do not enclose entire stream pipelines, since a pipeline

Table 2.1. Comparison to existing secure processing systems

System	TEE	Analytics	SG	Compute in TEE	Memory	Attestation
VC3 [39]	SGX	Batch	CIVA-	Mapper/reducer	Heap	Data lineage
Opaque [40]	SGX	Batch	CIVA0	Query plans	unreported	Data lineage
EnclaveDB [90]	SGX	Batch	CI-A-	Pre-compiled queries	unreported	TEE integrity
SafeBricks [91]	SGX	Pkt proc.	CI-A-	Net func. operators*	unreported	TEE integrity
SecureStream [41]	SGX	Stream	CI-	Lua programs	unreported	TEE integrity
StreamBox-TZ	TZ	Stream	CIV-	Vectorized primitives*	uArray	Log replay

SG: security guarantees.

C: data confidentiality; I: data integrity; V: verifiability; A: analytics confidentiality; 0: obliviousness

* TEE encloses only low-level computations; otherwise TEE encloses whole analytics.

Table 2.2. Selected trusted primitives (23 in total) and operators they constitute. These operators cover most listed in the Spark Streaming documentation [93].

Trusted Primitives	Popular Spark Streaming Operators
Sort, Merge, Segment, SumCnt, TopK, Concat, Join, Count, Sum, Unique, FilterBand, Median, ...	GroupByKey, Windowing, AvgPerKey, Distinct, SumByKey, AggregateByKey, SortByKey, TopKPerKey, CountByKey, CountByWindow, Filter, MedianByKey, TempJoin, Union, ...

must be dynamically scheduled for parallel processing over high-velocity data. We do eschew wrapping whole declarative operators in the data plane, as one operator instance contains internal thread-level parallelism and hence requires thread management logic. Our decision keeps the data plane lean, leaving out all control functions including threading and scheduling. In contrast, many other engines shown in Table 2.1 pull entire analytics to TEE. Our choice of exporting low-level primitives entails more TEE switches. Yet, the costs are smaller on modern ARM [75], [94] and can be amortized by batching, data batching, as will be discussed soon.

ii) The data plane incorporates minimum runtime functions: memory management and paging, which are vital to TEE integrity; cache coherence of parallel primitives, which is critical to parallelism. The data plane is agnostic to declarative operators and pipelines being executed.

For attestation, the data plane generates audit records on data ingress/egress, primitive executions, and watermarks. It curtails overhead through data batching and record compression.

Coping with secure memory shortage SBT may suffer from out of secure memory when the compute cost or data ingestion rate is excessive. SBT avoids data loss in such a situation by adding backpressure to source sensors, slowing down data ingestion. In our current implementation, SBT activates backpressure when ingestion exceeds a user-defined threshold; we leave an automatic flow control as our future work, i.e. online threshold tuning based on available secure memory and backlog.

2.5 Trusted Primitives and Optimizations

Parallel execution inside a TEE SBT utilizes task parallelism keeping a lean TEE without a threading library. The control plane invokes numerous primitives with worker threads, which then enter the TEE to execute the primitives in parallel. In TEE, all trusted primitives share a single cache-coherent memory address space, which makes data sharing easier and eschews copy costs. This contrasts to existing secure analytics engines that leave task parallelism untapped in a single TEE [39], [41].

Array-based algorithms to suit TEE Unlike common stream engines that use hash-based algorithms to reduce algorithmic complexity, we make a new design choice. We firmly favor algorithms with straightforward logic and little memory overhead, despite of their higher algorithmic complexity. Corresponding to contiguous arrays as the universal data containers in TEE, most primitives employ sequential-access algorithms over contiguous arrays, e.g, executing Merge-Sort across event arrays and scanning the resultant array to get the average value per key.

Trusted primitives and vectorization The trusted primitives in SBT are generic. They constitute most declarative stream operators, often referred to as Select-Projection-Join-GroupBy (SPJG) families, shown in Table 2.2. These stream operators are deemed representative in prior research [95].

To speed up the array-based up without causing TCB bloat, our insight is to map their internal data parallelism to ARM’s vector instructions [96]. To our knowledge, the vector instructions are barely employed to accelerate data analytics *within* TEEs, in spite of

their well-known performance gain. Vectorization results in low code complexity since the performance benefit stems from a CPU feature which is already part of the TCB.

Our optimization focuses on two core primitives, Sort and Merge, as they dominate the execution of stream analytics in our observation. Inspired by vectorized sort and merge on x86 [97], [98], we implement the ones with hand-writing ARMv8 NEON vector instructions for SBT. Our sort outperforms the ones in the C/C++ standard libraries by more than $2\times$, as will be shown in evaluation. This optimization is vital to the engine’s overall performance.

2.6 TEE Memory Management

Facing high-velocity streams in a TEE, SBT’s memory allocator addresses two challenges: *space efficiency*: it must construct a compact memory layout and reclaim memory timely due to limited physical memory; *lightweight*: the allocator must be simple to suit a low TCB. The challenges disqualify popular engines organizing events in hash tables (e.g. for grouping events by key) and depend on generic memory allocators [46]–[48], [51], [55]. The rationales are two: a hash table’s principle that trade space for time mismatches TEE’s limited memory; generic allocators are typically heavy due to complex optimization, which adds tens of KSLoS to TCB [99], [100].

SBT offers a special memory management for stream computations: it provides virtually unbounded buffers as the universal memory abstraction (§2.6.1); it places data by using (untrusted) consumption hints and large virtual address space (§2.6.2).

2.6.1 Unbounded Array

We device contiguous and virtually unbounded arrays called *uArrays*, the universal data containers used by computations within TEE. uArrays encapsulate all of the data flowing among trusted primitives in a pipeline; they also store operator states conventionally kept in hash tables.

An uArray is an append-only buffer in a contiguous memory region for same-type data objects. The lifecycle of uArray closely map to the producer/consumer pattern in streaming computations. One uArray can be in the following three different states. *Open*: after created,

an uArray grows dynamically as the producer primitive appends data objects to it. *Produced*: when data production is finished, the uArray’s end position is finalized; it becomes read-only and no more data can be appended. *Retired*: the uArray is no longer needed and its memory is subject to reclamation. According to their states, the memory allocator places or reclaims uArrays, as will be discussed in Section 2.6.2.

Types uArrays fall into different types depending on their scopes and enclosed data. A *streaming uArray* encapsulates data flowing from a producer primitive to a consumer primitive. A *state uArray* encapsulates operator state that outlives the lifespans of individual primitives. A *temporary uArray* live within a trusted primitive’s scope.

Low abstraction overhead An uArray transparently grows spanning a contiguous virtual memory region. Its growth is backed by the data plane’s on-demand paging which happens entirely inside TEE. Growing an uArray simply necessitates updating an integer index for most of the time. Compared to a manual buffer management, this mechanism allows the compiler to build more compact loops by waiving bounds checking of uArray in computation code. uArrays always grow in place. This contrasts to typical sequence containers (such as C++ `std::vector` and `java.util.ArrayList`) that grow transparently but require costly relocation. In Section 2.9, we will experimentally compare uArray against `std::vector`.

2.6.2 Placing uArrays in uGroups

Co-locating uArrays The memory allocator co-locates multiple uArrays as an uGroup to reclaim them consecutively. Spanning a contiguous virtual memory region, a uGroup consists of multiple *produced* or *retired* uArrays and optionally an *open* uArray at its end, as shown in Figure 2.5. The grouping is purely physical: it is at the discretion of the allocator, orthogonal to stream computations, and hence transparent to the trusted primitives and the control plane.

As shown in Figure 2.5, the allocator reclaims *consumed* uArrays by always starting from the beginning of an uGroup. When placing a new uArray, the allocator decides whether to create a new uGroup for the uArray, or append the uArray to an existing uGroup. In

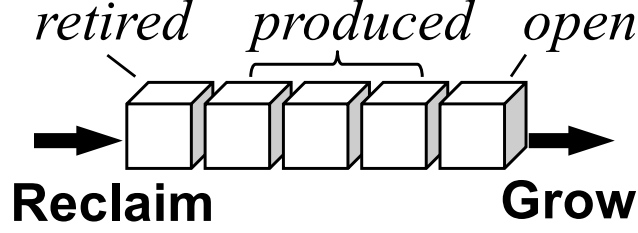


Figure 2.5. The uArrays in one uGroup

doing so, the allocator seeks to i) assure that each uGroup holds a sequence of uArrays to be consumed consecutively in the future; ii) minimize the total number of live uGroups, in order to compact TEE memory layout and minimizes the cost in tracking uGroups. To this end, our key is to guide placement with the control plane’s data consumption plan, as will be presented below.

Consumption hints Upon invoking a trusted primitive T , the control plane may provide two optional hints concerning the future consumption order for the output of T :

- *Consumed-in-parallel* (\parallel_k): the control plane will schedule k worker threads to consume a set of uArrays in parallel.
- *Consumed-after* ($b_1 \Leftarrow b_2$): the control plane will schedule worker threads for consuming uArray b_2 after uArray b_1 . The *consumed-after* relation is transitive. uArrays may form multiple *consumed-after* chains.

The control plane may specify these relations between new output uArrays (yet to be created) and existing uArrays.

Hint-guided placement The hints assist the data plane to generate compact memory layout and reclaim memory effectively. Upon allocating a uArray, the allocator examines the existing hints regarding to the uArray.

(\Leftarrow) prompts the allocator to place the uArrays on the same *consumed-after* chain in the same uGroup. Starting from the new uArray b under question, the allocator tracks back on its consumed-after chain, and places b after the first uArray that is both in state *produced* (i.e. its growth has finished) and is located at the end of an uGroup. If no such uArray is available on the chain, the allocator creates a new uGroup for b .

(\parallel_k) prompts the allocator to place uArrays $b_{1..k}$ in separate uGroups, so that delay in consuming any of the uArrays will not block the allocator from reclaiming the other uArrays. Our rationale is that despite $b_{1..k}$ are created at the same time, they are often consumed at different moments in the future: i) since SBT’s control plane threads independently fetch new uArrays for processing as they become available (§2.4), the starting moments for processing $b_{1..k}$ may vary widely, especially when the engine load is high; ii) even when k worker threads start processing $b_{1..k}$ simultaneously, straggling workers are not uncommon, due to non-determinism of a modern multicore’s thread scheduling and memory hierarchy [101].

The impacts of misleading hints SBT detects misleading hints in retrospect through remote attestation (§2.7). Since hints only influence TEE memory placement *policy* on the edge, misleading hints never cause data loss (§2.4.2) or violation of data security and TEE integrity. However, such hints may result in violation of result freshness slowing down analytics.

Managing virtual addresses All uGroups grow *in place* within one virtual address space. To avoid collision and costly relocation, the allocator places them far apart using the huge virtual address space dedicated to a TrustZone TEE. The space is 256TB on ARMv8, 10,000× larger than the physical DRAM (a few GBs). As a result, the allocator simply reserves a virtual address range as large as the entire TEE DRAM for each uGroup. We will validate such choice in Section 2.9.

2.7 Attestation for Correctness and Freshness

SBT collects evidences for cloud consumers to verify two properties: *correctness*, i.e. all ingested data is processed according to the stream pipeline declaration; *freshness*, i.e. the pipeline has low output delays.

The above objective has several notable aspects. i) We verify the behaviors of untrusted control plane, i.e., *which* primitives it invokes on *what* data and at *what* time. We do not verify trusted primitives, e.g. if a Sort primitive indeed produces ordered data. ii) Verifying data lineages at the pipeline’s intermediate operators or egress [43], [44] is insufficient to guarantee correctness, i.e. all data ingested so far is processed according to the stream

Field	Description	Length
Ts	Data plane timestamp	32 bits
Op	Primitive type, including ingress/egress	16 bits
WinNo	Monotonic window sequence number	16 bits
Data	An uArray ID or a watermark value	32 bits
Hint	An optional consumption hint	64 bits
Count	Number of data/hint fields that follow	16 bits

In/Egress	Op	Ts	Data					
Windowing	Op	Ts	Data	WinNo	Data			
Execution	Op	Ts	Cnt	Data...	Cnt	Data...	Cnt	Hints...

Figure 2.6. Audit records: fields (top) and layout (bottom)

pipeline. iii) The windows of stream computations and watermarks triggering the computations must be attested, which are keys to stream model (§2.2). iv) As the volume of evidences can be substantial, evidences must be compacted to save uplink bandwidth [27], [102].

Therefore, SBT provides the following verification mechanism. Agnostic to the pipeline being executed, the data plane monitors dataflows among primitive instances at the TEE boundary, and then generates audit records. For low overhead, it eschews building data lineages on-the-fly unlike much prior work [39], [43], [103]. The data plane compresses audit records and flushes to the cloud both periodically and upon externalizing any analytics result. We describe details below.

Audit records As being invoked by the control plane, the data plane generates *audit records*. As illustrated in Figure 2.6, the records track i) ingested and externalized uArrays, ii) associations between uArrays and windows, and iii) primitive executions (with optional hints supplied by the control plane) which establish *derived-from* relations among uArrays. The records further include ingested watermark values, which are crucial for determining output delays as will be discussed below. The data plane timestamps all the records. It generates monotonically increasing identifiers for recorded uArrays. We will evaluate the overhead of audit records in Section 2.9.

```

ts= 1 INGRESS data=0xF0
ts= 5 WND data_in=0xF0 win_no=0 data_out=0xF1
ts=10 SORT data_in=0xF1 data_out=0xF3
ts=15 INGRESS data=0xF4 (watermark=100)
ts=25 SUM data_in=0xF3,0xF4 data_out=0xF5
ts=28 WND data_in=0xF0 win_no=1 data_out=0xF6
ts=30 EGRESS data=0xF5

```

Figure 2.7. Sample audit records for the pipeline in Figure 2.2. Format is simplified. ts means processing timestamp.

Attesting analytics correctness The cloud verifier checks if all ingested uArrays flow through the expected trusted primitives. Such dataflows are deterministic given the arrivals of input data (including their windows), the watermarks, and the pipeline declaration. Hence, the verifier replays all ingestion records on its local copy of the same pipeline. It checks if all the records resulting from the replay match the ones reported by the edge (except timestamps). The replay is symbolic without actual computations and hence fast.

Note that the verification works for stateful operators as well. The state of a stream operator (e.g. temporal join) is only determined by all the inputs the operator has ever received. Since the cloud can verify that all the ingested uArrays correctly flow through the expected trusted primitives and thus stream operators, it knows that the operator’s current state must be correct, and then all results derived from the operator state must be correct.

Attesting result freshness The key for the verifier to calculate the delay of an output result R is to identify the watermark that triggers the externalization of R , according to the delay definition in Section 2.2.2. From the egress record of R , the verifier traces *backward* following the *derived-from* chain(s) until it reaches an execution record indicating that a watermark W triggers the execution. The verifier looks up the ingress record of W . It calculates the difference between W ’s ingress time and R ’s egress time to be the delay of R .

Example In Figure 2.7, an uArray with identifier 0xF0 is ingested and segmented into two uArrays (0xF1 and 0xF2) for window 0 and 1 respectively. Sort consumes uArray 0xF1 and produces uArray 0xF3. A watermark with value 100 arrives and completes window 0. Triggered by the watermark, SUM consumes uArray 0xF3 of window 0 and produces uArray 0xF5 as the result of window 0.

The cloud verifier replays the ingress records on its local pipeline copy and learns that uArray 0xF1 is processed adhering to the pipeline declaration while uArray 0xF2 is yet to be processed. It will assert analytics incorrectness if 0xF2 remains unprocessed until a future watermark completes window 1 (not shown). To verify result freshness, the verifier traces result 0xF5 backward to find its trigger watermark 0xF4 and calculates the output delay to be 15 ($30 - 15$).

Columnar compression of records The data plane compresses audit records by exploiting locality within one record field and known data distribution in each field. The data plane produces raw audit records in memory (with the format shown in Figure 2.6) and in a row order, i.e. one record after the other. Before uploading a sequence of records, it separates the record fields (i.e. columns) and applies different encoding schemes to individual columns: i) Huffman encoding for primitive types and data counts, the two columns likely contain skewed values; ii) delta encoding for timestamps, uArray identifiers, and window numbers, which increment monotonically. Our compression is inspired by columnar databases [104]. We will evaluate the efficacy of compression in Section 2.9.

2.8 Implementation

We build SBT for ARMv8 and atop OP-TEE [105] (v2.3). SBT reuses most control functions of StreamBox [47], an open-source research stream engine for x86 servers. Yet, as StreamBox mismatches a TEE (§2.4.1), SBT contributes a new architecture and a new data plane. SBT communicates with source sensors and cloud consumers over ZeroMQ TCP transport [106] which is known for good performance. The new implementation of SBT includes 12.4K SLoC.

Input batch size, a key parameter of SBT, trades off between delays in executing individual primitives, the rate of TEE entry/exit, and attestation cost. We empirically determine it as 100K events and will evaluate its impact (§2.9).

Opaque references for uArrays are 64-bit random integers generated by the data plane. It keeps the mappings from references to uArray addresses in a table, and validates opaque

references by table lookup. This incurs minor overhead, as live opaque references are often no more than a few thousands.

2.9 Evaluation

We answer the following questions through evaluation:

- Does SBT result in a small TCB? (§2.9.1)
- What is SBT’s performance and how is it compared to other engines? What is the overhead? (§2.9.2)
- How do our key designs impact performance (§2.9.3)?

2.9.1 TCB Analysis

TCB size Table 2.4 shows a breakdown of the SBT source code. Despite a sophisticated control plane, the data plane only adds 5K SLoC to the TCB. SBT’s memory management is in 740 SLoC, 9× fewer than glibc’s `malloc` and 20× fewer than `jemalloc` [100]. The size of data plane is 42.5 KB, a small fraction (16%) of the entire OP-TEE binary.

TCB interface The SBT’s data plane exports only four entry functions: two for data plane initialization/finalization, one for debugging, and one shared by all 23 trusted primitives. The last function accepts and returns opaque references (§2.4). No state is shared across the protection boundary.

Comparison with alternative TCBs Compared to enclosing whole applications in TCB [36]–[38], SBT keeps most of the engine out, shrinking the TCB by at least one order of magnitude. Compared to directly carving out [87], [89] the original StreamBox’s data functions for protection, SBT completely avoids sophisticated data structures (e.g. `AtomicHashMap` [107] used by StreamBox) that mismatch TCB. Compared to VC3 [39] that implements Map/Reduce operators in a TCB with ~9K SLoC, SBT supports much richer stream operators within a 2× smaller TCB.

Table 2.3. The test platform used in experiments

SoC	HiSilicon Kirin 620, TDP 36W	CPU	8x ARM Cortex-A53@1.2 GHz
Mem	2GB LPDDR3@800 MHz	OS	Normal: Debian 8 (Linux 4.4) Secure: OP-TEE 2.3

Table 2.4. A breakdown of the StreamBox-TZ source, of which 5K SLoC are in TCB. Binary code sizes shown in parentheses

Data Plane (Trusted)					
Primitives*		Mem Mgmt*	Misc*		Total
3.7K (32.5 KB)		0.7K (6 KB)	0.6K (4 KB)		5K (42.5 KB)
Control Plane (Untrusted)					
Control	Data types*	Operators*	Test*	Misc*	Total
23K	1.3K	4.1K	1K	1K	31K (348 KB)
Major Libraries (Untrusted)					
glibc 2.19	libstdc++ 3.4.2	libzmq 2.2	boost 1.54	Total	
1135K	110K	13K	37K	1.3M (3.1 MB)	

* New implementations of this work. Total = 12.4K SLoC.





2.9.2 Performance & Overhead

Methodology We evaluate SBT on a HiKey board as summarized in Table 2.3. We chose HiKey for its good OP-TEE support [105] and that it is among the few boards with TrustZone programmable by third parties. We built *Generator*, a program sends data streams over ZeroMQ TCP transport [106] to SBT. We run the cloud consumer on an x86 machine. Data streams are encrypted with 128-bit AES.

In the face of HiKey’s platform limitations, we set up the engine ingestion as follows.

- i) Although Gigabit Ethernet on edge platforms is common [108], [109], Hikey’s Ethernet interface (over USB) only has 20MB/sec bandwidth. We have verified that the interface is saturated by SBT with 4 cores. Hence, we report performance when SBT and *Generator* both run on HiKey communicating over ZeroMQ TCP, which still fully exercise the TCP/IP stack and data copy.
- ii) Although HiKey’s TEE is capable of directly operating Ethernet

Table 2.5. Engine versions for comparison (plots in Figure 2.8)

Legend & Version	Data Plane	In/Egress Path	Ingress Data	Egress Data
 StreamBox-TZ	in TEE	Trusted IO*	Encrypted	Encrypted
 SBT ClearIngress	in TEE	Trusted IO*	ClearTxt	Encrypted
 SBT IOviaOS	in TEE	via OS	Encrypted	Encrypted
 Insecure#	out TEE	in OS	ClearTxt	ClearTxt

* Through TrustZone Trusted IO directly to TEE

Equivalent to a StreamBox invoking StreamBox-TZ’s optimized stream compute

interface as trusted IO, our OP-TEE version lacks the needed drivers. Hence, we emulate SBT’s direct data ingestion to TEE by running the ingestion in a privileged process in the normal world, and bypassing data copy across the TEE boundary. Our test harness continuously replays pre-allocated secure memory buffers populated with events.

As summarized in Table 2.5, we test SBT as well as three modified versions: ***SBT ClearIngress*** ingests data in cleartext; this is allowed if source-edge links are trusted as defined in our threat model (§2.3). ***SBT IOviaOS*** does not exploit TrustZone’s trusted IO: the untrusted OS ingested (encrypted) data and copies the data across TEE boundary to the data plane. ***Insecure*** completely runs in the normal world with ingress and egress in cleartext, showing native performance. This is basically StreamBox [47] with SBT’s optimized stream computations (§2.5). We report the engine performance as its maximum input throughput when the pipeline output delay (defined in §2.2.2) remains under a target set by us.

Benchmarks We employ six benchmarks of processing sensor data streams from prior work [47], [48], [52], [110], [111]. They cover major stream operators and a variety of pipelines. We use fixed windows, each encompassing 1M events and spanning 1 second of event time. Each event consists of 3 fields (12 Bytes) unless stated otherwise. (1) **Top Values Per Key (TopK)** groups events based on keys and identifies the K largest values in each group in each window. (2) **Counting Unique Taxis (Distinct)** identifies unique taxi IDs and counts them per window. For input events, we use a dataset of taxi trip information containing 11 K distinct taxi IDs [110]. (3) **Temporal Join (Join)** joins events that have the same keys and

fall into same windows from two input streams. (4) **Windowed Aggregation (WinSum)** aggregates input values within each window. We use the Intel Lab Data `labdata_intel` consisting of real sensor values as input. (5) **Filtering (Filter)** filters out input data, of which field falls into to a given range in each window. We set 1% selectivity as done in prior work [111]. (6) **Power Grid (Power)**, derived from a public challenge [52], finds out houses with most high-power plugs. Ingesting a stream of per-plug power samples, it calculates the average power of each plug in a window and the average power over all plugs in all houses in the window. For each house, it counts the number of plugs that have a higher load than average. It emits the houses that have most high-power plugs in the window. The event for this benchmark is composed of 4 fields (16 Bytes).

Benchmark 2, 4, and 6 use real-world datasets; others use synthetic data sets of which fields are 32-bit random integers. Note that SBT’s `GroupBy` operator bases on sort and merge and is insensitive to key skewness [112].

End-to-end performance Figure 2.8 shows the throughputs of all benchmarks as a function of hardware parallelism. SBT can process up to multiple millions of events within sub-second output delays (labeled atop each plot). For simpler pipelines such as WinSum and Filter, SBT processes around 12M events/sec (140 MB/sec). This throughput saturates one GbE link which is common on IoT gateways [109]. Overall, SBT can use all 8 cores in a scalable manner.

SBT’s absolute performance is state of the art. We test three popular, insecure stream engines: Flink [92], designed for distributed environment and known for good single-node performance [113]; Esper [57], designed for a single machine; SensorBee [51], designed for sensor data processing on a single device. As shown in Figure 2.9, on the same hardware (HiKey) and the same benchmark (WinSum), we have measured that SBT’s throughput is at least one order of magnitude higher than the others. This is because i) our *Insecure* baseline has high performance for its rich task parallelism (inherited from StreamBox [47]) and native, vectorized stream computations (new contributions); ii) SBT only imposes modest security overhead, as will be shown later.

Comparison to secure stream engines The comparison is challenged by that TrustZone was rarely exploited for protecting data-intensive computations. To our knowledge,

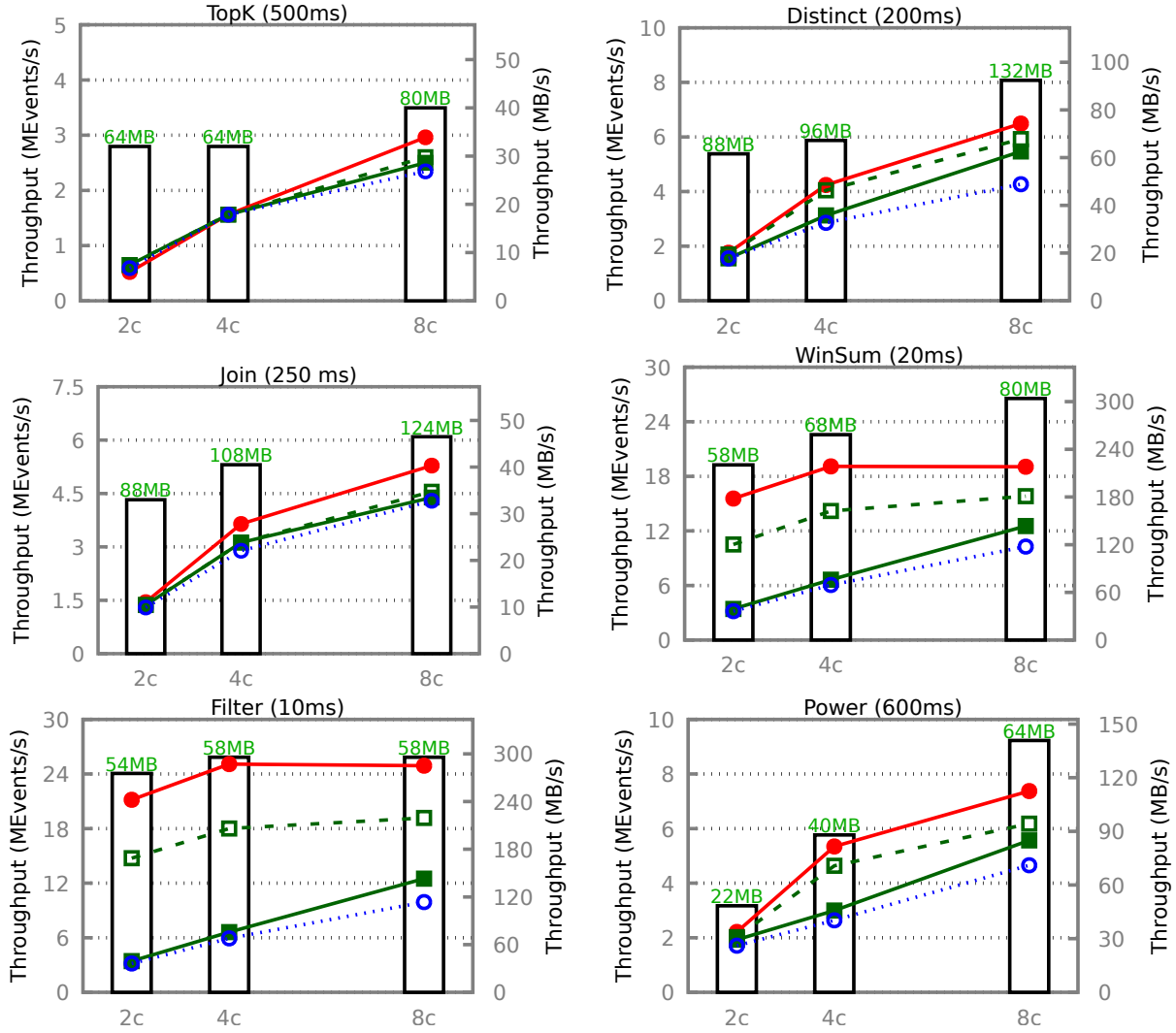


Figure 2.8. StreamBox-TZ throughput (lines, left/right y-axes) as a function of CPU cores (x-axis) under given output delays (above each plot). Steady consumptions of TEE memory as columns with annotated values. See Table 2.5 for legends and explanations.

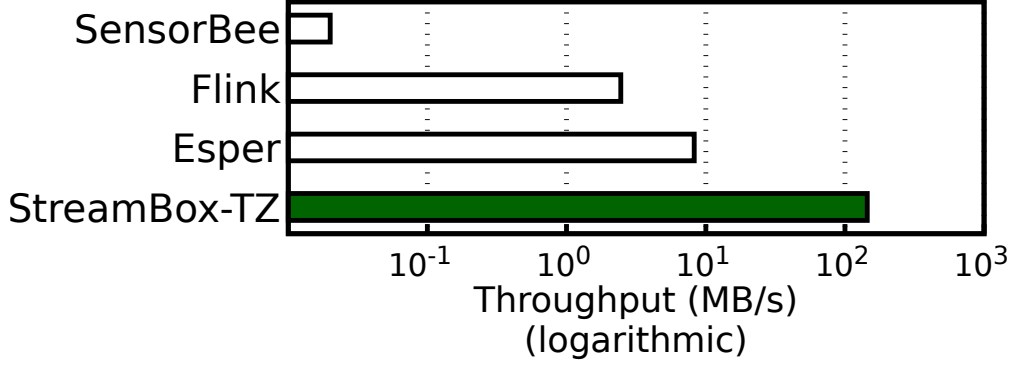


Figure 2.9. StreamBox-TZ achieves much higher throughput than commodity insecure engines [51], [57], [92] on HiKey. Benchmark: windowed aggregation; target output delay: 50ms.

i) no analytics engines use TrustZone for data protection and ii) no systems can partition an insecure stream engine for TrustZone. Note that popular secure analytics engines, e.g. VC3 [39] and Opaque [40], not only require SGX but also target batch processing instead of stream analytics. To this end, we qualitatively compare SBT with SecureStreams [41], the closest system we are aware of. Designed for an x86 cluster, SecureStreams uses SGX to protect stream operators and targets strong data security. On a benchmark similar to WinSum it was reported to achieve 10 MB/sec, one magnitude lower than SBT on WinSum. Furthermore, SecureStreams achieved such performance on a small x86 cluster which has much richer resource than HiKey: the former has faster CPUs (8x i7-6700@3.4GHz versus 8x Cortex-A53@1.2GHz), larger DRAM (16 GB versus 2 GB), higher power (130W versus 36W), and higher cost (\$600 versus \$65).

SBT’s advantage comes from i) data exchange via coherent memory inside one TEE, instead of exchanging encrypted messages among workers; ii) memory management specialized for streaming, and iii) vectorized computations.

Security overhead We investigate the overhead of the new security mechanism contributed by SBT – its isolated data plane. We assess the overhead as the throughput loss of *SBT ClearIngress* as compared to *Insecure* (i.e. native performance as StreamBox [47] invoking SBT’s stream computations), both paying same costs for data ingress. The target output delays are the same (labeled atop each plot in Figure 2.8). The security overhead

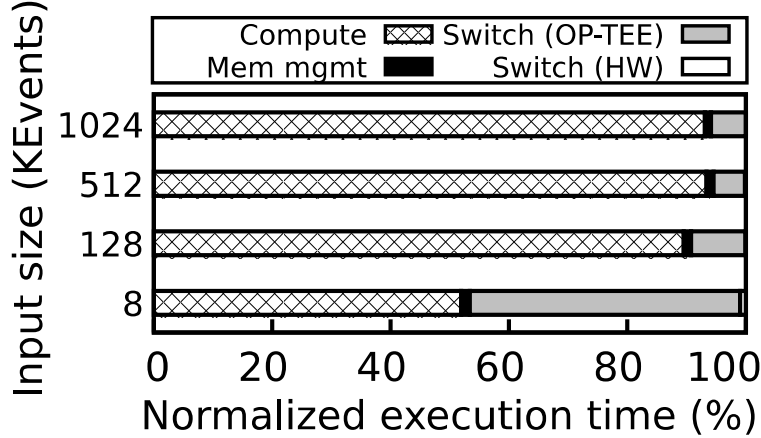


Figure 2.10. Run time breakdown of operator GroupBy under different input batch sizes. The control plane runs 8 threads to execute GroupBy in parallel. Total execution time is normalized to 100%.

is less than 25% in all benchmarks. This is similar to or lower than the reported overhead (20–70%) in recent TEE systems [36], [37], [87].

Overhead analysis The security overhead mostly comes from world switch, among operators and inside each operator. To understand the switch cost within an operator, we profile GroupBy, one of the most costly operators. We test different input batch sizes, which have a strong impact on TEE entry/exit rates and hence isolation overhead (§2.4). Figure 2.10 shows a run time breakdown. When each input batch contains 128K (close to the value we set for SBT) or more events, more than 90% of the CPU time is spent on actual computations in TEE. The CPU usage of TEE memory management is as low as 1–2%. In the extreme case where each input batch contains as few as 8K events, the overhead of world switch starts to dominate. Most of the world switch overhead comes from OP-TEE instead of the CPU hardware (a few thousand cycles per switch), suggesting room for OP-TEE optimization.

Impact of decrypting ingress data Decrypting ingress data is needed if source-edge links are untrusted (§2.3) and source must send encrypted data. It has substantial performance impact. By comparing SBT to *SBT ClearIngress*, turning on/off ingress decryption leads to 4% – 35% throughput difference when all 8 cores are in use. The performance gap is more pronounced for simple pipelines, which has higher ingestion throughput leading to higher decryption cost.

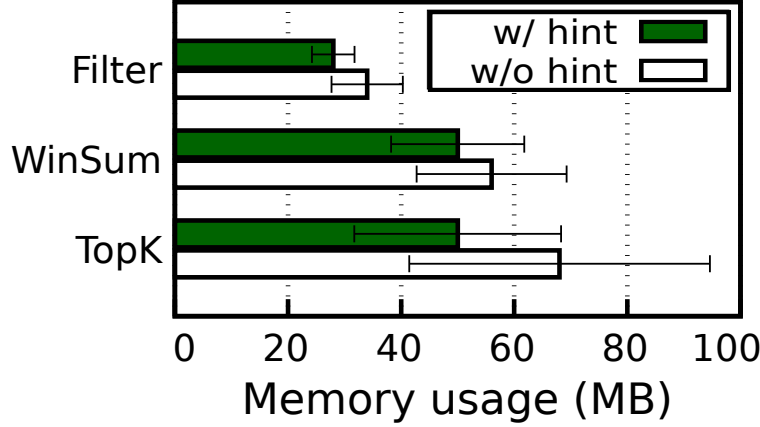


Figure 2.11. Without consumption hints, the allocator uses more TEE memory. Since memory usage fluctuates at run time, the error bars show two standard deviations below and above the average.

TEE memory usage While sustaining high throughput, SBT consumes a moderate amount of physical memory, ranging from 20 MB to 130 MB as shown in Figure 2.8. The memory usage is as low as 1–6% of the total system DRAM. The virtual memory usage is also low, often 1–5% of the entire virtual address space in OP-TEE. The memory usage increases with the throughput, since there will be more in-flight data. On the same platform, Flink’s memory consumption is 3× higher, due to its hash-based data structures and the use of JVM. This validates our choice of uArrays.

Attestation overhead Attestation incurs minor overhead to both the edge and the cloud. We measured that SBT produces 300–400 audit records per second across all our benchmarks, and spends a few hundred cycles on producing each record. Compressing such record streams on HiKey consumes 0.2% of total CPU time. Our consumer written in Python on a 4-core i7-4790 machine replays 57K records per second with a single core, suggesting a capability of attesting near 500 SBT instances simultaneously. We will evaluate the efficacy of record compression in Section 2.9.3.

2.9.3 Validation of Key Design Features

Exploitation of trusted IO As shown in Figure 2.8, a comparison between SBT and *SBT IOviaOS* demonstrates the advantage of directly ingesting data into TEE and bypassing

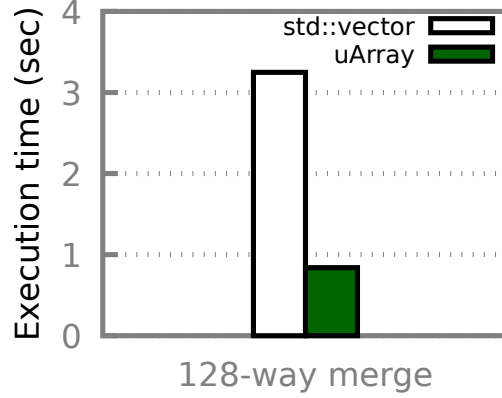


Figure 2.12. On-demand growth of uArrays vs. `std::vector`

the OS: SBT outperforms the latter by up to 20% in throughput due to reduction in moving ingested data.

Trusted primitive vectorization (§2.5) Our optimizations with ARM vector instructions are crucial. To show this, we examine `GroupBy`, one of the top hotspot operators. When we replace the vectorized Sort that underpins `GroupBy` with two popular implementations (`qsort()` from the the OP-TEE’s `libc` and `std::sort()` from the standard C++ library), we measured the throughput of `GroupBy` drops by up to $7\times$ and $2\times$, respectively. We have similar observation on other operators.

Efficacy of hint-guided memory placement (§2.6.2) We compare to an alternative design: the modified allocator acts based on the heuristics that all the uArrays produced by the same primitive belong to the same *generation* and are likely to be reclaimed altogether. Accordingly, the modified allocator places these uArrays in the same uGroup. As shown in Figure 2.11, in three benchmarks, the modified allocator increases memory usage by up to 35%. This is because, without hints, it cannot place uArrays based on future consumption.

uArray on-demand growth (§2.6.1) We compare uArray to `std::vector`, a widely used C++ sequence container with on-demand growth. We run a microbenchmark of N-way merge, an intensive procedure in trusted primitives. It iteratively merges 128 buffers (uArrays or vectors), each containing 512 KB (128K 32-bit random integers) until obtaining a monolithic buffer; as merge proceeds, buffers grow dynamically. As shown in Figure 2.12,

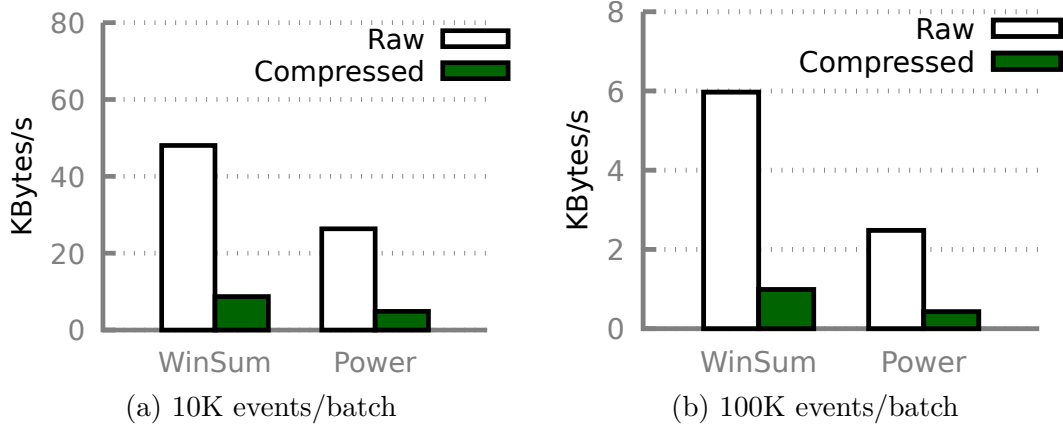


Figure 2.13. Compression of audit records saves uplink bandwidth substantially.

uArrays is $4\times$ faster than `std::vector`, because the allocation and paging in TEE that back uArray growth is much faster than that of a commodity OS.

Compression of audit records (§2.7) The compression significantly saves the uplink bandwidth. We test two benchmarks (WinSum and Power) on two extremes of the spectrum of computation cost, and test two very different input batch sizes. This is because simpler computations and smaller batch sizes generate audit records at higher rates. Figure 2.13 shows that SBT compresses audit records by $5\times$ – $6.7\times$. In an offline test using gzip to compress the same records, we find our compression ratios are $1.9\times$ higher than gzip. 2–40 KB/sec of uplink bandwidth is saved, which is significant compared to the uploaded analytics results, which are 144 bytes/sec for WinSum and 400 bytes/sec for Power.

2.10 Related Work

Secure data analytics DARKLY [114] protects sensor data by isolating computations in an OS process, resulting in a large TCB. VC3 [39] and SecureStreams [41] use SGX to protect the operators in distributed analytics. They lack optimizations for parallel execution in one TEE on the edge. To process data confidentiality, STYX [115] computes over encrypted data, a method likely prohibitively expensive to edge platforms. Opaque [40] protects data access patterns of distributed operators, targeting a threat out of our scope.

TCB minimization Minimizing TCB is a proven approach towards a trustworthy system. Flicker [64] directly executes security-sensitive code on baremetal hardware. Trustvisor [63]

shrinks its TCB to a specialized hypervisor. Sharing a similar goal, SBT addresses unique challenges in supporting data-intensive computation on a minimal TCB.

Trusted Execution Environments Much work isolates security-sensitive software components. Terra [116] supports isolation with a virtual machine. Many systems used TrustZone and SGX [50] for TEE. Some systems enclose in TEE whole applications [9], [36]–[38], while others partition existing programs for TEE [87]–[89]. These approaches often result in larger TCBs and/or higher overhead than SBT and are thus less desirable for SBT. TEE also sees various novel usage, including protecting mobile app classes [117], enforcing security policies [118], remote attestation of application control flows [119], and controlling data access [120]. None addresses data-intensive computation as SBT does.

Edge processing evolves from a vision [20], [21] to practice [28]–[30]. Most works focused on programming paradigms [121], developing and deploying application [27], [122], [123], and resource management [124]. Complementary to them, SBT focuses on secure analytics on the edge.

Stream processing systems, in response to big data challenges, evolve from single-threaded [125]–[129] to massive parallel systems [46], [53]–[55], [130], [131]. The existing systems focus on challenges, such as fault tolerance [46], fast reconfiguration [132], high parallelism [47], [48], and the use of GPUs [111]. Few systems achieve data security and performance simultaneously as SBT does.

2.11 Conclusions

This paper presents StreamBox-TZ (SBT), a secure stream analytics engine designed and optimized for a TEE on an edge platform. SBT offers strong data security, verifiable results, and competitive performance. On an octa core ARM machine, SBT processes up to tens of millions of events per second; its security mechanisms incur less than 25% overhead.

3. GPURIP: A 50-KB GPU STACK FOR CLIENT ML

3.1 Introduction

GPU stacks Smartphones or IoT devices commonly use GPUs to accelerate machine learning (ML). As shown in Figure 3.1(a), a modern GPU software stack spans ML frameworks (e.g. Tensorflow [133] and ncnn [134]), 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 has a large codebase. The runtime for Arm Mali, reported to be the most pervasive GPUs in the world [135], is a 48-MB executable; their driver has 45K SLoC [136]. The stack often has substantial proprietary code and undocumented interfaces.

Such a sophisticated GPU stack has created a number of difficulties. (1) Weak security [137]–[139]. 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) Difficult deployment. 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 [140]. Even on a supported OS, an ML app often only works with specific combinations of runtime/kernel versions [141]–[143] (3) Slow startup. Even launching a simple GPU job may take several seconds because of expensive stack initialization. This paper will show more details.

The complexity of a GPU stack is from its original design goal: to support interactive apps which generate numerous dynamic GPU jobs. By contrast, ML apps often run a *prescribed* set of GPU jobs (albeit on new input data) [144]; many ML apps run GPU job batches without UI; 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 GPURip 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 3.1(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 at a narrow interface – registers and memory, which

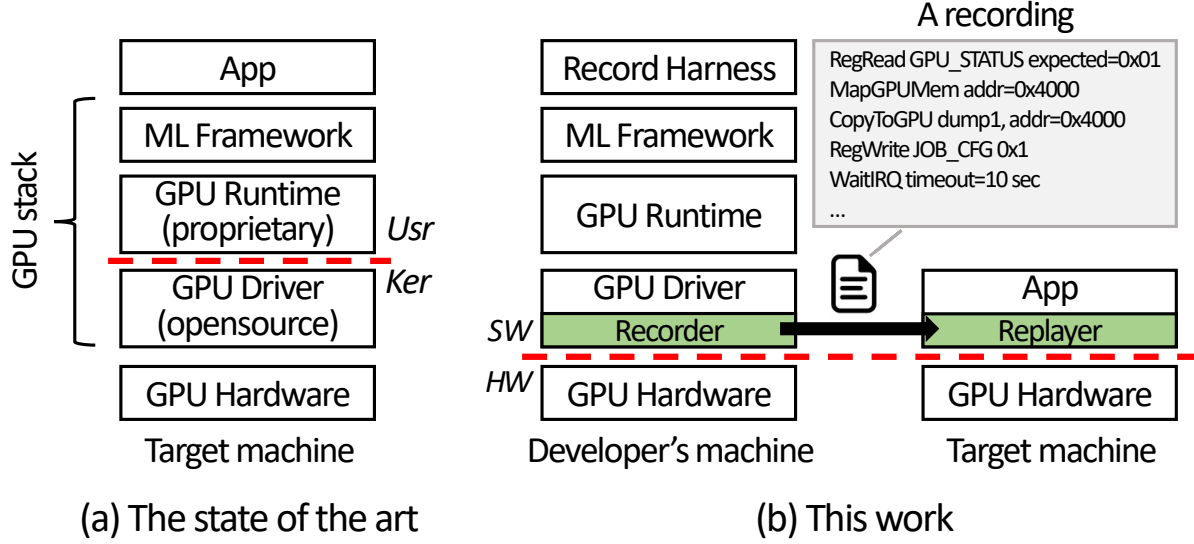


Figure 3.1. The overview of GPURip

is managed by an open-source driver. Through lightweight instrumentation, a recorder in the driver 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 the specified time intervals. Throughout the process, the recorder/replayer remain oblivious to the semantics of most register accesses and memory dumps.

Benefits GPURip offers the following benefits:

(1) *Security* First, GPURip better shields the GPU stack. The GPU stack serving ML apps is detached from target machines and instead resides on developer’s machine. The stack is therefore no longer exposed to many threats in the wild; it is protected as part of software supplychain, for which attacks require much higher capabilities and longer commitment [11]. Second, as the GPU stack is replaced with a simple replayer, the target machines have fewer vulnerabilities, e.g. kernel crash due to invalid GPU buffer [12]. Third, the replayer serves

as a strong *last line of defense* for GPU security: it only has a few K SLoC; it is independent of a commodity OS; it can be isolated within a TEE. See Section 3.7 for a security analysis.

(2) *Ease of ML deployment* The replayer runs in various environments: at user or kernel level of a commodity OS, in a TEE, in a library OS, and even baremetal. Section 3.6 will present the details. GPURip brings mature GPU compute such as Tensorflow NNs to these environments without porting full GPU stacks. GPURip is compatible with today’s GPU ecosystems. It requires no reverse engineering of proprietary GPU runtimes, commands, and shaders. Agnostic to GPU APIs, GPURip can record and replay diverse ML workloads.

(3) *Faster GPU invocation* GPURip 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 GPURip’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. GPURip’s approach to nondeterminism sets it apart from prior record-and-replay systems [145]–[147]: 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 GPURip requires low developer efforts, and such efforts are often amortized over a family of GPUs supported by one driver. We explore GPURip’s deployment ranging from smartphones to headless IoT devices. We investigate how to map an ML workload to GPURip recordings and quantify the impact of recording

granularities. We propose a scheduling mechanism for the replayer to share GPU with interactive apps.

Results GPURip works on a variety of GPUs (Arm Mali and Broadcom v3d), APIs (OpenCL, GLES compute, and Vulkan), ML frameworks (ACL [148], ncnn [134], TensorFlow [133], and DeepCL [149]), 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 40% lower to 13% higher.

This paper makes the following contributions:

1. GPURip, as 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.

3.2 Motivations

3.2.1 The GPU stack and its problems

CPU/GPU interactions As shown in Figure 3.2, 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 memory¹. 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.

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 [150], the v3d GPU executes 32K jobs. A game may rewrite shader sources for jobs [151]. Unable to foresee these jobs, the GPU stack generates their commands and shaders just in time.

¹↑GPU memory for short, with the understanding it is part of shared DRAM

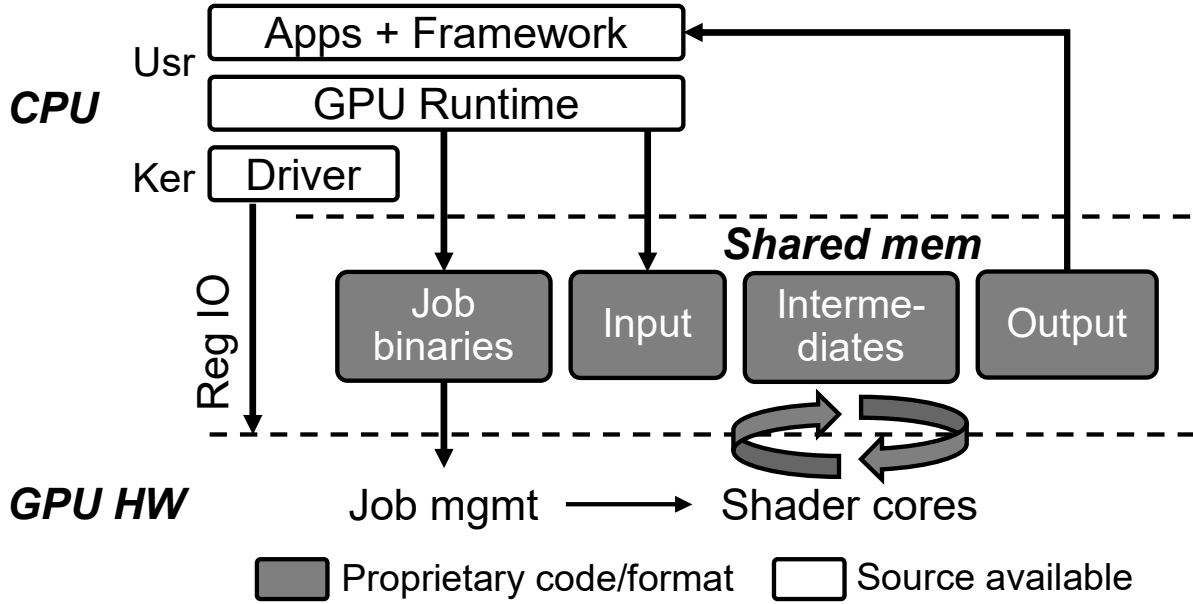


Figure 3.2. The software/hardware for an integrated GPU

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 [152].

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 [144], requesting a smaller set of GPU jobs repeatedly executed on different inputs. Popular neural networks (NN) often have tens of GPU jobs each (§3.7). 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. photo beautification or model retraining. 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 GPURip 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.

3.2.2 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.

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 the Mali [153] and one in v3d/vc4 [154], [155]). We confirm the low overhead of synchronous jobs: with six NN inferences on Mali G71 (see Table 3.5 for details), we find that enforcing synchronous jobs only adds 4% delays on average (max: 11%, min: 2%).

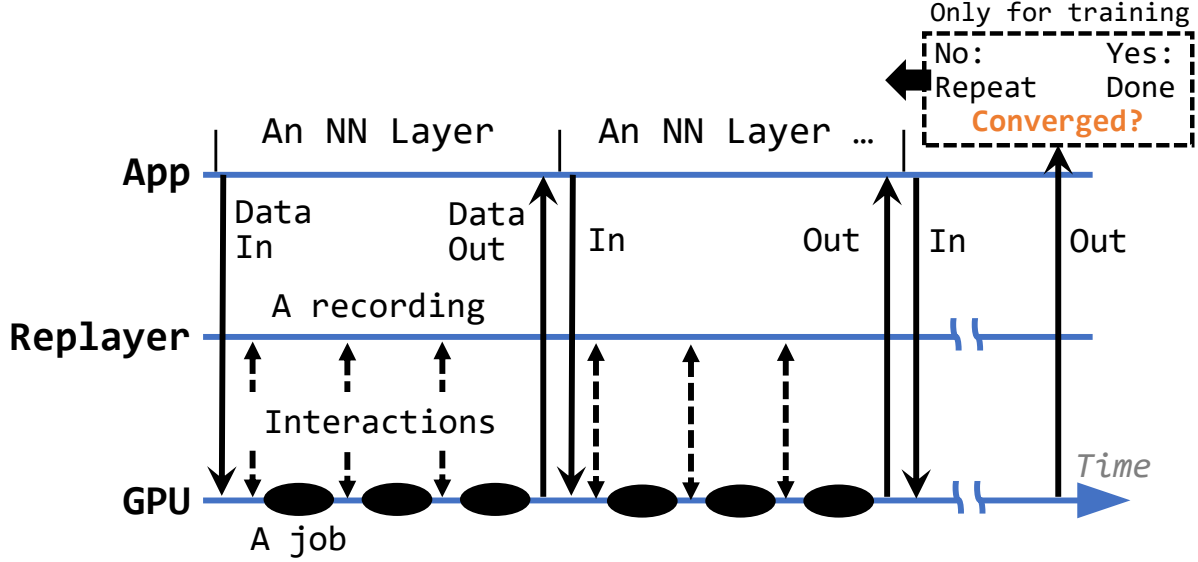


Figure 3.3. Replaying NN execution with GPURip

3.3 GPURip

3.3.1 Using GPURip

The boundary We record 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 [156] or ML frameworks [157], would require the replayer to incorporate extensive runtime or driver functionalities.

A recording encodes a fixed sequence of GPU jobs, including the CPU/GPU interactions (in an XML file) and GPU memory dumps (in ELF files) 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 GPURip 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 [134], [148], [158]. 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 3.3 shows two common NN workloads.

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

The only exception to the above requirements, to our knowledge, is a conditional NN [159] 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.

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 3.7 will evaluate these choices.

Developers’ 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 models supported by the driver. See Section 3.4 for examples. (2) Recording their ML workloads. The effort is *per GPU model*. With minor patches, a recording can further be shared across GPU models of the same family. Section 3.6

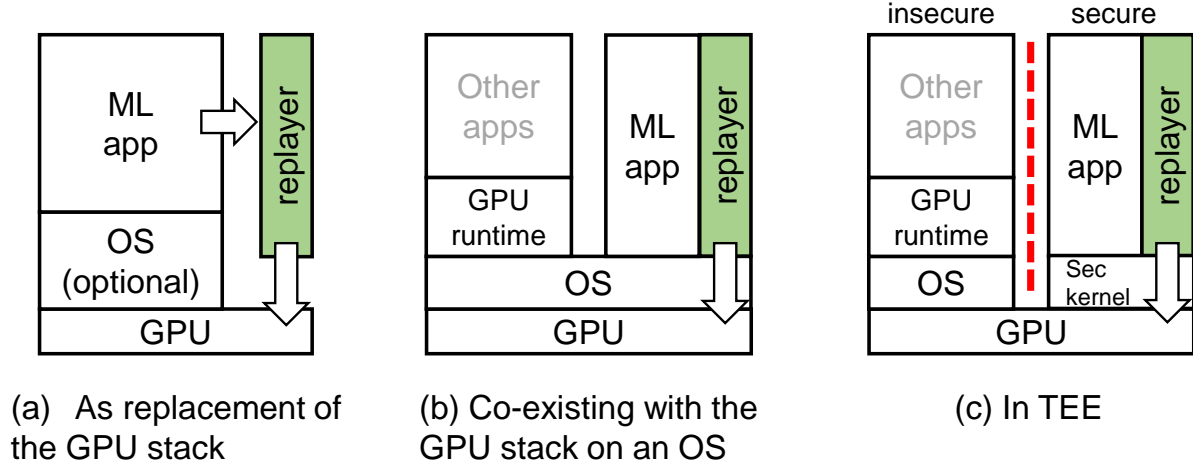


Figure 3.4. The different ways to deploy a GPURip replayer

describes our experiences. (3) Building a replayer. The effort is a few K SLoC *per deployment environment*, e.g. for a TEE.

Deploying the replayer Figure 3.4 shows three scenarios.

- *As a replacement for the GPU stack.* This applies to headless devices such as robots, where ML apps share GPU cooperatively. Each ML app run its own replayer instance.
- *Co-existing with a GPU stack on the same OS.* This applies to smartphones. Interactive apps run on the GPU stack as usual. When they are not using GPU, the OS runs ML with replay. Once the interactive apps ask for GPU, the OS preempts GPU from the ongoing replay with short delays (Section 3.5).
- *In TEE.* This applies to Arm TrustZone [160]. On the same device, the GPU stack runs in the normal world and ML runs atop a replayer in the secure world. A secure monitor at EL3 switches GPU between the two worlds. A replayer in an SGX enclave is possible, but would need additional support such as MMIO remoting or SGX’s extension for MMIO [161] because by default enclaves cannot directly access GPU registers.

Section 3.7 presents a security analysis for each scenario.

Table 3.1. 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 [136]	Y	Y	[163]	[153]	[164]	[165]	[166]
Bcom v3d [154]	Y	Y	NC	[167]	[168]	[169]	[170]
Bcom vc4 [171]	Y		NC	[155]	N/A	[172]	[173]
NV TegraX1 [174]	Y	Y	[175]	[176]	[177]	[178]	[179]
Qcom Adreno [180]	Y	Y	[181]	[182]	[183]	[184]	[185]

3.3.2 The GPU Model

GPURip builds on a small set of assumptions as summarized in Table 3.1. 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 [162]. 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 3.2.2.
- *GPU virtual memory.* The replayer can manipulate the GPU page tables and load memory dumps to physical addresses of its choice. GPURip *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 $\langle P, C, J \rangle$ 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 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 *golden*, 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 timing-related behaviors. The recorder discovers and summarizes them as replay actions, so that the replayer can tolerate (§3.4). (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) Af-

fecting 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.

3.4 Record

3.4.1 Interface Knowledge and Instrumentation

The knowledge needed by the recorder is in Table 3.1:

- 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 [186] or tracepoints [187]. We find manual instrumentation is more robust than tracing via page faults [188].

Developer efforts The efforts to extract interface knowledge and to instrument a driver are often shared by a family of GPU models supported by the driver. We confirm this is true for 6 GPU models supported by the Arm Bifrost driver [136] and 17 GPU models supported by the Adreno 6xx driver [180]. Although a driver may execute code conditionally depending on the GPU model in use, the GPU interfaces in a GPU family, i.e. register names and semantics, are often identical.

Table 3.2. Replay actions in a recording

Replay Actions	Descriptions
RegReadOnce (r,val,ignore)	Read register @r once. A return value \neq @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.

3.4.2 Register access

A recording consists of actions listed in Table 3.2. 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 [189], [190], 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 3.2. We tap in existing macros such as `wait_for()` [191], [192] and instrument tens of callsites per driver.

3.4.3 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, GPURip dumps all memory regions that may contain the job binary; to replay, GPURip 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 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 3.6](#) presents hardware-specific optimizations to further shrink memory dumps.

3.4.4 Locating input and output for a recording

Record by value vs. by address A recording accepts one or more input buffers. By default, GPURip 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 intent to reuse an input buffer’s values for replay, they may optionally annotate the input as “record by value” in the record harness. GPURip 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 does 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, GPURip adopts 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.

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, GPURip 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 **tupni**, 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.

3.4.5 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 [193], [194] and delay after requesting GPU to flush cache [195].

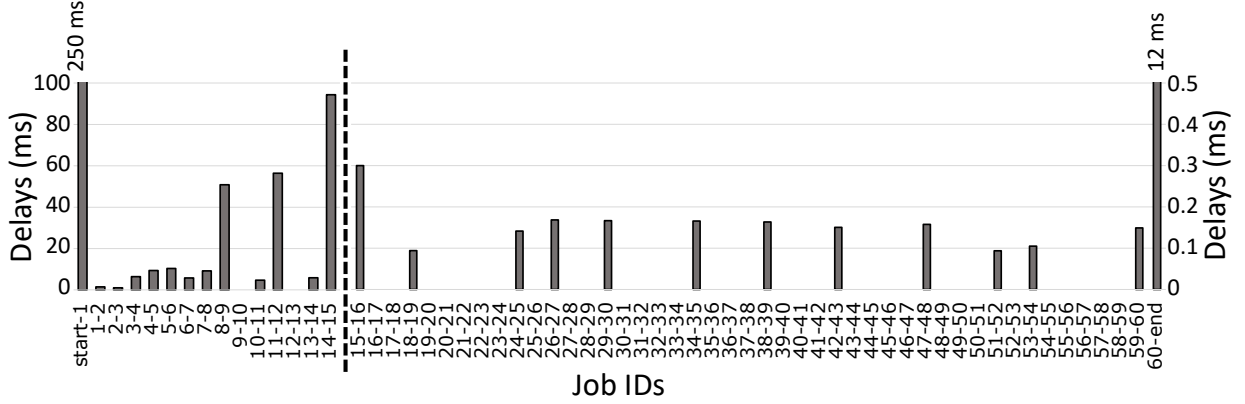


Figure 3.5. Intervals between CPU/GPU interactions, accumulated by GPU job. Intervals among earlier jobs are longer than later ones. Workload: AlexNet inference. ACL [148] on Mali G71. Excluded: GPU busy time; parameters loading IO

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 3.5 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 3.7. The recorder simply preserves the remaining intervals for replay.

3.5 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.

3.5.1 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 3.2) 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 3.7.1 will present discussions.

3.5.2 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 interrupt, acknowledging an interrupt, and checking interrupt sources, is done implicitly by replaying the corresponding actions.

3.5.3 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 3.7 evaluates preemption and checkpointing experimentally.

3.5.4 Handling replay failures

Replay failures are GPU state divergences due to non-preventable nondeterminism at run time. Based on our GPU model (§3.3), 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.

Table 3.3. GPURip implementations. * = used in evaluation. See Table 3.5 for evaluated recordings

GPU HW (Boards)	Compatible GPU stacks	Recordings	Replayers
Mali-G71 * (Hikey960)	1. ACL + Open CL *	Inference: 18 Training: 1	1.User* 2.TEE
Mali G51 (Odroid N2)	2. DeepCL + OpenCL *		
Mali G31 (Odroid C4)	3. ACL + GLES compute		
	4. Tensorflow + ACL + OpenCL Driver: Arm Mali r23p0-01re10		
Brcm v3d * (Raspberry Pi 4)	1. ncnv + Vulkan * 2. Py-videocore6 Driver: drm/v3d in Linux 5.11	Inference: 15 Math: 2	1.Kernel* 2.Baremetal

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, documents, 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.

3.6 Implementations and Experiences

As summarized in Table 3.3, we implement GPURip for Arm Mali (reported to ship billions of devices [135]) 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 as an experimental runtime fully written in Python).

3.6.1 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 3.7. Leveraging ArmNN [196], our prototype for Mali is compatible with TensorFlow NN models. We adds 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.

3.6.2 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 [197].

3.6.3 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 [197] for recording decompression (about 9 KB) and a baremetal library [198] 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 [199]. Linux configures power and clocks by accessing various registers, sometimes communicating with the SoC firmware [200]. 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 [201], [202]. 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.

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).

Table 3.4. Codebase comparisons. Binaries are stripped.

GPU	The original stack			Ours	
	ML Framework	Runtime	Driver	Rec	Replayer
Mali Bifrost	<ul style="list-style-type: none"> • ACL: 500 KSLoC, 30MB • DeepCL: 18 KSLoC, 2.1MB 	libmali.so: 48 MB	45K SLoC	0.7K SLoC	<ul style="list-style-type: none"> • Usr+kernel: 2.2+0.6 KSLoC 25KB+20KB • In-TEE: 1K SLoC, 10 KB
Bcm v3d	<ul style="list-style-type: none"> • ncnn: 223 KSLoC, 11 MB 	libvulkan_broadcom.so: 7 MB	3K SLoC	1K SLoC	<ul style="list-style-type: none"> • Kernel only: 1K SLoC; 107 KB (whole driver) • Baremetal: 4K SLoC, 50 KB

3.6.4 Reusing recordings across GPU models

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 of 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 3.7.5 reports replay performance of a patched recording.

3.7 Evaluation

We evaluate GPURip with the following questions.

- Does GPURip 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?

3.7.1 Analysis

Semantic bugs, e.g. emission of wrong GPU commands, may preexist in the GPU stack for recording. Such bugs may propagate to the target machines, resulting in wrong replay results. GPURip neither mitigates nor exacerbates these bugs. Fortunately, semantic bugs are rare in production GPU stacks to our knowledge. GPURip’s recorder and replayer may introduce semantic bugs. The chance, however, is slim: they are small as a few K SLoC with simple logic. Our validation experiments in Section 3.7.2 strengthen our confidence. We next focus on security, a major objective of GPURip.

Threat models Corresponding to three deployment scenarios in Figure 3.4: (1) a replayer on a commodity OS (at the user or kernel level) trusts the OS while facing local unprivileged and remote adversaries; (2) a replayer in TEE trusts the TEE kernel while facing the local OS adversaries and remote ones; (3) 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 [11]. We will nevertheless discuss the consequences of such attacks.

Security advantages over existing deployment (1) The GPU stack is better shielded on developers’ machines. (2) On target machines, the GPU stack is replaced by the replayer with fewer vulnerabilities. Many vulnerabilities of a GPU stack originate in rich features such as buffer management [12], [203] and fine-grained sharing [204]–[206]; they also come from complex interfaces such as framework APIs [13], IOCTLs [207], and directly mapped memory [208]. By comparison, the replayer does not have such features. It exposes only four simple functions. All replay actions have simple, well-defined semantics and are amenable to checks. (3) While both the replayer and the GPU driver act as the last lines of defense, the replayer’s defense offers stronger guarantees: its smaller code sizes, as shown in Table 3.4, ease an exhaustive verification; it can be protected in TEE and against the OS.

Thwarted attacks (1) When a replayer completely replaces the GPU stack on an OS, the whole kernel is free from GPU stack vulnerabilities that cause kernel information disclosure [204], kernel crash [12], and kernel memory corruption [203]. (2) When a replayer

coexists with the GPU stack on an OS, the app using the replayer is free vulnerabilities of the GPU runtime which cause unauthorized access to app memory [208], arbitrary code execution in the app [13], and app hang [209].

Attacks against GPURip (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 (§3.5.1). (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.

3.7.2 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 3.5 (a) 1,000 times each. We create random input, inject interference, and

Table 3.5. 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	Zippped
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	Zippped
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

compare the GPU’s outcome with the reference answers computed by CPU. The replayer always gives the correct results.

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.

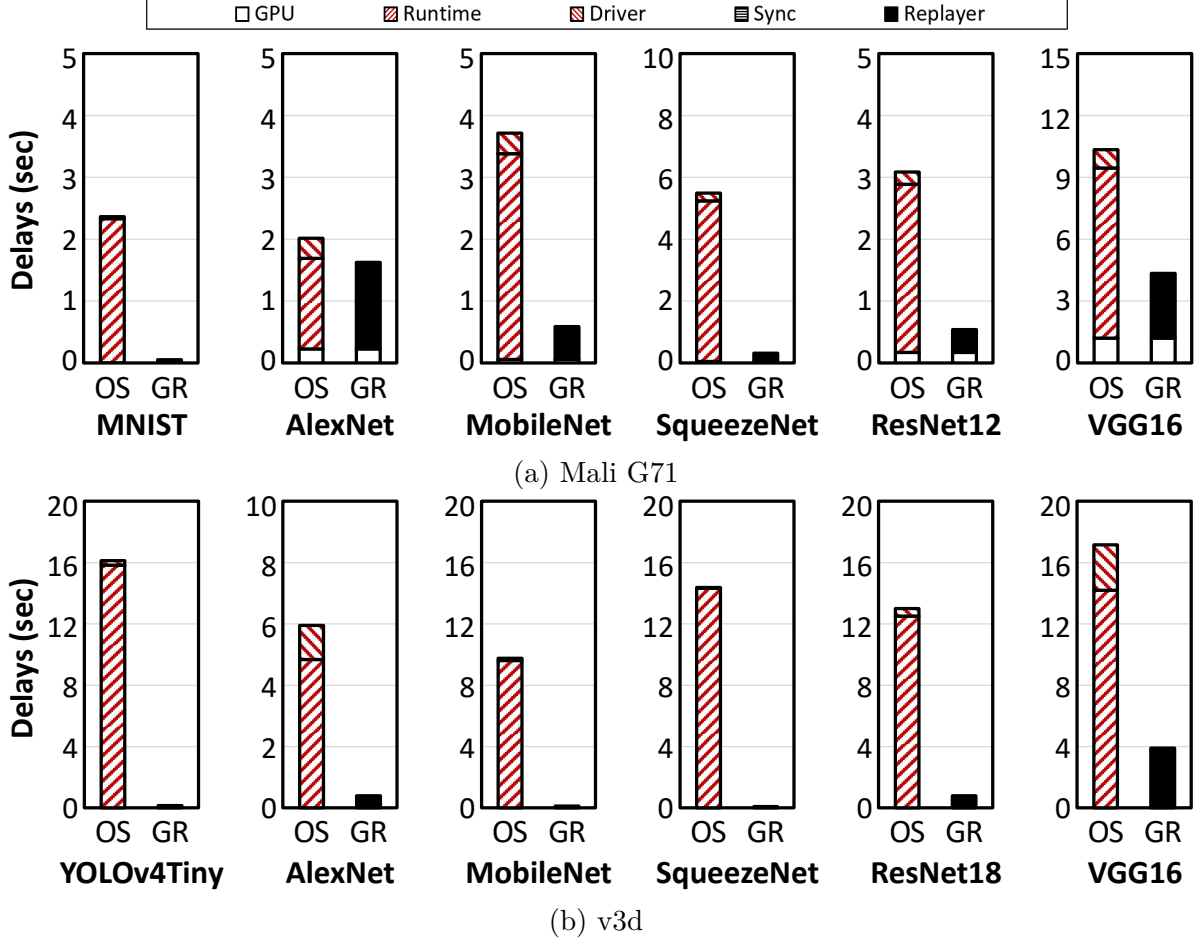


Figure 3.6. Startup delays prior to NN inference. The replayer (GR) takes much less time than the original GPU stack (OS).

3.7.3 Memory overheads

Recording sizes A GPU recording is as small as a few hundred KBs when compressed as shown in Table 3.5. The size is a small fraction of a smartphone app, which is often tens of MBs [210]. 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

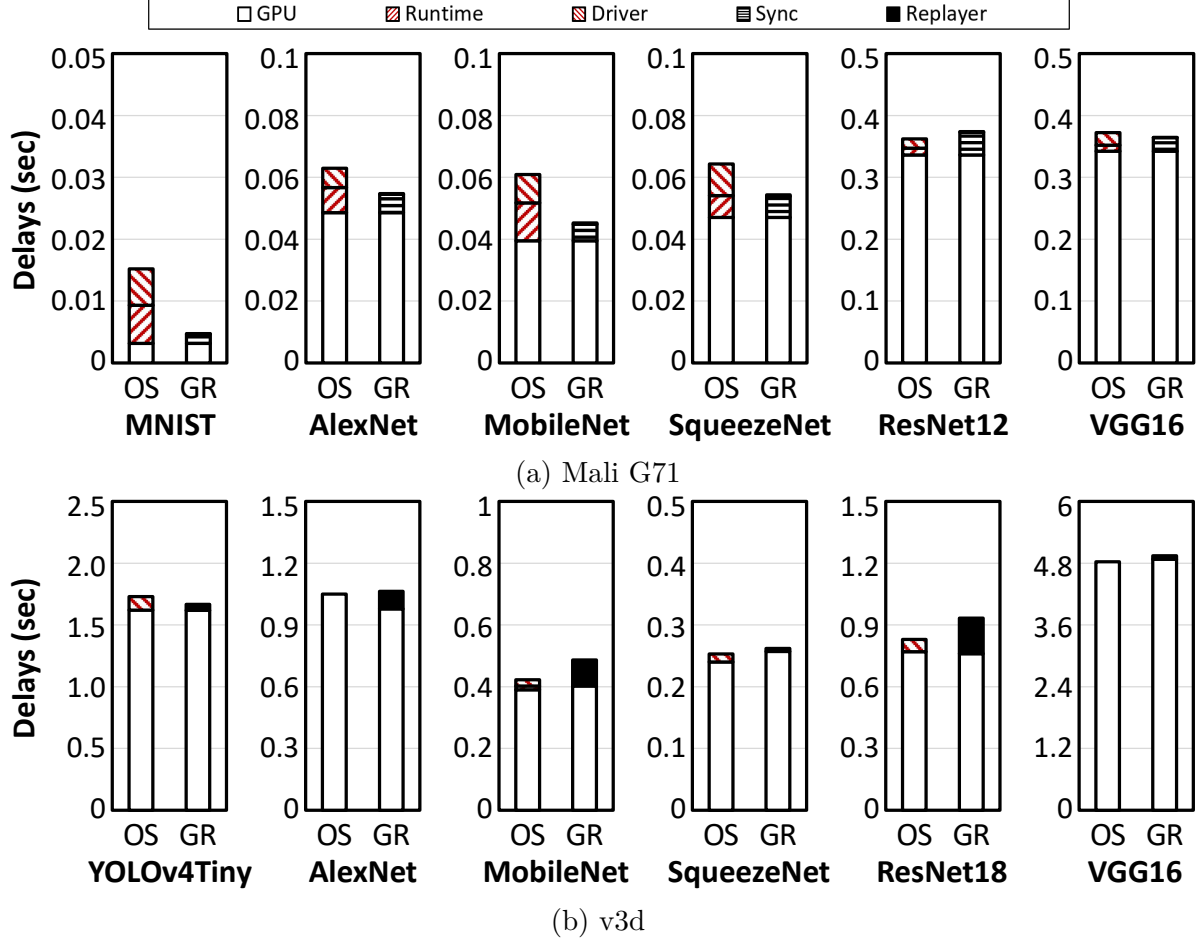


Figure 3.7. NN inference delays. The replayer (GR) incurs similar delays as compared to the original GPU stack (OS).

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.

3.7.4 Replay speed

We study the inference delays on a variety of NNs as listed in Table 3.5. Compared to the original GPU stacks, the replayer’s startup delays are significantly lower: by 26% – 98% (Mali) and lower by 77% – 99% (v3d); the replayer’s execution delays are similar: ranging from 44% lower to 13% higher (Mali) and from 4% lower to 19% higher (v3d).

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 3.6 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 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 [211] or built NN pipelines [212]. Compared to these point solutions, GPURip 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 3.7. 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, GPURip sees diminishing advantages and sometimes disadvantages. GPURip’s major overheads are (1) loading of memory dumps containing unneeded data that GPURip 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 GPURip shows similar advantages. Our benchmark is MNIST with DeepCL [149] 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 3.8, 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.

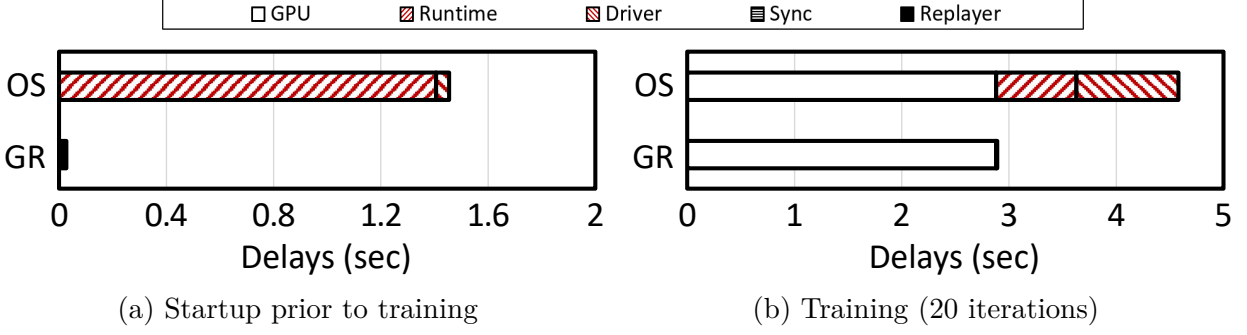


Figure 3.8. NN training delays. Benchmark: MNIST training atop DeepCL + OpenCL on Mali G71 (OS: orig stack; GR: GPURip).

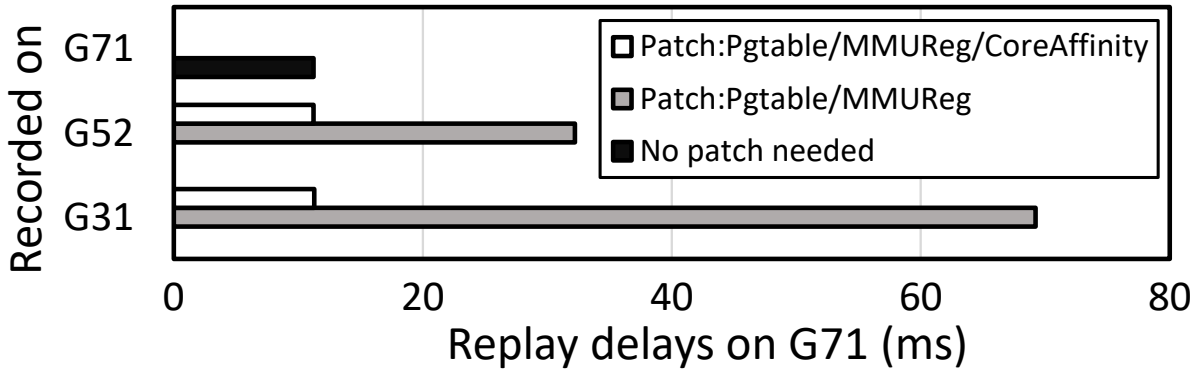


Figure 3.9. Mali G71 can replay recordings from other GPUs at full hardware speed. Benchmark: 16M elements vecadd

3.7.5 Validation of Key Designs

Cross-GPU record/replay (§3.6.4) Figure 3.9 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 (§3.4.5) Without the technique, the replayer’s NN inference will be 1.1x – 4.9x longer, as shown in Figure 3.10; startup delays will be up to two orders of magnitude longer, closer to that of a full stack (not shown in the figure).

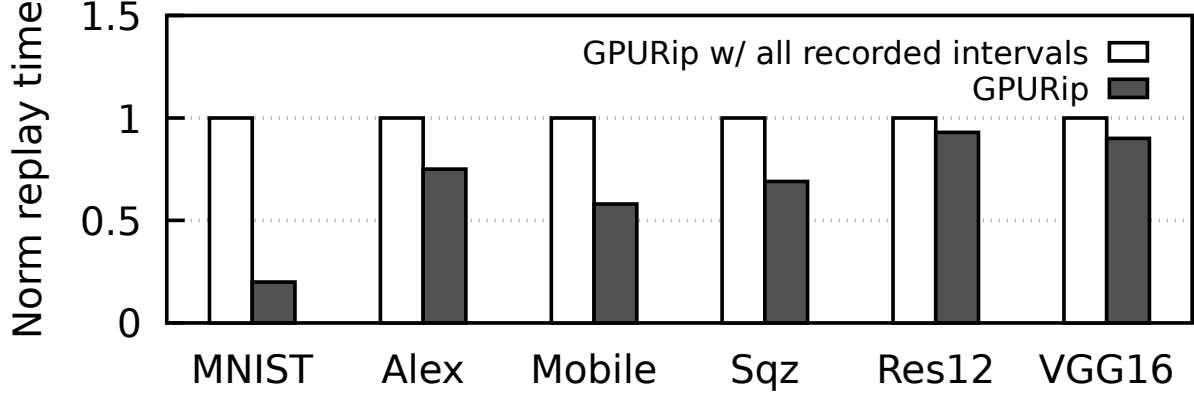


Figure 3.10. GPURip removes unnecessary intervals between replay actions. Benchmark: ACL NN inference atop Mali G71

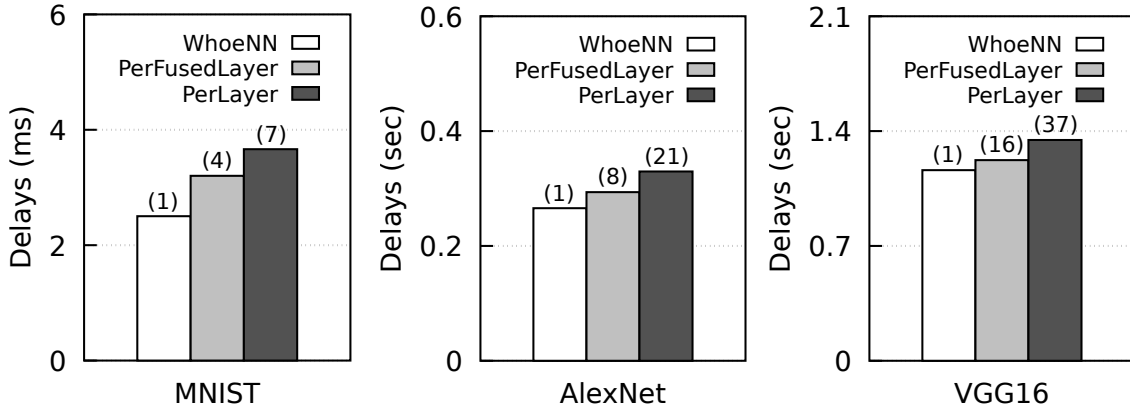


Figure 3.11. NN inference delays (including startup) with various granularities. The count of recordings is annotated.

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 [148] (a middle ground). Figure 3.11 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 3.6). We conclude that for NN inference, recording every fused layer is a useful tradeoff between composability and efficiency.

Preemption delay for interactivenss (§3.5.3) 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 (§3.5.3) 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.

3.8 Related Work

Record and replay was primarily used for diagnosis and debugging [145]–[147]. It has been applied to mobile UI apps [213], [214], web apps [215], virtual machines [216], networks [217], and whole systems [218]. None of prior work has applied the idea to the CPU/GPU interactions. Related to GPURip, Replaying syscalls and framework calls have been popular in reverse engineering GPU runtimes [219]–[222] and reducing GPU scheduling overhead [157], respectively. Unlike them, GPURip 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 [139] subsumes a full GPU stack to an app’s address space. Graviton [223] pushes the function of isolation and resource management from OS to a GPU’s command processor. Telekine [156] spans a GPU stack between local and cloud machines at the API boundary. HIX [161] ports the entire GPU stack to a secure enclave and restricts the IO interconnect. HETEE [224] 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, GPURip 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 [225] or to a remote server [226]. The interposed interfaces include GPU APIs [226], [227] and GPU MMIO [225], [228]. Notably, AvA [227] records and replays API calls during GPU VM migration. GPURip 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 **ulayer**. Notably, recent studies found CPU’s software inefficiency leaving GPU under-utilized, e.g. suboptimal CLFlush [229] or expensive data transformation [230]. While prior solutions fix the causes of inefficiency in the GPU stack [229], GPURip 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 [4], [5], [231]. They often support CPU-only compute and their workload transformation is orthogonal to GPURip. While Slalom [232] proposed secure GPU offloading, it requires GPU stack in TEE and limited to linear operations.

3.9 Concluding Remarks

Applicability to discrete GPUs The idea of GPURip is likely to apply. Our GPU hardware assumptions (§3.3.2) see counterparts on discrete GPUs albeit in different forms, e.g. registers and memory mapped via PCIe. The new challenges include more complex CPU/GPU interactions, higher GPU dynamism, and recording cost due to larger memory dumps.

Summary GPURip pre-records GPU executions for replay on new input data without a GPU stack. GPURip identifies key GPU/CPU interactions and memory states, works around proprietary GPU internals, and prevents replay divergence. The resultant replayer is tiny, portable, and quick to launch.

4. GPU ACCELERATION IN TRUSTZONE VIA SAFE AND PRACTICAL RECORDING

4.1 Introduction

GPU in TrustZone Trusted execution environments (TEE) has been a popular facility for secure GPU computation [161], [223]. By isolating GPU from the untrusted OS of the same machine, it ensures the GPU computation’s confidentiality and integrity. This paper focuses on GPU computation in TrustZone, the TEE on ARM-based personal devices. For these devices, in-TEE GPU compute is especially useful, as they often run GPU-accelerated ML on sensitive data, e.g. user’s health activities, speech audio samples, and video frames.

GPU stack mismatches TrustZone Towards isolating the GPU *hardware*, TrustZone is already capable [233], [234], which is contrast to other TEEs such as SGX. The biggest obstacle is the GPU *software* stack, which comprises ML frameworks, a userspace runtime, and a device driver. The stack is large, e.g. the runtime for Mali GPUs is an executable binary of 48 MB; it has deep dependency on a POSIX OS, e.g. to run JIT compilation; it is known to contain vulnerabilities [138], [205], [208]. Such a feature-rich stack mismatches the TEE, which expects minimalist software for strong security [88], [89], [160]. Recognizing the mismatch, prior works either transform the GPU stack [223] or the workloads [232], [235], [236]. They suffer from drawbacks including high engineering efforts and loss of compatibility, as will be analyzed in Section 4.2.

Goal & overall approach *Can the TrustZone TEE run GPU-accelerated compute without an overhaul of the GPU stack?* To this end, a recent approach called GPURip shows high promise [237]. It executes a GPU-accelerated workload \mathcal{W} , e.g. neural network (NN) inference, in two phases. (1) In the record phase, developers run \mathcal{W} on a full GPU stack and log CPU/GPU interactions as a series of register accesses and memory dumps. (2) In the replay phase, a target program replays the pre-recorded CPU/GPU interactions on new input data without needing a GPU stack. GPURip well suites TEE. The record phase can be done in a safe environment which faces low threats. After record is done *once*, replay can happen within the TEE *repeatedly*. The replayer can be as simple as a few KSLoC, has little external dependency, and contains no vulnerabilities seen in a GPU stack [205], [206], [208].

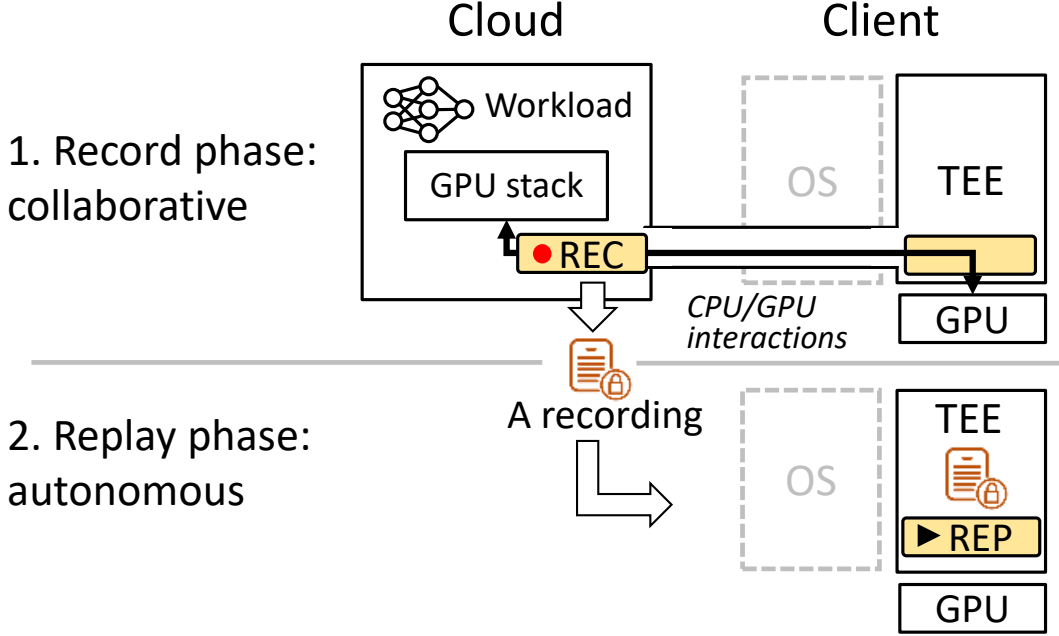


Figure 4.1. System overview. Shaded components belong to CoDry

Note that it is crucial to record and replay at CPU/GPU boundary; recording at higher levels, e.g. ML framework APIs, would bloat the TEE with implementation of these APIs.

Yet, a key, unsolved problem is the *recording environment*, where the full GPU stack is exercised and CPU/GPU interactions are logged. The recording environment must simultaneously (1) enjoy strong security and (2) access the *exact* GPU hardware that will be used for replay. These requirements preclude recording on the OS of the same mobile device, as TEE does not trust the OS. They also preclude recording on a developer’s machine, because it can be difficult for developers to predict and possess all GPU hardware models that their workloads *may* execute on. Section 4.2 will present details on today’s diverse mobile GPUs.

Key idea We present a novel approach called collaborative dryrun (CoDry), in which the TEE leverages the cloud for GPU recording. As shown in Figure 4.1, a cloud service hosts the GPU software stack without hosting any GPU hardware. To record, the TEE on a mobile device (referred to as the “client”) requests the cloud to run a workload, e.g. NN inference. The cloud exercises its GPU stack without executing the actual GPU computation; it tunnels all the resultant CPU/GPU interactions between the GPU stack and the physical GPU isolated in the client TEE. The cloud logs all the interactions as a *recording* for the

workload. In future execution of the workload on new inputs, the TEE replays the recording on its physical GPU without invoking the cloud service.

CoDry addresses the needs for a secure, manageable recording environment. First, unlike mobile devices which face high threats from numerous apps, the cloud service runs on rigorously managed infrastructures and only exposes a small attack surface – authenticated, encrypted communication with the client TEE. Importantly, the cloud service never learns the TEE’s sensitive data, e.g. ML input and model parameters. Second, the cloud service accesses the exact, diverse GPU hardware (Figure 4.3) without the hassle of hosting them. It is responsible for hosting drivers for the GPU hardware, a task which we will show as practical.

Challenges and Designs The main challenge arises from spanning CPU/GPU interactions over the connection between the cloud and the client. A GPU workload generates frequent register accesses (more than 95% are read), accesses to shared memory, and interrupts. If the GPU stack and the GPU hardware were co-located on the same machine, each interaction event takes no more than microseconds; since we distribute them over wireless connection, each event will take milliseconds or seconds. Forwarding the interactions naively results in formidable delays, rendering CoDry unusable.

To overcome the long delays, we exploit two insights. (1) The sequence of GPU register accesses consists of many *recurring segments*, corresponding to driver routines repeatedly invoked in GPU workloads, e.g. for job submission and GPU cache flush. By learning these segments, the cloud service can predict most register accesses and their outcomes. (2) Unlike IO-as-a-service [238] which must produce correct results, the cloud only has to extract *replayable interactions* for later actual executions. With the insights, CoDry automatically instruments the GPU driver code in the cloud to implement the following mechanisms.

(1) *Register access deferral*. While each register access was designed to be executed on the physical GPU synchronously, the cloud service queues and commits multiple accesses to the client GPU in a batch, coalescing their network round trips. Since register accesses are interleaved with the driver execution in program order, the cloud service represents the values of uncommitted register reads as symbols and allows symbolic execution of the driver. After

the register reads are completed by the client GPU, the cloud replaces symbolic variables with concrete register values.

(2) *Register access speculation.* To further mask the network delay of a commit, the cloud service predicts the outcomes of register reads in the commit. Without waiting for the commit to finish, the cloud allows the driver to continue execution based on the predicted read values. The cloud validates the speculation after the client returns the actual register values. In case of misprediction, both the cloud and the client leverage the GPU replay technique to rapidly rollback to their most recent valid states.

(3) *Metastate-only synchronization.* Despite physically distributed memories, the driver in the cloud and the client GPU must maintain a synchronized memory view. We reduce the synchronization *frequencies* by tapping in GPU hardware events; we reduce the synchronization *traffic* by only synchronizing GPU’s metastate – GPU shaders, command lists, and job descriptions – while omitting workload data, which constitutes the majority of GPU memory. As a result, we preserve correct CPU/GPU interactions while forgoing the compute result correctness, a unique opportunity of dryrun.

Results We build CoDry atop Arm platforms and Mali Bifrost, a popular family of mobile GPUs, and evaluate it on a series of ML workloads. Compared to naive approach, CoDry lowers the recording delays by two order of magnitude, from several hundred seconds to 10 – 40 seconds; it reduces the client energy consumption by up to 99%. Its replay incurs 25% lower delays as compared to insecure, native execution of the workloads.

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

- A novel architecture called CoDry, where the cloud and the client TEE collaboratively exercise the GPU stack for recording CPU/GPU interactions.
- 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 the device tree for VMs to probe GPU without hosting the GPU; a TEE module managing GPU for record and replay.

4.2 Motivations

4.2.1 Mobile GPUs

This paper focuses on mobile GPUs which share memory with CPU.

GPU stack and execution workflow As shown in Figure 4.4, 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.

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 pagetables, configures GPU hardware, and submits the GPU job. The GPU 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 interactions 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 pagetables, allowing them to access shared memory using GPU virtual addresses.
- GPU interrupts, which signal GPU job status.

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

4.2.2 Prior Approaches

Our goal is to run GPU compute inside the TrustZone TEE, for which prior approaches are inadequate.

Porting GPU stack to TEE One approach is to pull the GPU stack to the TEE (“lift and shift”) [6], [161]. 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 [156], [223], also see significant drawbacks: they still require high engineering efforts and sometimes even hardware modification. The ported GPU code is likely to introduce vulnerabilities to the TEE [12], [13], [208], bloating the TCB and weakens 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 [232], [239], ML workload transformation [4], [231], and result validation [240]. They lack GPU acceleration or support limited GPU operators, often incurring significant efficiency loss.

4.2.3 GPURip in TrustZone

Unlike prior approaches, GPURip provides a new way to execute GPU-accelerated compute [237]. (1) In the record phase, app developers run their ML workload *once* on a trusted GPU stack; at the driver level, a recorder logs all the CPU/GPU interactions – register accesses, GPU memory dumps which enclose GPU commands and shaders, and interrupt events. These interactions constitute a *recording* for the ML workload. (2) In the replay phase, a target app in the TEE supplies new input to the recording. The TEE does not need a GPU stack but only a simple replayer (30 KB) for interpreting and executing the logged interactions.

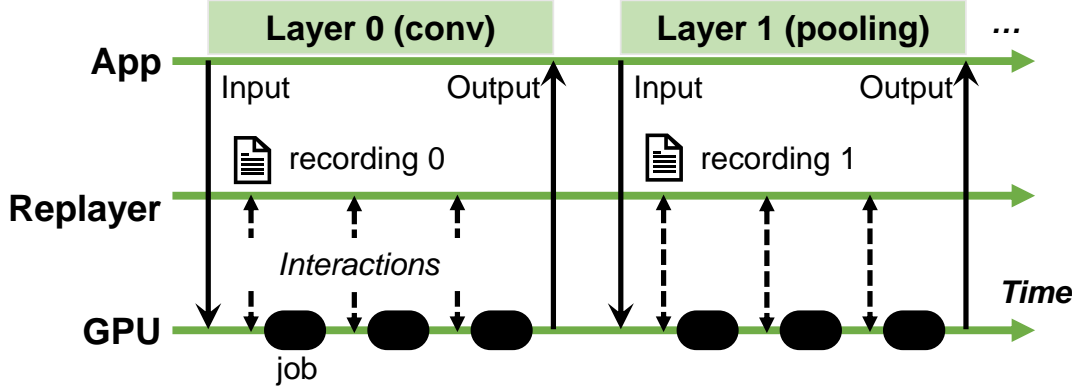


Figure 4.2. A timeline for replaying NN inference

Figure 4.2 exemplifies how GPURip works for NN inference. To record, developers run the ML inference once and produce a sequence of recordings, one for each NN layer; each NN layer invokes multiple GPU jobs, e.g. convolution or pooling. 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 is GPURip practical? (1) An ML workload such as NN often runs pre-defined GPU jobs. High-level GPU APIs can be translated to GPU primitives ahead of time; at run time, the workload does not need the stack's dynamic features, e.g. JIT and fine-grained sharing. (2) An NN often has a static GPU job graph with no conditional branches among jobs. A single record run can exercise all the GPU jobs and record them. (3) Nondeterministic GPU events can be systematically prevented or tolerated, allowing the replayer to faithfully reproduce the recorded jobs. For instance, the recorder can serialize GPU job submission and avoid nondeterministic interrupts.

4.2.4 The Problem of Recording Environment

To apply GPURip to TrustZone, a missing component is the recording environment where the GPU stack is exercised and recordings are produced. Obviously, the environment should be trustworthy to the TEE. What is more important, the environment must have access to the GPU hardware that matches the GPU for replay. Recording with the exact GPU

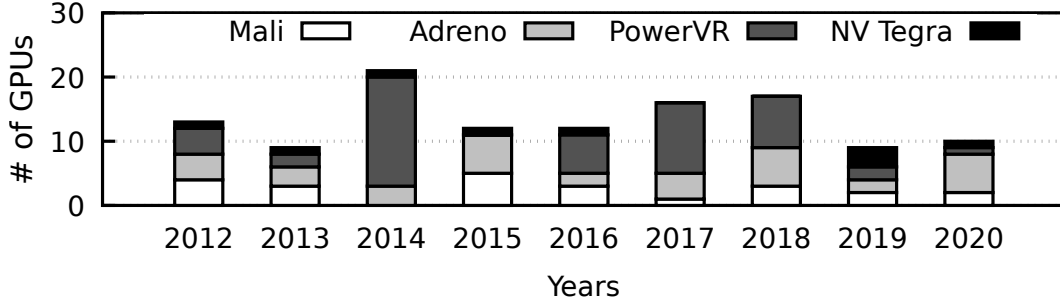


Figure 4.3. Numbers of mobile GPU models per year [241]

model is crucial. In our experience, one shall not even record with a different GPU model from the *same* GPU family, because replay can be broken by subtle hardware differences: (1) register values which reflect the GPU’s hardware configuration, e.g. shader core count, based on which the JIT compiler generate and optimize GPU shaders; (2) encodings of GPU pagetables; (3) encodings of shared memory, with which GPU communicates its execution status with CPU.

Can recording be done on developers’ machines? While developers’ machines can be trustworthy [10], it would be a heavy burden for the developers to foresee all possible client GPUs and possess the exact GPU models for recording. As shown in Figure 4.3, mobile GPUs are highly diverse [242]: today’s SoCs see around 80 mobile GPU models in four major families (Apple, PVR, Mali, and Adreno); no GPU models are dominating the market; new GPU models are rolled out frequently.

Can recording be done on a “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 tedious if not impractical. Not designed to be hosted, mobile devices do not conform to the size, power, heat dissipation 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.

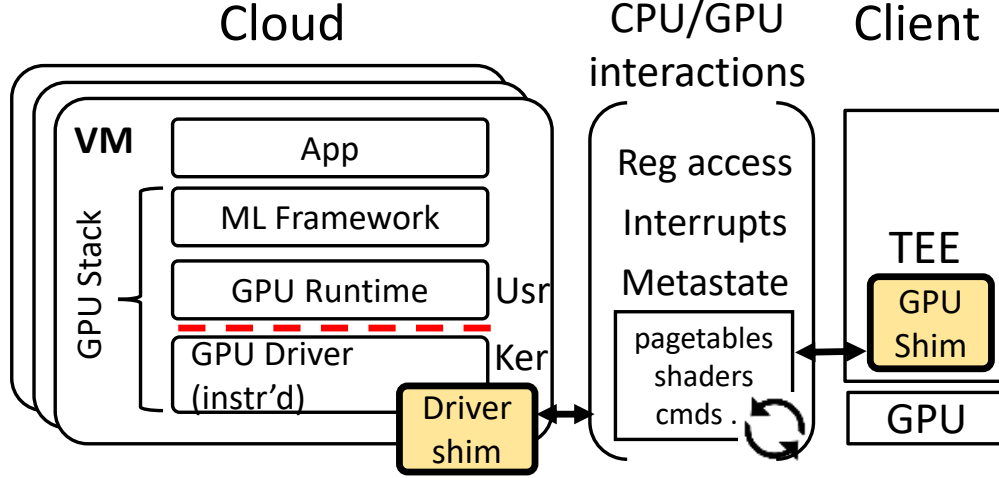


Figure 4.4. CoDry’s online recording. The cloud VM collaboratively runs GPU stack with the client GPU.

4.3 CoDry

We advocate for a new recording environment: dryrun the GPU stack in the cloud while using the physical GPUs on the clients.

4.3.1 The Approach

Figure 4.4 illustrates our approach. (1) Developers write an ML workload as usual, e.g. MNIST inference atop Tensorflow. They are oblivious to the TEE, the GPU model, and the cloud service. (2) Before executing the workload for the first time, the client TEE requests the cloud service to dryrun the workload. As the cloud runs the GPU stack, it forwards the access to GPU hardware to the client TEE and receives the GPU’s response from the latter. In the mean time, 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 data; it no longer involves the cloud.

Our approach fundamentally differs from remote I/O or I/O-as-a-service [238]. Our goal is neither to execute GPU compute in the cloud [243], [244] (in fact, the cloud has no physical GPUs) nor run the GPU stack *precisely* in the cloud, e.g. for software testing [245]. It is to

extract the software’s stimuli to GPU and the GPU’s response. This allows CoDry to skip much communications and optimize the cloud execution.

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

1. *Rich resources.* The cloud can run a GPU stack that is too big to fit in the TEE; it can also host multiple variants of GPU stack, catering to different APIs and frameworks used by ML workloads.
2. *Secure.* The cloud isolates the GPU stack in a safer environment. In contrast to client mobile devices which often run a myriad of apps and face threats such as clickbait and malware, the cloud infrastructure has more rigorous security measures [246], [247]. As the dryrun service uses dedicated VMs that only serve authenticated TEEs, the attack surface of the GPU stack is minimized.
3. *No sensitive data exposed.* A client TEE’s invocation of dryrun service never gives away its ML model weights or inputs, because recording by design does not need them. For this reason, the dryrun service does not have to be hosted in a cloud TEE, e.g. SGX. Section 4.7.1 will present a detailed security analysis.

Can the cloud emulate GPUs? One may wonder if the cloud operates with software-based GPU emulators [248], thereby avoid communicating with client GPUs. Building such emulators is difficult, as it would require precise emulation of GPU interfaces and behaviors. However, modern GPUs are diverse [242]; they often have undisclosed behaviors and interfaces; their hardware quirks are not uncommon.

Will the cloud see GPU driver explosion? The cloud VMs for dryrun need to install drivers for all GPU models on clients. Fortunately, maintaining the drivers will not add much burden, as the total number of needed GPU drivers is small. A single GPU driver often supports many GPU models of the same family [136], [249]; these GPUs share much driver code while differing in register definitions, hardware revisions, and erratum. For instance, Mali Bifrost and Qualcomm Adreno 6xx drivers each support 6 and 7 GPUs [250], [251]. As Section 4.6 will show, by crafting the kernel device tree, we can incorporate multiple GPU drivers in one Linux kernel image to be used by the cloud VMs.

4.3.2 The CoDry architecture

Figure 4.4 shows the architecture. The cloud service manages multiple VM images, each installed with a variant 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. All the communication between the cloud VM and the TEE is authenticated and encrypted.

CoDry’s recorder comprises two shims for the cloud (DriverShim) and for 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 interrupts 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.

4.3.3 Challenge: long network delays

A GPU stack is designed under the assumption that CPU and GPU co-locate on an on-chip interconnect with sub-microsecond delays. CoDry breaks the assumption by spanning the interconnect over the Internet with tens of ms or even seconds of delays. As a result, the GPU driver is blocked frequently. The GPU driver frequently issues register accesses; each register access stalls the driver for one round trip time (RTT). Taking MNIST inference as an example, the GPU driver roughly issues 2800 register accesses, taking 117 seconds on cellular network.

Long RTTs also make memory synchronization slow. CoDry needs to synchronize the memory views of the driver (cloud) and the GPU (client). When they run on the same machine, the driver and the GPU exchange extensive information via shared memory: commands, shader code, and input/output data. When the driver and the GPU are distributed,

maintaining such a shared memory illusion may see prohibitive slowdown. As we will show in Section 4.5, classic distributed shared memory (DSM) misses key opportunity in dryrun.

The long recording delay, often hundreds of seconds shown in Section 2.9, render CoDry unusable. (1) An ML workload has to wait long before its first execution in TEE. (2) During a record run, the TEE must exclusively owns the GPU, blocking the normal-world GPU apps for long and hurting the system interactivity. (3) The cloud cost is increased, because CoDry keeps the VMs alive for extended time. (4) The GPU stack often throws exceptions, because the long delays violate many timing assumptions implicitly made by the stack code.

4.4 Hiding Register Access Delays

To overcome the long network delays in CPU/GPU interactions, we retrofit known I/O optimizations to exploit new opportunities.

4.4.1 Register Access Deferral

Problem By design, a GPU driver weaves GPU register accesses into its instruction stream; it executes register accesses and CPU instructions synchronously in program order. For example in Figure 4.5(a), the driver cannot issue the second register access until the first access and the CPU instructions preceding the second register access complete. 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.

Basic idea We coalesce the round trips by making register accesses asynchronous: as shown in Figure 4.5(b), DriverShim defers register accesses as the driver executes, until the driver cannot continue execution without the value from any 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.

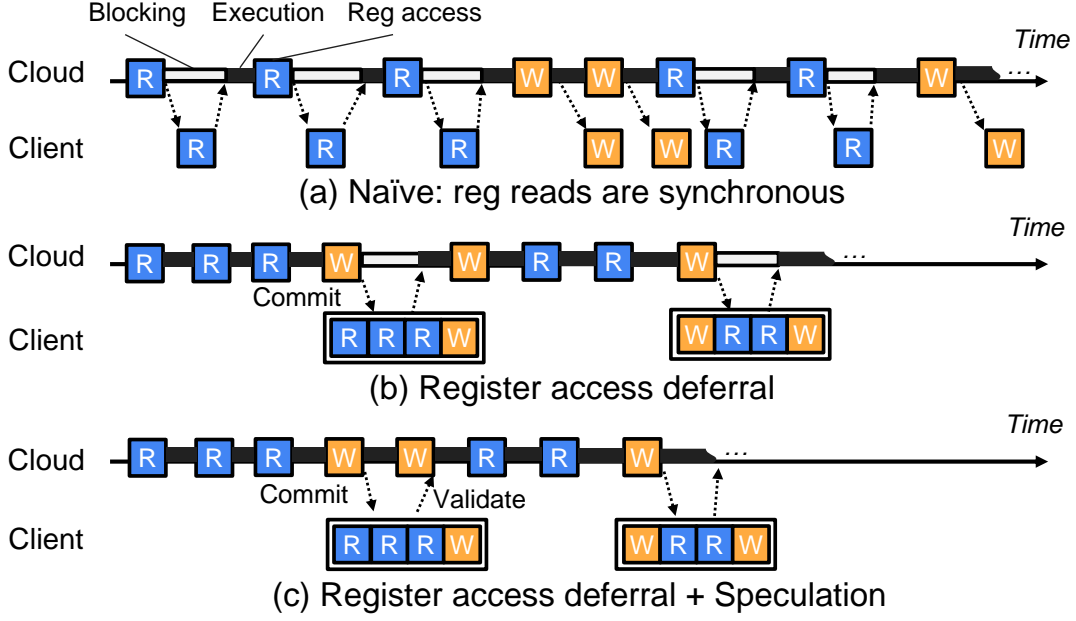


Figure 4.5. CoDry’s strategies for hiding long RTTs

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 4.1 (a) shows examples: variable `qrk_mmu` depends on the read from register `MMU_CONFIG`; the write to `MMU_CONFIG` on line 7 depends on the register read on line 3. 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 can be encoded in a later register write to be queued, e.g. `reg_write(MMU_CONFIG, $\mathcal{S} \mid 0x10$)`, where \mathcal{S} is a symbol. After a commit returns concrete register values, DriverShim

```

1 // detect hardware quirks
2 qrk_shader = reg_read(SHADER_CONFIG);
3 qrk_mmu     = reg_read(MMU_CONFIG);

4 // configure GPU MMU accordingly
5 if (dev->coherency == COHERENCY_ACE)
6     qrk_mmu |= MMU_ALLOW_SNOOP_DISPARITY;

7 reg_write(MMU_CONFIG, qrk_mmu);
8 ...
9 // commit

```

Deferral

$\mathcal{S}_1 = \text{READ}(\text{TILER_CONFIG})$

$\mathcal{S}_2 = \text{READ}(\text{MMU_CONFIG})$

$\text{WRITE}(\text{MMU_CONFIG}, (\mathcal{S}_1 | 0 \times 10))$

Symbolic expressions

qrk_shader	\mathcal{S}_1
qrk_mmu	$\mathcal{S}_1 0 \times 10$

(a) Data dependency

<pre>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 }</pre>	<div>Deferral</div> <div>\mathcal{S}_1=READ(JOB_IRQ_STATUS)</div> <div>WRITE(JOB_IRQ_CLEAR, \mathcal{S}_1)</div> <div>\mathcal{S}_2=READ(TILER_PRESENT)</div> <div>\mathcal{S}_3=READ(SHADER_PRESENT)</div>
---	--

(b) Control dependency (symbolic expressions omitted)

Listing 4.1 Code examples of data and control dependencies. The register accesses are deferred in the queue; the driver keeps running with symbolic values until commit.

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 uncommitted register read, as shown in Listing 4.1 (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 [252]. 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 The 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 another thread attempts to read X before the commit? Will it read the stale value of X ?

DriverShim provides a known memory model of *release consistency* [253] 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 [254], 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 one hot function but not entering another, 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.

4.4.2 Speculation

Basic idea Even with deferred register accesses, each commit is still synchronous taking one RTT (Figure 4.5(b)). DriverShim further makes *some* commits asynchronous to hide their RTTs. The idea is shown in Figure 4.5(c): rather than waiting for a commit \mathcal{C} to complete, DriverShim predicts the values of all register reads enclosed in \mathcal{C} and continues driver execution with the predicated values; later, when \mathcal{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, as will be discussed below. Misprediction incurs performance penalty but does not violate correctness.

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

Why recurring segments? We identify the following common causes. (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 prediction when the history of commits shows high confidence.

When DriverShim is about to make a commit \mathcal{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 \mathcal{C} : if all the k historical commits have returned identical sequences of register read values, DriverShim uses the values for prediction; otherwise, DriverShim avoids speculation for \mathcal{C} , executing it synchronously instead. k is a configurable parameter controlling the DriverShim’s 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, not differentiating 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 from 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 4.1 (b) shows an example: if the read of `JOB_IRQ_STATUS` (line 9) is found to be mispredicted after the second commit (line 10), the driver already contains 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 prevent 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 4.1 (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 follow 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 variable 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.

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 [237] for both parties to restart 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 4.7.3.

4.4.3 Offloading polling loops

A GPU driver often invokes polling loops, e.g. to busy wait for register value changes as shown in Listing 4.2. 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 com-

```

1  u32 cmd = PGT_UPDATE;
2  Int max = MAX_LOOP;
3  u32 val = reg_read(MMU_STATUS);

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

6  if (max == 0) } Predicate to
7      return -1   Predict
8  else reg_write(MMU_CMD, cmd);

```

Listing 4.2 Code example of a polling loop.

mit. (2) Speculation on a polling loop is difficult: by design above, DriverShim must predict the iteration count before 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 of 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 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 external impact, e.g. locking and printk().

Simple polling loops allow optimizations as will be discussed below. DriverShim uses static analysis to find all of them in the GPU driver. Complex polling loops that misfit the definition above are rare; DriverShim just executes them without optimizations.

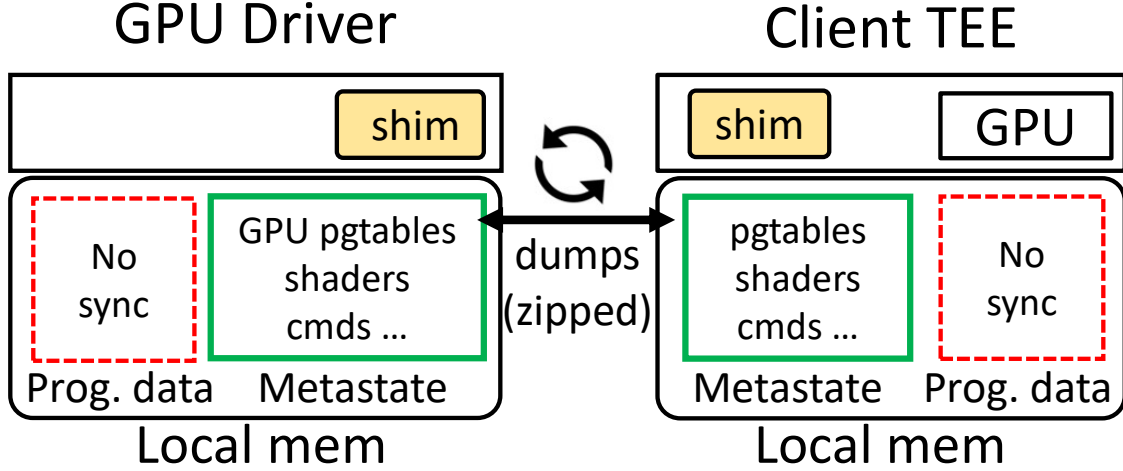


Figure 4.6. Selective memory synchronization of GPU metastate but not program data

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 4.4.1, 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 4.2), DriverShim extracts and predicts the *predicate* on the iteration count, e.g. $(\text{max} \neq 0)$, which is more predictable. When the client returns the actual iteration count, DriverShim evaluates the predicate in order to validate the prediction.

4.5 Memory Synchronization

Problem While the driver (cloud) and the GPU (client) run on their own local memories, we need to synchronize the view of shared memory between them as in Figure 4.6. Memory synchronization has been a central issue in distributed execution[244], [245], [253], [255]. 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 [244] or synchronizing shared memory of a data-race free program at the lock/unlock operations [253].

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 make an educated guess 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 error-prone to build CoDry based on such brittle, 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, because individual GPU compute jobs are sizable [237]. With the constraint, the driver prepares GPU jobs (and accesses the shared memory) only when the GPU is idle; the GPU is executing jobs (and accesses the memory) only when the driver is idle. As a result, we maintain an invariant:

The driver and the client GPU will never access the shared memory simultaneously.

When to synchronize? The cloud and client synchronize when the GPU is about to become busy or idle:

- *Cloud* \Rightarrow *client*. Right before the register write that starts a new GPU job, DriverShim dumps kernel memory that the driver allocates for the 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 state needed for the new job, and has updated the GPU pagetables for mapping the memory state.
- *Client* \Rightarrow *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. Specifically, the GPU cache flush action is either prescribed in the command stream [256] or requested at the beginning of the interrupt handler [257].

To further safeguard the aforementioned invariant, we implement continuous validation. 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 pagetable when the GPU becomes idle; any spurious access from GPU will be trapped and reported.

What to synchronize? As shown in Figure 4.6, 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 job descriptors. Synchronizing program data, including input/output and intermediate GPU buffers, is unnecessary. This is effective as program data constitutes most of GPU memory footprint.

How to locate metastate in the shared memory, given that the memory layout is often 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 [258]. (2) For GPU hardware lacking permission bits, CoDry infers the usage of memory regions from IOCTL() flags used by ML workloads to map these regions. For instance, a region mapped as readonly cannot hold GPU commands, because the GPU runtime needs the write permission to emit GPU commands. (3) If the above knowledge is unavailable, the DriverShim simply replaces an ML workload’s inputs and parameters as zeros. Doing so will result in abundant zeros in the GPU’s program data, making memory dumps highly compressible.

Atop selective memory synchronization, we apply standard compression techniques. Both shims use range encoding to compress memory dumps; each shim calculates and transfers the deltas of memory dumps between consecutive synchronization points.

4.6 Implementations

Platforms We implement the CoDry prototype on the following platforms. The cloud service runs on Odroid C4, an Arm board with 4 Cortex-A55 cores. The client runs on Hikey960 which has 4 Cortex-A73 and A53 cores, and a Mali G71 MP8 GPU. Our choice

of Arm processors for the cloud is for engineering ease rather than a hard requirement; the cloud service can run on x86 machines with binary translation [245].

The cloud service runs Debian 9.4 (Linux v4.14) with a GPU stack composed of a ML framework (ACL v20.05 [148]), a runtime (`libmali.so`), and a driver (Mali Bifrost r24 [136]). Under the cloud 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 [259], 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 (§4.4.1), 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 4.4 and 4.5.

DriverShim communicates with the client via TCP-based messages in our custom formats. To avoid potential timeout due to network communications, we add a fixed delay (e.g. 3 seconds) to all the timeout values in the driver. We prepare and install GPU devicetrees in the cloud VM, so the GPU stack can run transparently even a physical GPU is not present [245]. 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 [260]. The communication is authenticated and encrypted by SSL 3.0 with the TEE, before it forwarded through the normal-world OS.

By design, the client’s trusted firmware dynamically switches the GPU between the normal world and the TEE with a configurable TrustZone address space controller (TZASC) [234]. Yet, our client platform (Hikey960) has a proprietary TZASC which lacks public documen-

tation [75]. 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 may need to access SoC resources not managed by the GPU driver, e.g. power/clock for GPU. While the TEE may invoke related kernel functions in the normal-world OS via RPC [245], we protect these resources inside the TEE as did in prior work for stronger security [234].

4.7 Evaluation

The evaluation answers the following questions.

- Is CoDry secure against attacks? (§ 4.7.1)
- What are the delays of CoDry? (§ 4.7.2)
- Are CoDry’s optimizations significant? (§ 4.7.3)
- What is the energy implication of CoDry? (§4.7.4)

4.7.1 Security Analysis

Threat model We trust the cloud service, assuming its GPU stack is being attested [246], [247]. We trust the client’s TEE 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.

Integrity CoDry’s *recording integrity* is collaboratively 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. CoDry’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 CoDry’s *recording* never leaks program data from TEE, e.g. ML model parameters or inputs, since recording does not require such data. It however may leak some information about the ML workload, as the workload *code* such as GPU shaders moves through the network. Although the network traffic is encrypted, it may nevertheless leak workload information, e.g. NN types, via side channels. Such side channels can be mitigated by orthogonal solutions [40], [156].

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 [78], [261].

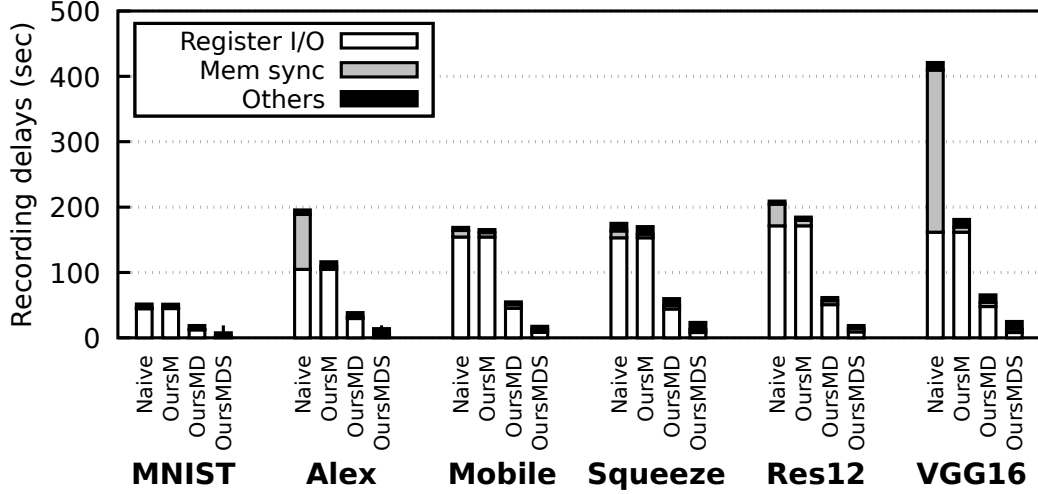
Availability Like any cloud-based service, *recording* availability of CoDry depends on network conditions and the cloud availability, which are vulnerable to DDoS attacks. Its *replay* availability is at the same level of the TrustZone TEE, given the GPU power is managed by the TEE not the OS [234].

4.7.2 Performance

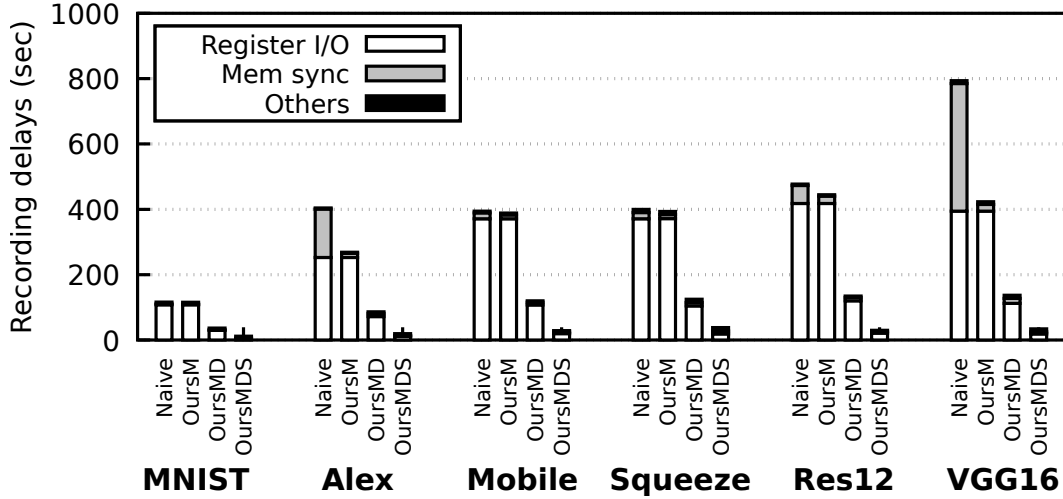
Methodology As shown in Table 4.1, we test CoDry on inference with 6 popular NNs running atop ARM Compute Library [148]. We measure CoDry’s recording delay under two network conditions as controlled by NetEm [262]: i) WiFi-like (20 ms RTT, 80 Mbps) and ii) cellular-like (50 ms RTT, 40 Mbps) [263]. The hardware platform is described in Section 4.6.

We study the following versions:

- **Naive** incurs a round trip per register access and synchronizes entire GPU memory before/after a GPU job.
- **OursM** includes selective memory synchronization (§4.5).
- **OursMD**, in addition to **OursM**, includes register access deferral (§4.4.1); it generates per-commit round trips.
- **OursMDS** additionally includes speculation (§4.4.2). It represents CoDry with all our techniques,



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



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

Figure 4.7. Recording delays of CoDry(OursMDS) are significantly lower than other versions

Recording delays Figure 4.7 shows the end-to-end recording delays. Naive incurs long recording delays even on WiFi ranging from 52 seconds (MNIST, a small NN) to 423 seconds (VGG16, a large NN). Such delays become much higher on the cellular network, range from 116 seconds to 795 seconds. As discussed in Section 4.3.3, such high delays not only slow down ML workload launch but also hurts interactivity because the TEE must lock the GPU during recording. Compared to Naive, OursMDS reduces the delays by up to 95% to 18

Table 4.1. Statistics of record runs, showing CoDry significantly reduces network round trips that block the recording and the memory synchronization traffic

NNs (# GPU jobs)	# Blocking RoundTrip			MemSync (MB)	
	OursM	OursMD	OursMDS	Naive	OursM
MNIST (23)	2837	585	65	3.07	0.75
AlexNet (60)	5008	1392	196	454.91	4.22
MobileNet (104)	7307	2097	320	37.39	11.79
SqueezeNet (98)	7373	2049	303	41.26	11.3
ResNet12 (111)	8326	2352	345	151.16	12.96
VGG16 (96)	7662	2184	309	1215.23	10.21

Table 4.2. Replay delays of CoDry (OursMDS) are similar to Native, which executes benchmarks on the GPU stack in the normal world of the same device

	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

seconds (WiFi) and 30 seconds (cellular) on average. We deem these delays as acceptable, as they are comparable to mobile app installation delays reported to be 10 – 50 seconds [264].

Replay delays CoDry’s replay incurs minor overhead in workload execution as shown in Table 4.2. Compares native executions, CoDry’s replay delays range from 68% lower to 3% higher (25% lower on average). CoDry performance advantage comes from its removal of the complex GPU stack. We notice that these results are consistent with the prior work [237].

4.7.3 Validation of key designs

Efficacy of deferral As shown in Figure 4.7 (OursM vs. OursMD), register access deferral reduces the overall delays by 65% (WiFi) and 69% (cellular). Table 4.1 further shows that the deferral reduces the number of round trips by 73% on average. With deferral, each commit encapsulates 3.8 register accesses on average.

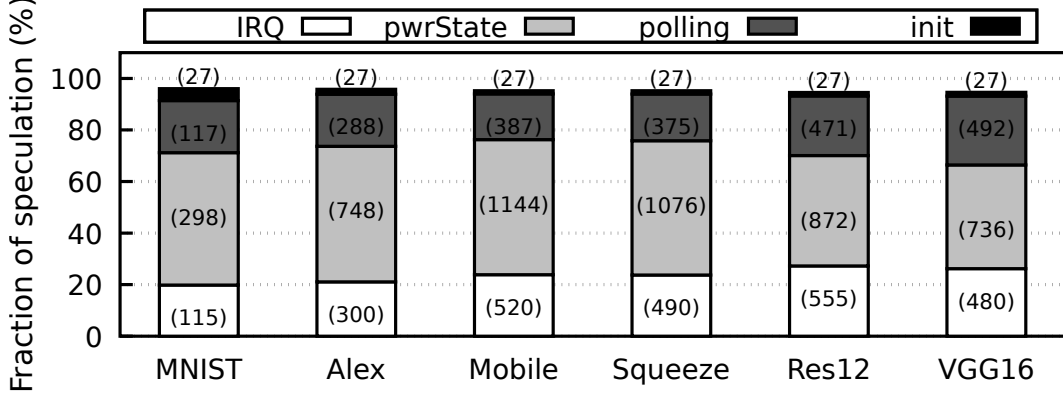


Figure 4.8. Breakdown of speculation; the actual number of commits for each segment are shown in parentheses.

Efficacy of speculation We run all six benchmarks with retaining register access history in between, allowing CoDry to reuse history across benchmarks. Figure 4.7 (OursMDS vs. OursMD) shows that speculation reduces the recording delays by 60% to 74%. Table 4.1 further shows OursMDS achieves 86 % reduced number of round trips on average. Such benefit mainly come from coalescing round trips of asynchronous commits.

We further investigate the speculation success rates and find 95% of commits (99% register accesses) satisfy the speculation criteria (§4.4.2). These commits are generated by GPU driver routines that roughly fall into four categories. (1) *Init*: probe hardware configuration when the driver is loaded. (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 4.8 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 driver reads and writes a register LATEST_FLUSH_ID which reflects the GPU cache state and can be nondeterministic.

Misprediction cost For the above reasons, we have not observed misprediction in our 1,000 runs of each workload. To validate that CoDry can handle misprediction, we artificially inject into record runs wrong register values. In all the cases of injection, CoDry always detects mismatches between the speculative and the injected register value, initiating rollback

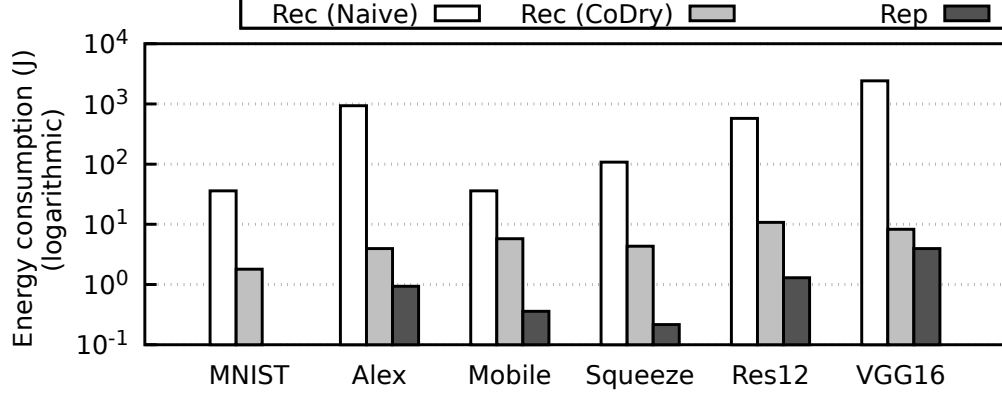


Figure 4.9. Energy consumption for record and replay

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 is 1 and 3 seconds for MNIST and VGG16, respectively. The delays are primarily dominated by driver reload and GPU job recompilation, which overshadow the replay delays on the client GPU hardware.

Selective memory synchronization Figure 4.7 (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 4.1 shows the network traffic for memory synchronization is reduced by 72 – 99%.

Polling offloading (§4.4.3) The numbers of polling instances range from 117 (MNIST) to 492 (VGG16), that generate from 130 to 550 round trips. Offloading polling reduces the total round trips by 13 – 58, making the cost of polling instance one RTT; 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 RTTs often become hidden.

4.7.4 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 does not run any other foreground

applications. Each workload runs 500 iterations and we report the average energy. Figure 4.9 shows the results.

Record. The energy consumed by recording is moderate, ranging from 1.8 – 8.2 J, which is comparable to one by installing a mobile app, e.g. 16 J for Snapchat (80MB) on the same device. Note that it is one-time consumption per workload. Compared to **Naive**, CoDry reduces the system energy consumption 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 delays in Table 4.2. The replaying energy is comparable with the native execution on the original GPU stack of the client device (not shown in the figure).

4.8 Related Work

Remote I/O is adopted for cross-device I/O sharing [255], [265] and task offloading [156], [243]. Unlike CoDry, however, their remoting boundary is at higher-level – device file [255], Android binder IPC [265], and runtime API [156]. Such clean-cut boundaries ease a coarse-grained I/O remoting, e.g. function-level RPC calls. To apply to TEE, however, the client TEE must keep a part of (e.g. device driver) or the entire I/O stack while bloating TCB.

Similar to CoDry, prior works have explored the lowest software level; For efficient dynamic analysis, they forward I/O from VM to mobile system [245] or low-level memory access from emulator to real device [266], [267]. However, their cross-device interfaces are wired, faster than what CoDry addresses, (i.e. wireless connection). CoDry has a different goal: hosting a dryrun service for GPU recording, mitigating communication cost.

Device isolation with TEE Recent works propose TEE-based solutions for GPU isolation by hiding GPU stack in the TEE [6], [161] or security-critical GPU interfaces in the GPU hardware [223]. They, however, require hardware modification and/or bloat TCB inside TEE. Favorable to the insight given by GPURip [237], CoDry offers a remote recording service for clients to reproduce GPU compute without the stack in TEE.

Leveraging TrustZone components, prior works build a trusted path locally, e.g. for secure device control [234] or remotely [8], e.g. to securely display confidential text [235] and

image [268]. Their techniques are well-suited to CoDry for i) discarding adversarial access to the GPU while recording and replaying; ii) building secure channel between cloud VM and client TEE;

Speculative execution is widely explored by prior works; based on caching and prefetching, they facilitate asynchronous file I/O [269]–[271] or speed up VM replication [272] and distributed systems [273]. Unlike such works, CoDry does not prefetch I/O access ahead of time; instead, CoDry hide I/O latency by speculatively continue driver’s workflow deferring read values while replacing them as symbolic expression; it then commit when facing value/control dependencies.

Mobile cloud offloading There has been previous works on cloud offloading [244], [253], [274], which partitions mobile application into two parts: one for local device and the other for the cloud. Facilitating application-layer virtual machine or runtime, each part of the application cooperatively runs from both sides continuously. Unlike them, CoDry does not require runtime or vm support from the client; the offloading is also temporal for dryrun of GPU compute to capture the interactions.

GPU record and replay has been explored to dig out GPU command stream semantics [219]–[221], enhance performance [157], migrate runtime calls [222], and reproduce computation [237]. While they care *what* to record, CoDry’s focus is *how* to record; CoDry addresses costly interaction overhead for remote GPU recording.

Secure client ML Much works has been proposed to protect model and user privacy [5], [275] and/or to secure ML confidentiality [4], [231]. However, they all lack GPU-acceleration which is crucial for resource-hungry client devices. While recent work [232] suggests a verifiable GPU compute with TEE, the complexity of homomorphic encryption significantly burdens client devices.

4.9 Conclusions

CoDry provides a cloud service for GPU recording in a secure way; it performs GPU dryrun interacting with the client GPUs over long wireless communication. Retrofitting

known I/O optimization techniques, CoDry significantly reduces the time and energy consumed by client to get a GPU recording.

5. CONCLUSIONS

On-device computation brings in multiple benefits to end users, such as saving computation and communication cost over a remote server. While deployed in many critical areas, its key applications are designed without strong security guarantees. In concert with growing demands towards on-device computation, its security concerns will be more significant.

Aiming at trustworthy on-device computation, this dissertation contributes the following new system design:

- **StreamBox-TZ: Secure Stream Analytics at the Edge with TrustZone** With the goal of secure stream analytics engine at the edge, we present StreamBox-TZ in Chapter 2. StreamBox-TZ guarantees data confidentiality and integrity via isolated data plane designed and optimized for a TEE based on ARM TrustZone; the constant remote attestation guarantees both analytics correctness and result freshness while incurring low overhead. StreamBox-TZ keeps a low complexity in TEE while only adding 42.5 KB executable to the TCB. The security overhead incurred by StreamBox-TZ is less than 25%.
- **GPURip: A 50-KB GPU Stack for Client ML** In chapter 3, we propose a novel way for secure GPU computation on mobile and embedded devices. Leveraging record and replay technique, GPURip reproduces GPU computation without support of original stack. Thereby, it significantly reduces attack surfaces stem from large and vulnerable GPU stack when run time. We address key challenges towards making GPURip 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). Our evaluation demonstrate GPURip works with a variety of integrated GPU hardware, GPU APIs, ML frameworks, and 33 neural network (NN) implementations for inference or training.

- **CoDry: GPU Acceleration in TrustZone via Safe and Practical Recording**

Finally, Chapter 4 introduces CoDry, a holistic design for GPU-accelerated computation in TrustZone TEE. CoDry addresses GPURip’s key missing piece – the recording environment, which needs both strong security and access to diverse mobile GPUs. In CoDry, a mobile device (which processes the GPU hardware) and a trustworthy cloud service (which runs the GPU stack) exercise the GPU hardware/software in a collaborative, distributed fashion. To overcome numerous network round trips and long delays, CoDry contributes optimizations specific to mobile GPUs: register access deferral, speculation, and metastate-only synchronization. Our evaluation demonstrates CoDry is practical with fast service time (only tens of seconds).

In conclusion, this dissertation presents new design choices for key applications (stream analytics and machine learning) in mobile/edge environments. By minimizing software stack combined with hardware-supported TEE and/or involving trustworthy cloud in the computation, we achieves the goal of trustworthy on-device computation.

REFERENCES

- [1] S. Nunna, A. Kousaridas, M. Ibrahim, M. Dillinger, C. Thuemmler, H. Feussner, and A. Schneider, “Enabling real-time context-aware collaboration through 5g and mobile edge computing,” in *2015 12th International Conference on Information Technology - New Generations*, 2015, pp. 601–605. DOI: [10.1109/ITNG.2015.155](https://doi.org/10.1109/ITNG.2015.155).
- [2] C. She, Y. Duan, G. Zhao, T. Q. S. Quek, Y. Li, and B. Vucetic, “Cross-layer design for mission-critical iot in mobile edge computing systems,” *IEEE Internet of Things Journal*, vol. 6, no. 6, pp. 9360–9374, 2019. DOI: [10.1109/JIOT.2019.2930983](https://doi.org/10.1109/JIOT.2019.2930983).
- [3] Y. Hajizadeh, “Machine learning in oil and gas; a swot analysis approach,” *Journal of Petroleum Science and Engineering*, vol. 176, pp. 661–663, 2019, ISSN: 0920-4105. DOI: <https://doi.org/10.1016/j.petrol.2019.01.113>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0920410519301275>.
- [4] 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*, ser. MobiCom ’19, Los Cabos, Mexico: Association for Computing Machinery, 2019, ISBN: 9781450361699. DOI: [10.1145/3300061.3345447](https://doi.org/10.1145/3300061.3345447). [Online]. Available: <https://doi-org.ezproxy.lib.purdue.edu/10.1145/3300061.3345447>.
- [5] 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*, ser. MobiSys ’20, Toronto, Ontario, Canada: Association for Computing Machinery, 2020, pp. 161–174, ISBN: 9781450379540. DOI: [10.1145/3386901.3388946](https://doi.org/10.1145/3386901.3388946). [Online]. Available: <https://doi-org.ezproxy.lib.purdue.edu/10.1145/3386901.3388946>.
- [6] 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*, ser. IoTDI ’21, Charlottesville, VA, USA: Association for Computing Machinery, 2021, pp. 67–79, ISBN: 9781450383547. DOI: [10.1145/3450268.3453524](https://doi.org/10.1145/3450268.3453524). [Online]. Available: <https://doi.org/10.1145/3450268.3453524>.
- [7] Z. Yao, S. Mirzamohammadi, A. Amiri Sani, and M. Payer, “Milkomeda: Safeguarding the mobile gpu interface using webgl security checks,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 1455–1469, ISBN: 9781450356930. DOI: [10.1145/3243734.3243772](https://doi.org/10.1145/3243734.3243772). [Online]. Available: <https://doi.org/10.1145/3243734.3243772>.

- [8] 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*, ser. APSys ’14, Beijing, China: Association for Computing Machinery, 2014, ISBN: 9781450330244. DOI: [10.1145/2637166.2637225](https://doi.org/10.1145/2637166.2637225). [Online]. Available: <https://doi.org/10.1145/2637166.2637225>.
- [9] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, “Trustshadow: Secure execution of unmodified applications with arm trustzone,” in *Proceedings of the 15th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, ser. MobiSys ’17, Niagara Falls, New York, USA: ACM, 2017, pp. 488–501, ISBN: 978-1-4503-4928-4. DOI: [10.1145/3081333.3081349](https://doi.acm.org/10.1145/3081333.3081349). [Online]. Available: <http://doi.acm.org/10.1145/3081333.3081349>.
- [10] N. Tomas, J. Li, and H. Huang, “An empirical study on culture, automation, measurement, and sharing of devsecops,” in *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, 2019, pp. 1–8. DOI: [10.1109/CyberSecPODS.2019.8884935](https://doi.org/10.1109/CyberSecPODS.2019.8884935).
- [11] Cybersecurity and Infrastructure Security Agency, NIST, *Defending Against Software Supply Chain Attacks*, <https://www.cisa.gov/publication/software-supply-chain-attacks/>.
- [12] *CVE-2019-20577: Smmu page fault in mali gpu driver*, <https://nvd.nist.gov/vuln/detail/CVE-2019-20577>, 2019.
- [13] *CVE-2014-1376: Improper restriction to unspecified opencl api calls*, <https://nvd.nist.gov/vuln/detail/CVE-2014-1376>, 2014.
- [14] Intel, “*iot solutions for upstream oil and gas*”, <https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/oil-and-gas-iot-brief.pdf>, 2017.
- [15] hortonworks, “*iot and predictive big data analytics for oil and gas*”, <https://hortonworks.com/solutions/oil-gas/>, 2017.
- [16] Documentation, “*a new reality for oil & gas*”, https://www.cisco.com/c/dam/en_us/solutions/industries/energy/docs/OilGasDigitalTransformationWhitePaper.pdf, 2017.
- [17] Magazine, “*smart grids: Everything you need to know*”, <https://www.cleverism.com/smart-grids-everything-need-know/>, 2014.
- [18] M. P. Andersen, S. Kumar, C. Brooks, A. von Meier, and D. E. Culler, “Distil: Design and implementation of a scalable synchrophasor data processing system,” in *2015 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, Nov. 2015, pp. 271–277. DOI: [10.1109/SmartGridComm.2015.7436312](https://doi.org/10.1109/SmartGridComm.2015.7436312).

- [19] M. P. Andersen and D. E. Culler, “Btrdb: Optimizing storage system design for time-series processing,” in *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, ser. FAST’16, Santa Clara, CA: USENIX Association, 2016, pp. 39–52, ISBN: 978-1-931971-28-7. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2930583.2930587>.
- [20] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: Vision and challenges,” *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, Oct. 2016, ISSN: 2327-4662. DOI: [10.1109/JIOT.2016.2579198](https://doi.org/10.1109/JIOT.2016.2579198).
- [21] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos, “Edge analytics in the internet of things,” *IEEE Pervasive Computing*, vol. 14, no. 2, pp. 24–31, Apr. 2015, ISSN: 1536-1268. DOI: [10.1109/MPRV.2015.32](https://doi.org/10.1109/MPRV.2015.32).
- [22] E. I. W. Group. (2018). Iot developer survey 2018, [Online]. Available: <https://blogs.eclipse.org/post/benjamin-cab%5C%3%5C%A9/key-trends-iot-developer-survey-2018%7D>.
- [23] M. G. Institute, *The internet of things: Mapping the value beyond the hype*, <http://www.dell.com/learn/us/en/uscorp1/press-releases/2016-04-14-dell-further-democratizes-advanced-analytics>, 2016.
- [24] X. Wu, R. Dunne, Q. Zhang, and W. Shi, “Edge computing enabled smart firefighting: Opportunities and challenges,” in *Proceedings of the Fifth ACM/IEEE Workshop on Hot Topics in Web Systems and Technologies*, ser. HotWeb ’17, San Jose, California: ACM, 2017, 11:1–11:6, ISBN: 978-1-4503-5527-8. DOI: [10.1145/3132465.3132475](https://doi.org/10.1145/3132465.3132475). [Online]. Available: <http://doi.acm.org/10.1145/3132465.3132475>.
- [25] Symantec, *Internet Security Threat Report*, 2017.
- [26] Wind River, *SECURITY IN THE INTERNET OF THINGS – Lessons from the Past for the Connected Future*, 2017.
- [27] D. Vasisht, Z. Kapetanovic, J. Won, X. Jin, R. Chandra, S. Sinha, A. Kapoor, M. Sudarshan, and S. Stratman, “Farmbeats: An iot platform for data-driven agriculture,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA: USENIX Association, 2017, pp. 515–529, ISBN: 978-1-931971-37-9. [Online]. Available: <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vasisht>.
- [28] Microsoft, *Microsoft azure iot edge– extending cloud intelligence to edge devices*, <https://azure.microsoft.com/en-us/services/iot-edge/>, 2017.

- [29] CISCO, *White paper: The cisco edge analytics fabric system*, <http://www.cisco.com/c/dam/en/us/products/collateral/analytics-automation-software/edge-analytics-fabric/eaf-whitepaper.pdf>, 2016.
- [30] Dell, *Dell further democratizes advanced analytics with latest release of statistica*, <http://www.dell.com/learn/us/en/uscorp1/press-releases/2016-04-14-dell-further-democratizes-advanced-analytics>, 2016.
- [31] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, “An empirical study on configuration errors in commercial and open source systems,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP ’11, Cascais, Portugal: ACM, 2011, pp. 159–172, ISBN: 978-1-4503-0977-6. DOI: [10.1145/2043556.2043572](https://doi.org/10.1145/2043556.2043572). [Online]. Available: <http://doi.acm.org/10.1145/2043556.2043572>.
- [32] Art Manion, *CERT/CC Blog – Anatomy of Java Exploits*, <https://insights.sei.cmu.edu/cert/2013/01/anatomy-of-java-exploits.html/>, 2013.
- [33] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, “Linux kernel vulnerabilities: State-of-the-art defenses and open problems,” in *Proceedings of the Second Asia-Pacific Workshop on Systems*, ser. APSys ’11, Shanghai, China: ACM, 2011, 5:1–5:5, ISBN: 978-1-4503-1179-3. DOI: [10.1145/2103799.2103805](https://doi.org/10.1145/2103799.2103805). [Online]. Available: <http://doi.acm.org/10.1145/2103799.2103805>.
- [34] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, “Bug characteristics in open source software,” *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.
- [35] J. H. Saltzer and M. D. Schroeder, “The protection of information in computer systems,” *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.
- [36] S. Arnavotov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’Keeffe, M. L. Stillwell, D. Goltzsche, D. Eysers, R. Kapitza, P. Pietzuch, and C. Fetzer, “Scone: Secure linux containers with intel sgx,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’16, Savannah, GA, USA: USENIX Association, 2016, pp. 689–703, ISBN: 978-1-931971-33-1. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3026877.3026930>.
- [37] C.-C. Tsai, D. E. Porter, and M. Vij, “Graphene-sgx: A practical library os for unmodified applications on sgx,” in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC ’17, Santa Clara, CA, USA: USENIX Association, 2017, pp. 645–658, ISBN: 978-1-931971-38-6. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3154690.3154752>.

- [38] A. Baumann, M. Peinado, and G. Hunt, “Shielding applications from an untrusted cloud with haven,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, CO: USENIX Association, 2014, pp. 267–283, ISBN: 978-1-931971-16-4. [Online]. Available: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann>.
- [39] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, “Vc3: Trustworthy data analytics in the cloud using sgx,” in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP ’15, Washington, DC, USA: IEEE Computer Society, 2015, pp. 38–54, ISBN: 978-1-4673-6949-7. DOI: [10.1109/SP.2015.10](https://doi.org/10.1109/SP.2015.10). [Online]. Available: <http://dx.doi.org/10.1109/SP.2015.10>.
- [40] 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)*, Boston, MA: USENIX Association, 2017, pp. 283–298, ISBN: 978-1-931971-37-9. [Online]. Available: <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng>.
- [41] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, “Securestreams: A reactive middleware framework for secure data stream processing,” in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS ’17, Barcelona, Spain: ACM, 2017, pp. 124–133, ISBN: 978-1-4503-5065-5. DOI: [10.1145/3093742.3093927](https://doi.org/10.1145/3093742.3093927). [Online]. Available: <http://doi.acm.org/10.1145/3093742.3093927>.
- [42] S. Kim, J. Han, J. Ha, T. Kim, and D. Han, “Enhancing security and privacy of tor’s ecosystem by using trusted execution environments,” in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA: USENIX Association, 2017, pp. 145–161, ISBN: 978-1-931971-37-9. [Online]. Available: <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kim-seongmin>.
- [43] B. Glavic, K. Sheykh Esmaili, P. M. Fischer, and N. Tatbul, “Ariadne: Managing fine-grained provenance on data streams,” in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, ser. DEBS ’13, Arlington, Texas, USA: ACM, 2013, pp. 39–50, ISBN: 978-1-4503-1758-0. DOI: [10.1145/2488222.2488256](https://doi.org/10.1145/2488222.2488256). [Online]. Available: <http://doi.acm.org/10.1145/2488222.2488256>.
- [44] Y. L. Simmhan, B. Plale, and D. Gannon, “A survey of data provenance in e-science,” *SIGMOD Rec.*, vol. 34, no. 3, pp. 31–36, Sep. 2005, ISSN: 0163-5808. DOI: [10.1145/1084805.1084812](https://doi.org/10.1145/1084805.1084812). [Online]. Available: <http://doi.acm.org/10.1145/1084805.1084812>.
- [45] *ARM TrustZone*, <http://www.arm.com/products/processors/technologies/trustzone/index.php>.

- [46] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ACM, 2013, pp. 423–438.
- [47] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, “Streambox: Modern stream processing on a multicore machine,” in *Proceedings of the 2017 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC’17, Philadelphia, PA: USENIX Association, 2017, pp. 231–242, ISBN: 978-1-931971-10-2.
- [48] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing, “Trill: A high-performance incremental query processor for diverse analytics,” *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 401–412, 2014.
- [49] *Apache Beam*, <https://beam.apache.org/>.
- [50] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, “Innovative instructions and software model for isolated execution,” in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP ’13, Tel-Aviv, Israel: ACM, 2013, 10:1–10:1, ISBN: 978-1-4503-2118-1. DOI: [10.1145/2487726.2488368](https://doi.org/10.1145/2487726.2488368). [Online]. Available: <http://doi.acm.org/10.1145/2487726.2488368>.
- [51] Preferred networks, *Sensorbee: Lightweight stream processing engine for iot*, <http://sensorbee.io/>, 2017.
- [52] Z. Jerzak and H. Ziekow, “The debs 2014 grand challenge,” in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS ’14, Mumbai, India: ACM, 2014, pp. 266–269, ISBN: 978-1-4503-2737-4. DOI: [10.1145/2611286.2611333](https://doi.org/10.1145/2611286.2611333). [Online]. Available: <http://doi.acm.org/10.1145/2611286.2611333>.
- [53] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, “Naiad: A timely dataflow system,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP ’13, Farmington, Pennsylvania: ACM, 2013, pp. 439–455, ISBN: 978-1-4503-2388-8. DOI: [10.1145/2517349.2522738](https://doi.org/10.1145/2517349.2522738). [Online]. Available: <http://doi.acm.org/10.1145/2517349.2522738>.
- [54] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, “The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing,” *Proceedings of the VLDB Endow.*, vol. 8, no. 12, pp. 1792–1803, Aug. 2015, ISSN: 2150-8097. DOI: [10.14778/2824032.2824076](https://doi.org/10.14778/2824032.2824076). [Online]. Available: <http://dx.doi.org/10.14778/2824032.2824076>.

- [55] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou, “Streamscope: Continuous reliable distributed processing of big data streams,” in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI’16, Santa Clara, CA: USENIX Association, 2016, pp. 439–453, ISBN: 978-1-931971-29-4. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2930611.2930640>.
- [56] A. Arasu, S. Babu, and J. Widom, “The cql continuous query language: Semantic foundations and query execution,” *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, Jun. 2006, ISSN: 1066-8888. DOI: [10.1007/s00778-004-0147-z](https://doi.org/10.1007/s00778-004-0147-z). [Online]. Available: <http://dx.doi.org/10.1007/s00778-004-0147-z>.
- [57] EsperTech, *Esper*, <http://www.espertech.com/esper/>, 2017.
- [58] J. W. Voun, R. Jhala, and S. Lerner, “Relay: Static race detection on millions of lines of code,” in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE ’07, Dubrovnik, Croatia: ACM, 2007, pp. 205–214, ISBN: 978-1-59593-811-4. DOI: [10.1145/1287624.1287654](https://doi.org/10.1145/1287624.1287654). [Online]. Available: <http://doi.acm.org/10.1145/1287624.1287654>.
- [59] R. Clapis, *GO GET MY/VULNERABILITIES: AN IN-DEPTH ANALYSIS OF GO LANGUAGE RUNTIME AND THE NEW CLASS OF VULNERABILITIES IT INTRODUCES*, Blackhat Asia 2017, 2017.
- [60] *CVE-2010-3190: Untrusted search path vulnerability in the microsoft foundation class (mfc) library*, <https://nvd.nist.gov/vuln/detail/CVE-2010-3190>, 2010.
- [61] *CVE-2008-0171: Boost.regex allows context-dependent attackers to cause failed assertion and crash*, <https://nvd.nist.gov/vuln/detail/CVE-2008-0171s>, 2008.
- [62] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, “Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems,” *SIGPLAN Not.*, vol. 43, no. 3, pp. 2–13, Mar. 2008, ISSN: 0362-1340. DOI: [10.1145/1353536.1346284](https://doi.org/10.1145/1353536.1346284). [Online]. Available: <http://doi.acm.org/10.1145/1353536.1346284>.
- [63] J. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, “Trustvisor: Efficient tcb reduction and attestation,” in *Security and Privacy (SP), 2010 IEEE Symposium on*, May 2010, pp. 143–158. DOI: [10.1109/SP.2010.17](https://doi.org/10.1109/SP.2010.17).
- [64] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, “Flicker: An execution infrastructure for tcb minimization,” *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 4, pp. 315–328, Apr. 2008, ISSN: 0163-5980. DOI: [10.1145/1357010.1352625](https://doi.org/10.1145/1357010.1352625). [Online]. Available: <http://doi.acm.org/10.1145/1357010.1352625>.

- [65] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, “Inktag: Secure applications on an untrusted operating system,” in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’13, Houston, Texas, USA: ACM, 2013, pp. 265–278, ISBN: 978-1-4503-1870-9. DOI: [10.1145/2451116.2451146](https://doi.org/10.1145/2451116.2451146). [Online]. Available: <http://doi.acm.org/10.1145/2451116.2451146>.
- [66] *CVE-2018-8822: Incorrect buffer length handling in the ncp_read_kernel function could be exploited by malicious ncpfs servers to crash the kernel or execute code*, <https://nvd.nist.gov/vuln/detail/CVE-2018-8822>, 2017.
- [67] *CVE-2017-11176: The mq_notify function in the linux kernel allows attackers to cause a denial of service or possibly have unspecified other impact*, <https://nvd.nist.gov/vuln/detail/CVE-2017-11176>, 2017.
- [68] *CVE-2009-2493: Active template library does not properly restrict use of oleloadfrom-stream in instantiating objects from data streams, which allows remote attackers to execute arbitrary code*, <https://nvd.nist.gov/vuln/detail/CVE-2009-2493>, 2009.
- [69] *CVE-2017-12629: Remote code execution occurs in apache solr*, <https://nvd.nist.gov/vuln/detail/CVE-2017-12629>, 2017.
- [70] *CVE-2016-10229: Udp.c in the linux kernel before 4.5 allows remote attackers to execute arbitrary code*, <https://nvd.nist.gov/vuln/detail/CVE-2016-10229>, 2016.
- [71] S. Saroiu and A. Wolman, “I am a sensor, and i approve this message,” in *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, ser. HotMobile ’10, Annapolis, Maryland: ACM, 2010, pp. 37–42, ISBN: 978-1-4503-0005-6. DOI: [10.1145/1734583.1734593](https://doi.org/10.1145/1734583.1734593). [Online]. Available: <http://doi.acm.org/10.1145/1734583.1734593>.
- [72] H. Liu, S. Saroiu, A. Wolman, and H. Raj, “Software abstractions for trusted sensors,” in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’12, Low Wood Bay, Lake District, UK: ACM, 2012, pp. 365–378, ISBN: 978-1-4503-1301-8. DOI: [10.1145/2307636.2307670](https://doi.org/10.1145/2307636.2307670). [Online]. Available: <http://doi.acm.org/10.1145/2307636.2307670>.
- [73] P. Gilbert, J. Jung, K. Lee, H. Qin, D. Sharkey, A. Sheth, and L. P. Cox, “Youprove: Authenticity and fidelity in mobile sensing,” in *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys ’11, Seattle, Washington: ACM, 2011, pp. 176–189, ISBN: 978-1-4503-0718-5. DOI: [10.1145/2070942.2070961](https://doi.org/10.1145/2070942.2070961). [Online]. Available: <http://doi.acm.org/10.1145/2070942.2070961>.

- [74] T. Trippel, O. Weisse, W. Xu, P. Honeyman, and K. Fu, “WALNUT: Waging doubt on the integrity of MEMS accelerometers with acoustic injection attacks,” in *Proceedings of the 2nd Annual IEEE European Symposium on Security and Privacy*, IEEE European Symposium on Security and Privacy (Oaklawn), Paris, France, Apr. 2017.
- [75] 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)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 541–556, ISBN: 978-1-931971-40-9. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hua>.
- [76] *CVE-2015-4421: In huawei mate7*, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4421>, 2015.
- [77] *CVE-2015-4422: In huawei mate7*, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4422>, 2015.
- [78] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, “Armageddon: Cache attacks on mobile devices,” in *USENIX Security Symposium*, 2016, pp. 549–564.
- [79] Y. Xu, W. Cui, and M. Peinado, “Controlled-channel attacks: Deterministic side channels for untrusted operating systems,” in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP ’15, Washington, DC, USA: IEEE Computer Society, 2015, pp. 640–656, ISBN: 978-1-4673-6949-7. DOI: [10.1109/SP.2015.45](https://doi.org/10.1109/SP.2015.45). [Online]. Available: <https://doi.org/10.1109/SP.2015.45>.
- [80] A. Becher, Z. Benenson, and M. Dornseif, “Tampering with motes: Real-world physical attacks on wireless sensor networks,” in *International Conference on Security in Pervasive Computing*, Springer, 2006, pp. 104–118.
- [81] R. Anderson and M. Kuhn, “Low cost attacks on tamper resistant devices,” in *International Workshop on Security Protocols*, Springer, 1997, pp. 125–136.
- [82] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “Sel4: Formal verification of an os kernel,” in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP ’09, Big Sky, Montana, USA: ACM, 2009, pp. 207–220, ISBN: 978-1-60558-752-3. DOI: [10.1145/1629575.1629596](https://doi.org/10.1145/1629575.1629596). [Online]. Available: <http://doi.acm.org/10.1145/1629575.1629596>.

- [83] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman, “Protecting data on smartphones and tablets from memory attacks,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’15, Istanbul, Turkey: ACM, 2015, pp. 177–189, ISBN: 978-1-4503-2835-7. DOI: [10.1145/2694344.2694380](https://doi.org/10.1145/2694344.2694380). [Online]. Available: <http://doi.acm.org/10.1145/2694344.2694380>.
- [84] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, “Case: Cache-assisted secure execution on arm processors,” in *Security and Privacy (SP), 2016 IEEE Symposium on*, IEEE, 2016, pp. 72–90.
- [85] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu, “Facade: A compiler and runtime for (almost) object-bounded big data applications,” in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ser. ASPLOS ’15, Istanbul, Turkey: ACM, 2015, pp. 675–690, ISBN: 978-1-4503-2835-7. DOI: [10.1145/2694344.2694345](https://doi.org/10.1145/2694344.2694345). [Online]. Available: <http://doi.acm.org/10.1145/2694344.2694345>.
- [86] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, “Securekeeper: Confidential zookeeper using intel sgx,” in *Proceedings of the 17th International Middleware Conference*, ser. Middleware ’16, Trento, Italy: ACM, 2016, 14:1–14:13, ISBN: 978-1-4503-4300-8. DOI: [10.1145/2988336.2988350](https://doi.org/10.1145/2988336.2988350). [Online]. Available: <http://doi.acm.org/10.1145/2988336.2988350>.
- [87] J. Lind, C. Priebe, D. Muthukumaran, D. O’Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eysers, R. Kapitza, C. Fetzer, and P. Pietzuch, “Glamdring: Automatic application partitioning for intel sgx,” in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (ATC)*, ser. USENIX ATC ’17, Santa Clara, CA, USA: USENIX Association, 2017, pp. 285–298, ISBN: 978-1-931971-38-6. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3154690.3154718>.
- [88] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, “Panoply: Low-tcb linux applications with SGX enclaves,” in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, The Internet Society, 2017. [Online]. Available: <https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/panoply-low-tcb-linux-applications-sgx-enclaves/>.
- [89] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury, “Automated partitioning of android applications for trusted execution environments,” in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, ser. ICSE ’16, Austin, Texas: ACM, 2016, pp. 923–934, ISBN: 978-1-4503-3900-1. DOI: [10.1145/2884781.2884817](https://doi.org/10.1145/2884781.2884817). [Online]. Available: <http://doi.acm.org/10.1145/2884781.2884817>.

- [90] C. Priebe, K. Vaswani, and M. Costa, “Enclavedb: A secure database using SGX,” in *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P)*, ser. SP ’18, IEEE, 2018, pp. 264–278. DOI: [10.1109/SP.2018.00025](https://doi.org/10.1109/SP.2018.00025). [Online]. Available: <https://doi.org/10.1109/SP.2018.00025>.
- [91] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy, “Safebricks: Shielding network functions in the cloud,” in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Renton, WA: USENIX Association, 2018, pp. 201–216, ISBN: 978-1-931971-43-0. [Online]. Available: <https://www.usenix.org/conference/nsdi18/presentation/poddar>.
- [92] Apache, *Apache flink: Scalable stream and batch data processing*, <https://flink.apache.org/>, 2017.
- [93] A. Spark, “*spark streaming programming guide*”, <https://spark.apache.org/docs/latest/streaming-programming-guide.html>, 2016.
- [94] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, “Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, ACM, 2014, pp. 90–102.
- [95] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, “The mondrian data engine,” in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, ser. ISCA ’17, Toronto, ON, Canada: ACM, 2017, pp. 639–651, ISBN: 978-1-4503-4892-8. DOI: [10.1145/3079856.3080233](https://doi.org/10.1145/3079856.3080233). [Online]. Available: <http://doi.acm.org/10.1145/3079856.3080233>.
- [96] Arm, *Arm neon technology*, <https://developer.arm.com/technologies/neon>, 2018.
- [97] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, “Sort vs. hash revisited: Fast join implementation on modern multi-core cpus,” *Proceedings of the VLDB Endow.*, vol. 2, no. 2, pp. 1378–1389, 2009, ISSN: 2150-8097. DOI: [10.14778/1687553.1687564](https://doi.org/10.14778/1687553.1687564). [Online]. Available: <https://doi.org/10.14778/1687553.1687564>.
- [98] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, “Multi-core, main-memory joins: Sort vs. hash revisited,” *Proceedings of the VLDB Endow.*, vol. 7, no. 1, pp. 85–96, 2013, ISSN: 2150-8097. DOI: [10.14778/2732219.2732227](https://doi.org/10.14778/2732219.2732227). [Online]. Available: <http://dx.doi.org/10.14778/2732219.2732227>.
- [99] Intel, *Intel threading building blocks*, <https://software.intel.com/en-us/intel-tbb>, 2017.

- [100] J. E. David Goldblatt Dave Watson, *Jemalloc memory allocator*, <http://jemalloc.net/>, 2017.
- [101] A. Aviram, S.-C. Weng, S. Hu, and B. Ford, “Efficient system-enforced deterministic parallelism,” *Proceedings of Commun. of the ACM*, vol. 55, no. 5, pp. 111–119, 2012, ISSN: 0001-0782. DOI: [10.1145/2160718.2160742](https://doi.org/10.1145/2160718.2160742). [Online]. Available: <http://doi.acm.org/10.1145/2160718.2160742>.
- [102] Mohammad Marashi, Tech Crunch, *Satellites are critical for IoT sector to reach its full potential*, <https://techcrunch.com/2017/06/08/satellites-are-critical-for-iot-sector-to-reach-its-full-potential/>, 2017.
- [103] S. Ma, X. Zhang, and D. Xu, “Protracer: Towards practical provenance tracing by alternating between logging and tainting,” in *Proceedings of 23rd Network and Distributed System Security Symposium (NDSS)*, The Internet Society, 2016. [Online]. Available: <http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/protracer-towards-practical-provenance-tracing-alternating-logging-tainting.pdf>.
- [104] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O’Neil, P. O’Neil, A. Rasin, N. Tran, and S. Zdonik, “C-store: A column-oriented dbms,” in *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, ser. VLDB ’05, Trondheim, Norway: VLDB Endowment, 2005, pp. 553–564, ISBN: 1-59593-154-6. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1083592.1083658>.
- [105] Linaro, *Op-tee: Open portable trusted execution environment*, <https://www.op-tee.org/>, 2017.
- [106] iMatix Corporation, *Zeromq*, <http://zeromq.org/>, 2018.
- [107] Facebook, *Folly*, <https://github.com/facebook/folly#folly-facebook-open-source-library>, 2017.
- [108] Marvell Armada 8K family processors, <http://www.marvell.com/embedded-processors/armada-80xx/>.
- [109] NXP Semiconductors, *Imx 7dual family of applications processors datasheet*, <https://www.nxp.com/docs/en/data-sheet/IMX7DCEC.pdf>, 2017.
- [110] Z. Jerzak and H. Ziekow, “The debs 2015 grand challenge,” in *Proceedings of the 9th ACM International Conference on Distributed and Event-Based Systems (DEBS)*, ser. DEBS ’15, Oslo, Norway: ACM, 2015, pp. 266–268, ISBN: 978-1-4503-3286-6. DOI: [10.1145/2675743.2772598](https://doi.org/10.1145/2675743.2772598). [Online]. Available: <http://doi.acm.org/10.1145/2675743.2772598>.

- [111] A. Kolios, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch, “Saber: Window-based hybrid stream processing for heterogeneous architectures,” in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD ’16, San Francisco, California, USA: ACM, 2016, pp. 555–569, ISBN: 978-1-4503-3531-7. DOI: [10.1145/2882903.2882906](https://doi.org/10.1145/2882903.2882906). [Online]. Available: <http://doi.acm.org/10.1145/2882903.2882906>.
- [112] M.-C. Albutiu, A. Kemper, and T. Neumann, “Massively parallel sort-merge joins in main memory multi-core database systems,” *Proceedings of the VLDB Endow.*, vol. 5, no. 10, pp. 1064–1075, 2012, ISSN: 2150-8097. DOI: [10.14778/2336664.2336678](https://doi.org/10.14778/2336664.2336678). [Online]. Available: <http://dx.doi.org/10.14778/2336664.2336678>.
- [113] A. Krettek and M. Winters, “the curious case of the broken benchmark: Revisiting apache flink® vs. databricks runtime”, <https://data-artisans.com/blog/curious-case-broken-benchmark-revisiting-apache-flink-vs-databricks-runtime>, 2017.
- [114] S. Jana, A. Narayanan, and V. Shmatikov, “A scanner darkly: Protecting user privacy from perceptual applications,” in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, ser. SP ’13, Washington, DC, USA: IEEE Computer Society, 2013, pp. 349–363, ISBN: 978-0-7695-4977-4. DOI: [10.1109/SP.2013.31](https://doi.org/10.1109/SP.2013.31). [Online]. Available: <http://dx.doi.org/10.1109/SP.2013.31>.
- [115] J. J. Stephen, S. Savvides, V. Sundaram, M. S. Ardekani, and P. Eugster, “Styx: Stream processing with trustworthy cloud-based execution,” in *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, ser. SoCC ’16, Santa Clara, CA, USA: ACM, 2016, pp. 348–360, ISBN: 978-1-4503-4525-5. DOI: [10.1145/2987550.2987574](https://doi.org/10.1145/2987550.2987574). [Online]. Available: <http://doi.acm.org/10.1145/2987550.2987574>.
- [116] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, “Terra: A virtual machine-based platform for trusted computing,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, ser. SOSP ’03, Bolton Landing, NY, USA: ACM, 2003, pp. 193–206, ISBN: 1-58113-757-5. DOI: [10.1145/945445.945464](https://doi.org/10.1145/945445.945464). [Online]. Available: <http://doi.acm.org/10.1145/945445.945464>.
- [117] N. Santos, H. Raj, S. Saroiu, and A. Wolman, “Using arm trustzone to build a trusted language runtime for mobile applications,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ser. ASPLOS ’14, Salt Lake City, Utah, USA: ACM, 2014, pp. 67–80, ISBN: 978-1-4503-2305-5. DOI: [10.1145/2541940.2541949](https://doi.org/10.1145/2541940.2541949). [Online]. Available: <http://doi.acm.org/10.1145/2541940.2541949>.

- [118] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A.-R. Sadeghi, “Regulating arm trustzone devices in restricted spaces,” in *Proceedings of the 14th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, ser. MobiSys ’16, Singapore, Singapore: ACM, 2016, pp. 413–425, ISBN: 978-1-4503-4269-8. DOI: [10.1145/2906388.2906390](https://doi.org/10.1145/2906388.2906390). [Online]. Available: <http://doi.acm.org/10.1145/2906388.2906390>.
- [119] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, “C-flat: Control-flow attestation for embedded systems software,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, ACM, 2016, pp. 743–754.
- [120] F. Chen, “Cross-platform data integrity and confidentiality with graduated access control,” PhD thesis, The University of British Columbia, 2016.
- [121] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, “Spanedge: Towards unifying stream processing over central and near-the-edge data centers,” in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, 2016, pp. 168–178. DOI: [10.1109/SEC.2016.17](https://doi.org/10.1109/SEC.2016.17).
- [122] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, “Towards wearable cognitive assistance,” in *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, ser. MobiSys ’14, Bretton Woods, New Hampshire, USA: ACM, 2014, pp. 68–81, ISBN: 978-1-4503-2793-0. DOI: [10.1145/2594368.2594383](https://doi.org/10.1145/2594368.2594383). [Online]. Available: <http://doi.acm.org/10.1145/2594368.2594383>.
- [123] K. Bhardwaj, M. W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan, “Fast, scalable and secure onloading of edge functions using airbox,” in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, 2016, pp. 14–27. DOI: [10.1109/SEC.2016.15](https://doi.org/10.1109/SEC.2016.15).
- [124] S. Nastic, H. L. Truong, and S. Dustdar, “A middleware infrastructure for utility-based provisioning of iot cloud systems,” in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, 2016, pp. 28–40. DOI: [10.1109/SEC.2016.35](https://doi.org/10.1109/SEC.2016.35).
- [125] W. Thies, M. Karczmarek, and S. P. Amarasinghe, “Streamit: A language for streaming applications,” in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC ’02, London, UK, UK: Springer-Verlag, 2002, pp. 179–196, ISBN: 3-540-43369-4. [Online]. Available: <http://dl.acm.org/citation.cfm?id=647478.727935>.
- [126] M. C. Stanley Zdonik Michael Stonebraker, *Streambase systems*, <http://www.tibco.com/products/tibco-streambase>, 2017.
- [127] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, “Telegraphcq: Continuous dataflow processing,” in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, ACM, 2003, pp. 668–668.

- [128] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos, “Semantics of data streams and operators,” in *Proceedings of the 10th International Conference on Database Theory (ICDT)*, ser. ICDT’05, Edinburgh, UK: Springer-Verlag, 2005, pp. 37–52, ISBN: 3-540-24288-0, 978-3-540-24288-8. DOI: [10.1007/978-3-540-30570-5_3](https://doi.org/10.1007/978-3-540-30570-5_3). [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30570-5_3.
- [129] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, “Gigascope: A stream database for network applications,” in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, ACM, 2003, pp. 647–651.
- [130] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, “Timestream: Reliable stream computation in the cloud,” in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, ser. EuroSys ’13, Prague, Czech Republic: ACM, 2013, pp. 1–14, ISBN: 978-1-4503-1994-2. DOI: [10.1145/2465351.2465353](https://doi.org/10.1145/2465351.2465353). [Online]. Available: <http://doi.acm.org/10.1145/2465351.2465353>.
- [131] Twitter, *Heron*, <https://twitter.github.io/heron/>, 2017.
- [132] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, “Drizzle: Fast and adaptable stream processing at scale,” in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, ser. SOSP ’17, Shanghai, China: ACM, 2017, pp. 374–389, ISBN: 978-1-4503-5085-3. DOI: [10.1145/3132747.3132750](https://doi.org/10.1145/3132747.3132750). [Online]. Available: <http://doi.acm.org/10.1145/3132747.3132750>.
- [133] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, “Tensorflow: A system for large-scale machine learning,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283, ISBN: 978-1-931971-33-1. [Online]. Available: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi>.
- [134] Tencent, *Tencent ncnn framework*, <https://github.com/Tencent/ncnn>.
- [135] Arm, *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/>.
- [136] Arm, *Open Source Mali Bifrost GPU Kernel Drivers*, <https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-drivers/bifrost-kernel>.
- [137] S. Lee, Y. Kim, J. Kim, and J. Kim, “Stealing webpages rendered on your browser by exploiting gpu vulnerabilities,” in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 19–33. DOI: [10.1109/SP.2014.9](https://doi.org/10.1109/SP.2014.9).

- [138] R. D. Pietro, F. Lombardi, and A. Villani, “Cuda leaks: A detailed hack for cuda and a (partial) fix,” vol. 15, no. 1, Jan. 2016, ISSN: 1539-9087. DOI: [10.1145/2801153](https://doi.org/10.1145/2801153). [Online]. Available: <https://doi.org/10.1145/2801153>.
- [139] Z. Yao, Z. Ma, Y. Liu, A. Amiri Sani, and A. Chandramowlishwaran, “Sugar: Secure gpu acceleration in web browsers,” ser. ASPLOS ’18, Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 519–534, ISBN: 9781450349116. DOI: [10.1145/3173162.3173186](https://doi.org/10.1145/3173162.3173186). [Online]. Available: <https://doi.org/10.1145/3173162.3173186>.
- [140] Arm, *Vulkan SDK for Android*, <https://github.com/ARM-software/vulkan-sdk>.
- [141] Google, *Tensorflow GPU support*, <https://www.tensorflow.org/install/gpu>.
- [142] Khronos Group, *Khronos Conformant Products*, <https://www.khronos.org/conformance/adopters/conformant-products>.
- [143] NVIDIA, *CUDA Compatibility*, <https://docs.nvidia.com/deploy/cuda-compatibility/index.html>.
- [144] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, “A first look at deep learning apps on smartphones,” in *The World Wide Web Conference*, ser. WWW ’19, San Francisco, CA, USA: Association for Computing Machinery, 2019, pp. 2125–2136, ISBN: 9781450366748. DOI: [10.1145/3308558.3313591](https://doi.org/10.1145/3308558.3313591). [Online]. Available: <https://doi.org/10.1145/3308558.3313591>.
- [145] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, L. T. X. Phan, M. Sherr, and W. Zhou, “Detecting covert timing channels with time-deterministic replay,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 541–554, ISBN: 978-1-931971-16-4. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chen_ang.
- [146] Y. Ji, S. Lee, M. Fazzini, J. Allen, E. Downing, T. Kim, A. Orso, and W. Lee, “Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking,” in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 1705–1722, ISBN: 978-1-939133-04-5. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity18/presentation/jia-yang>.
- [147] M. Yan, Y. Shalabi, and J. Torrellas, “Replayconfusion: Detecting cache-based covert channel attacks using record and replay,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–14.
- [148] Arm, *Arm Compute Library*, <https://github.com/ARM-software/ComputeLibrary>.
- [149] H. Perkins, *Opencl library to train deep convolutional networks*, <https://github.com/hughperkins/DeepCL>.

- [150] Quake3 World, *Quak3: Demo System*, <https://www.quake3world.com/q3guide/demos.html>.
- [151] id-Software, *DOOM-3-BFG: LoadGLSLShader()*, https://github.com/id-Software/DOOM-3-BFG/blob/master/neo/renderer/RenderProgs_GLSL.cpp#L960.
- [152] Alyssa Rosenzweig, *Dissecting the Apple M1 GPU, part II*, <https://rosenzweig.io/blog/asahi-gpu-part-2.html>.
- [153] Android kernel, *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.
- [154] Linux, *drm/v3d Broadcom V3D Graphics Driver*, <https://www.kernel.org/doc/html/latest/gpu/v3d.html>.
- [155] Linux, *drm/vc4 Broadcom VC4 Graphics Driver: submit_cl()*, https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/vc4/vc4_gem.c#L369.
- [156] 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)*, Santa Clara, CA: USENIX Association, Feb. 2020, pp. 817–833, ISBN: 978-1-939133-13-7. [Online]. Available: <https://www.usenix.org/conference/nsdi20/presentation/hunt>.
- [157] W. Kwon, G.-I. Yu, E. Jeong, and B.-G. Chun, “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., 2020, pp. 8343–8354. [Online]. Available: <https://proceedings.neurips.cc/paper/2020/file/5f0ad4db43d8723d18169b2e4817a160-Paper.pdf>.
- [158] TensorFlow, *Example NNs by TensorFlow*, https://www.tensorflow.org/lite/guide/hosted_models.
- [159] Y. Ioannou, D. P. Robertson, D. Zikic, P. Kotschieder, J. Shotton, M. Brown, and A. Criminisi, “Decision forests, convolutional networks and the models in-between,” *CoRR*, vol. abs/1603.01250, 2016. arXiv: [1603.01250](https://arxiv.org/abs/1603.01250). [Online]. Available: <http://arxiv.org/abs/1603.01250>.
- [160] 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)*, Renton, WA: USENIX Association, Jul. 2019, pp. 537–554, ISBN: 978-1-939133-03-8. [Online]. Available: <https://www.usenix.org/conference/atc19/presentation/park-heejin>.

- [161] 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*, ACM, 2019, pp. 455–468, ISBN: 978-1-4503-6240-5. DOI: [10.1145/3297858.3304021](https://doi.org/10.1145/3297858.3304021).
- [162] NVIDIA, *Host1x hardware description*, <http://http.download.nvidia.com/tegra-public-appnotes/host1x.html>.
- [163] Android kernel, *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.
- [164] Android kernel, *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.
- [165] Android kernel, *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.
- [166] Android kernel, *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.
- [167] Linux, *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.
- [168] Linux, *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.
- [169] Linux, *drm/v3d Broadcom V3D Graphics Driver: v3d_reset()*, https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/v3d/v3d_gem.c#L110.
- [170] Linux, *drm/v3d Broadcom V3D Graphics Driver: v3d_irq()*, https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/v3d/v3d_irq.c#L85.
- [171] Linux, *drm/vc4 Broadcom VC4 Graphics Driver*, <https://www.kernel.org/doc/html/latest/gpu/vc4.html>.
- [172] Linux, *drm/vc4 Broadcom VC4 Graphics Driver: vc4_reset()*, https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/vc4/vc4_gem.c#L286.
- [173] Linux, *drm/vc4 Broadcom VC4 Graphics Driver: vc4_irq()*, https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/vc4/vc4_irq.c#L196.

- [174] NVIDIA, *NVIDIA Tegra Linux Driver*, <https://nv-tegra.nvidia.com/gitweb/?p=linux-nvgpu.git;a=summary>.
- [175] NVIDIA, *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#l140>.
- [176] NVIDIA, *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#l1318>.
- [177] NVIDIA, *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>.
- [178] NVIDIA, *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>.
- [179] NVIDIA, *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=6dc57fec39c444e4c4448be61dd19c55693daf1#l39.
- [180] Linux, *Qualcomm adreno graphics driver*, <https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/msm/adreno>.
- [181] Linux, *Qualcomm adreno graphics driver: a6xx_flush()*, https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/msm/adreno/a6xx_gpu.c#L54.
- [182] Linux, *Qualcomm Adreno Graphics Driver: a6xx_submit()*, https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/msm/adreno/a6xx_gpu.c#L140.
- [183] Linux, *Qualcomm Adreno Graphics Driver: msm_gpummu_map()*, https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/msm/msm_gpummu.c#L28.
- [184] Linux, *Qualcomm Adreno Graphics Driver: a6xx_recover()*, https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/msm/adreno/a6xx_gpu.c#L934.

- [185] Linux, *Qualcomm Adreno Graphics Driver: a6xx_irq()*, https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/msm/adreno/a6xx_gpu.c#L1046.
- [186] Linux, *drm/v3d Broadcom V3D Graphics Driver: register accessors*, https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/v3d/v3d_drv.h#L170.
- [187] Android kernel, *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.
- [188] *In-kernel memory mapped i/o tracing*, <https://www.kernel.org/doc/html/latest/trace/mmiotrace.html>.
- [189] Linux, *drm/v3d Broadcom V3D Graphics Driver: v3d_clean_caches()*, https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/v3d/v3d_gem.c#L189.
- [190] Android kernel, *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.
- [191] Linux, *drm/v3d Broadcom V3D Graphics Driver: wait_for()*, https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/v3d/v3d_drv.h#L290.
- [192] NVIDIA, *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>.
- [193] NVIDIA, *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.
- [194] Android kernel, *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.
- [195] NVIDIA, *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.
- [196] Arm, *ArmNN*, <https://github.com/ARM-software/armnn>.

- [197] Jean-loup Gailly and Mark Adler, *A Massively Spiffy Yet Delicately Unobtrusive Compression Library*, <https://zlib.net/>.
- [198] rsta2, *A C++ bare metal environment for Raspberry Pi with USB (32 and 64 bit)*, <https://github.com/rsta2/circle/>.
- [199] C. Xu, F. X. Lin, Y. Wang, and L. Zhong, “Automated os-level device runtime power management,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’15, Istanbul, Turkey: ACM, 2015, pp. 239–252, ISBN: 978-1-4503-2835-7. DOI: [10.1145/2694344.2694360](https://doi.org/10.1145/2694344.2694360). [Online]. Available: <http://doi.acm.org/10.1145/2694344.2694360>.
- [200] Raspberry Pi Foundation, *RaspberryPi Firmware Mailbox property interface*, <https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface>.
- [201] L. Foundation, *Data plane development kit (DPDK)*, <http://www.dpdk.org>.
- [202] L. Foundation, *The user space io (UIO)*, <https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html>.
- [203] *CVE-2014-0972: Unprivileged gpu command streams can change the iommu page table*, <https://nvd.nist.gov/vuln/detail/CVE-2014-0972>, 2014.
- [204] *CVE-2017-18643: Information disclosure on samsung gpu*, <https://nvd.nist.gov/vuln/detail/CVE-2017-18643>, 2017.
- [205] *CVE-2020-11179: Qualcomm adreno gpu ringbuffer corruption / protected mode bypass*, <https://nvd.nist.gov/vuln/detail/CVE-2020-11179>, 2020.
- [206] *CVE-2019-14615: Information leakage vulnerability on the intel integrated gpu architecture*, <https://nvd.nist.gov/vuln/detail/CVE-2019-14615>, 2019.
- [207] *CVE-2019-10520: Out of memory in snapdragon*, <https://nvd.nist.gov/vuln/detail/cve-2019-10520>, 2019.
- [208] *CVE-2019-5068: Exploitable shared memory permission vulnerability in mesa 3d graphics library*, <https://nvd.nist.gov/vuln/detail/CVE-2019-5068>, 2019.
- [209] *CVE-2018-6253: Vulnerability in the opengl usermode drivers*, <https://nvd.nist.gov/vuln/detail/CVE-2018-6253>, 2018.
- [210] Brendon Boshell, *Average App File Size: Data for Android and iOS Mobile Apps*, <https://sweetpricing.com/blog/2017/02/average-app-file-size/>.

- [211] A. Developer, *Mali offline compiler*, <https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/components/mali-offline-compiler>.
- [212] Tencent, *ncnn Pipeline Cache*, <https://github.com/Tencent/ncnn/blob/master/src/pipelinecache.cpp/>.
- [213] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, “Reran: Timing- and touch-sensitive record and replay for android,” in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 72–81.
- [214] Z. Qin, Y. Tang, E. Novak, and Q. Li, “Mobiplay: A remote execution based record-and-replay tool for mobile applications,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16, Austin, Texas: Association for Computing Machinery, 2016, pp. 571–582, ISBN: 9781450339001. DOI: [10.1145/2884781.2884854](https://doi.org/10.1145/2884781.2884854). [Online]. Available: <https://doi.org/10.1145/2884781.2884854>.
- [215] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan, “Mahimahi: Accurate record-and-replay for HTTP,” in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, Santa Clara, CA: USENIX Association, Jul. 2015, pp. 417–429, ISBN: 978-1-931971-225. [Online]. Available: <https://www.usenix.org/conference/atc15/technical-session/presentation/netravali>.
- [216] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, “Revirt: Enabling intrusion analysis through virtual-machine logging and replay,” *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 211–224, Dec. 2003, ISSN: 0163-5980. DOI: [10.1145/844128.844148](https://doi.org/10.1145/844128.844148). [Online]. Available: <https://doi.org/10.1145/844128.844148>.
- [217] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, “Ofrewind: Enabling record and replay troubleshooting for networks,” in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC’11, Portland, OR: USENIX Association, 2011, p. 29.
- [218] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang, “R2: An application-level kernel for record and replay,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08, San Diego, California: USENIX Association, 2008, pp. 193–208.
- [219] alyssa rosenzweig, *Dissecting the apple m1 gpu*, <https://rosenzweig.io/blog/asahi-gpu-part-1.html>.
- [220] Grate, *Open source reverse-engineering tools aiming at nvidia tegra2+3d engine*, <https://github.com/grate-driver/grate>.
- [221] T. M. 3. G. Library, *Panfrost*, <https://docs.mesa3d.org/drivers/panfrost.html>.

- [222] ARM-software, *Software for capturing gles calls of an application and replaying them on a different device*, <https://github.com/ARM-software/patrace>.
- [223] 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)*, Carlsbad, CA: USENIX Association, Oct. 2018, pp. 681–696, ISBN: 978-1-939133-08-3. [Online]. Available: <https://www.usenix.org/conference/osdi18/presentation/volos>.
- [224] J. Zhu, R. Hou, X. Wang, W. Wang, J. Cao, B. Zhao, Z. Wang, Y. Zhang, J. Ying, L. Zhang, and D. Meng, “Enabling rack-scale confidential computing using heterogeneous trusted execution environment,” in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1450–1465. DOI: [10.1109/SP40000.2020.00054](https://doi.org/10.1109/SP40000.2020.00054).
- [225] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, “Gpuvmm: Why not virtualizing gpus at the hypervisor?,” Philadelphia, PA: USENIX Association, Jun. 2014, pp. 109–120, ISBN: 978-1-931971-10-2. [Online]. Available: <https://www.usenix.org/conference/atc14/technical-sessions/presentation/suzuki>.
- [226] J. Duato, A. J. Pena, F. Silla, J. C. Fernandez, R. Mayo, and E. S. Quintana-Orti, “Enabling cuda acceleration within virtual machines using rcuda,” in *2011 18th International Conference on High Performance Computing*, IEEE, 2011, pp. 1–10.
- [227] H. Yu, A. M. Peters, A. Akshintala, and C. J. Rossbach, “Ava: Accelerated virtualization of accelerators,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 807–825, ISBN: 9781450371025. DOI: [10.1145/3373376.3378466](https://doi.org/10.1145/3373376.3378466). [Online]. Available: <https://doi.org/10.1145/3373376.3378466>.
- [228] M. Dowty and J. Sugerman, “Gpu virtualization on vmware’s hosted i/o architecture,” *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 3, pp. 73–82, Jul. 2009, ISSN: 0163-5980. DOI: [10.1145/1618525.1618534](https://doi.org/10.1145/1618525.1618534). [Online]. Available: <https://doi.org/10.1145/1618525.1618534>.
- [229] S. Kim, S. Oh, and Y. Yi, “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*, ser. HotMobile ’21, Virtual, United Kingdom: Association for Computing Machinery, 2021, pp. 57–63, ISBN: 9781450383233. DOI: [10.1145/3446382.3448606](https://doi.org/10.1145/3446382.3448606). [Online]. Available: <https://doi.org/10.1145/3446382.3448606>.
- [230] S. (Xi, Y. Yao, K. Bhardwaj, P. Whatmough, G.-Y. Wei, and D. Brooks, “Smaug: End-to-end full-stack simulation infrastructure for deep learning workloads,” *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, Nov. 2020, ISSN: 1544-3566. DOI: [10.1145/3424669](https://doi.org/10.1145/3424669). [Online]. Available: <https://doi.org/10.1145/3424669>.

- [231] 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*, vol. abs/1807.00969, 2018. arXiv: [1807.00969](https://arxiv.org/abs/1807.00969). [Online]. Available: [http://arxiv.org/abs/1807.00969](https://arxiv.org/abs/1807.00969).
- [232] F. Tramèr and D. Boneh, “Slalom: Fast, verifiable and private execution of neural networks in trusted hardware,” in *International Conference on Learning Representations*, 2019. [Online]. Available: <https://openreview.net/forum?id=rJVorjCcKQ>.
- [233] F. Brasser, D. Gens, P. Jauernig, A.-R. 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. [Online]. Available: <https://www.ndss-symposium.org/ndss-paper/sanctuary-arming-trustzone-with-user-space-enclaves/>.
- [234] 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*, ser. MobiSys ’18, Munich, Germany: Association for Computing Machinery, 2018, pp. 1–13, ISBN: 9781450357203. DOI: [10.1145/3210240.3210334](https://doi.org/10.1145/3210240.3210334). [Online]. Available: <https://doi.org/10.1145/3210240.3210334>.
- [235] 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*, ser. MobiSys ’17, Niagara Falls, New York, USA: Association for Computing Machinery, 2017, pp. 197–210, ISBN: 9781450349284. DOI: [10.1145/3081333.3081346](https://doi.org/10.1145/3081333.3081346). [Online]. Available: <https://doi.org/10.1145/3081333.3081346>.
- [236] 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)*, USENIX Association, Aug. 2020, pp. 1039–1056, ISBN: 978-1-939133-17-5. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity20/presentation/poddar>.
- [237] H. Park and F. X. Lin, “Tinystack: A minimal GPU stack for client ML,” *CoRR*, vol. abs/2105.05085, 2021. arXiv: [2105.05085](https://arxiv.org/abs/2105.05085). [Online]. Available: <https://arxiv.org/abs/2105.05085>.
- [238] 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*, ser. HotOS ’19, Bertinoro, Italy: Association for Computing Machinery, 2019, pp. 66–72, ISBN: 9781450367271. DOI: [10.1145/3317550.3321446](https://doi.org/10.1145/3317550.3321446). [Online]. Available: <https://doi.org/10.1145/3317550.3321446>.

- [239] 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 *Proceedings of The 33rd International Conference on Machine Learning*, M. F. Balcan and K. Q. Weinberger, Eds., ser. Proceedings of Machine Learning Research, vol. 48, New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 201–210. [Online]. Available: <https://proceedings.mlr.press/v48/gilad-bachrach16.html>.
- [240] 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*, ser. ISCA '19, Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 487–498, ISBN: 9781450366694. DOI: [10.1145/3307650.3322251](https://doi.org/10.1145/3307650.3322251). [Online]. Available: <https://doi.org/10.1145/3307650.3322251>.
- [241] GadgetVersus, *Various lists of graphics cards*, <https://gadgetversus.com/graphics-card/>.
- [242] Kashish Kumawat, Tech Centurion, *Mobile GPU Rankings 2021 (Adreno/Mali/PowerVR)*, <https://www.techcenturion.com/mobile-gpu-rankings>.
- [243] 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*, ser. MobiSys '15, Florence, Italy: Association for Computing Machinery, 2015, pp. 121–135, ISBN: 9781450334945. DOI: [10.1145/2742647.2742657](https://doi.org/10.1145/2742647.2742657). [Online]. Available: <https://doi.org/10.1145/2742647.2742657>.
- [244] 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*, ser. EuroSys '11, Salzburg, Austria: ACM, 2011, pp. 301–314, ISBN: 978-1-4503-0634-8. DOI: [10.1145/1966445.1966473](https://doi.org/10.1145/1966445.1966473). [Online]. Available: <http://doi.acm.org/10.1145/1966445.1966473>.
- [245] 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)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 291–307, ISBN: 978-1-939133-04-5. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity18/presentation/talebi>.
- [246] N. Santos, K. P. Gummadi, and R. Rodrigues, “Towards trusted cloud computing,” in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, ser. HotCloud'09, San Diego, California: USENIX Association, 2009.

- [247] 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)*, Bellevue, WA: USENIX Association, Aug. 2012, pp. 175–188, ISBN: 978-931971-95-9. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/santos>.
- [248] 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)*, 2016, pp. 255–262.
- [249] Arm, *Open Source Mali Midgard GPU Kernel Drivers*, <https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-drivers/midgard-kernel>.
- [250] 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.
- [251] 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.
- [252] Linux, *delays - Information on the various kernel delay / sleep mechanisms*, <https://www.kernel.org/doc/Documentation/timers/timers-howto.txt/>, 2021.
- [253] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, “Comet: Code offload by migrating execution transparently,” ser. OSDI’12, Hollywood, CA, USA: USENIX Association, 2012, pp. 93–106, ISBN: 9781931971966.
- [254] Linux, *The Kernel Concurrency Sanitizer (KCSAN)*, <https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html>, 2021.
- [255] 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*, ser. MobiSys ’14, Bretton Woods, New Hampshire, USA: Association for Computing Machinery, 2014, pp. 259–272, ISBN: 9781450327930. DOI: 10.1145/2594368.2594370. [Online]. Available: <https://doi.org/10.1145/2594368.2594370>.
- [256] Android kernel, *Arm Bifrost Graphics Driver: JS_CONFIG_START_FLUSH_CLEAN_INVALIDATE*, https://android.googlesource.com/kernel/arm64/+refs/tags/android-11.0.0_r0.67/drivers/gpu/arm/midgard/mali_midg_regmap.h#345.
- [257] Linux, *drm/vc4 Broadcom VC4 Graphics Driver: vc4_flush_cache()*, https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/vc4/vc4_gem.c#L429.

- [258] 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.
- [259] Clang, *a C language family frontend for LLVM*, <https://clang.llvm.org/>.
- [260] Global Platform, *Globalplatform made simple guide: Trusted execution environment*, 2016.
- [261] 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.*, vol. 2016, p. 980, 2016.
- [262] S. Hemminger, “Network emulation with netem,” Linux conf au, 2005.
- [263] 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*, ser. CoNEXT ’18, Heraklion, Greece: Association for Computing Machinery, 2018, pp. 307–319, ISBN: 9781450360807. DOI: [10.1145/3281411.3281430](https://doi.org/10.1145/3281411.3281430). [Online]. Available: <https://doi.org/10.1145/3281411.3281430>.
- [264] 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)*, 2019, pp. 270–277. DOI: [10.1109/DSC.2019.00048](https://doi.org/10.1109/DSC.2019.00048).
- [265] 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*, ser. MobiSys ’17, Niagara Falls, New York, USA: Association for Computing Machinery, 2017, pp. 332–344, ISBN: 9781450349284. DOI: [10.1145/3081333.3081348](https://doi.org/10.1145/3081333.3081348). [Online]. Available: <https://doi.org/10.1145/3081333.3081348>.
- [266] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, “Avatar: A framework to support dynamic security analysis of embedded systems’ firmwares,” in *NDSS*, 2014.
- [267] 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.: USENIX Association, Aug. 2015. [Online]. Available: <https://www.usenix.org/conference/woot15/workshop-program/presentation/koscher>.

- [268] 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*, ser. MobiSys ’21, Virtual Event, Wisconsin: Association for Computing Machinery, 2021, pp. 122–135, ISBN: 9781450384438. DOI: [10.1145/3458864.3467887](https://doi.org/10.1145/3458864.3467887). [Online]. Available: <https://doi.org/10.1145/3458864.3467887>.
- [269] 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: USENIX Association, Feb. 1999. [Online]. Available: <https://www.usenix.org/conference/osdi-99/automatic-io-hint-generation-through-speculative-execution>.
- [270] 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*, ser. SOSP ’05, Brighton, United Kingdom: Association for Computing Machinery, 2005, pp. 191–205, ISBN: 1595930795. DOI: [10.1145/1095810.1095829](https://doi.org/10.1145/1095810.1095829). [Online]. Available: <https://doi.org/10.1145/1095810.1095829>.
- [271] 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*, ser. OOPSLA ’11, Portland, Oregon, USA: Association for Computing Machinery, 2011, pp. 259–274, ISBN: 9781450309400. DOI: [10.1145/2048066.2048088](https://doi.org/10.1145/2048066.2048088). [Online]. Available: <https://doi.org/10.1145/2048066.2048088>.
- [272] 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: USENIX Association, Apr. 2008. [Online]. Available: <https://www.usenix.org/conference/nsdi-08/remus-high-availability-asynchronous-virtual-machine-replication>.
- [273] 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*, ser. OSDI’08, San Diego, California: USENIX Association, 2008, pp. 29–42.
- [274] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “Maui: Making smartphones last longer with code offload,” ser. MobiSys ’10, San Francisco, California, USA: ACM, 2010, pp. 49–62, ISBN: 978-1-60558-985-5. DOI: [10.1145/1814433.1814441](https://doi.org/10.1145/1814433.1814441). [Online]. Available: <http://doi.acm.org/10.1145/1814433.1814441>.

- [275] 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*, ser. MobiSys '21, Virtual Event, Wisconsin: Association for Computing Machinery, 2021, pp. 94–108, ISBN: 9781450384438. DOI: [10.1145/3458864.3466628](https://doi.org/10.1145/3458864.3466628). [Online]. Available: <https://doi.org/10.1145/3458864.3466628>.