Operating and Runtime Systems towards an Efficient and Secure Edge

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

by

Liwei Guo

December 2022

© 2022 Liwei Guo

Abstract

Located near user data, the edge is a preferred place for executing latency and security-sensitive tasks such as sensor data harvesting and processing. For instance, a smart speaker interacts with user speeches impromptu; it does so by running Natural Language Processing (NLP) inferences purely on device without transmitting the captured personal audio to the cloud for crucially preserving user privacy.

However, existing operating and runtime systems are inadequate in executing such tasks efficiently or securely. First, they suffer from poor efficiency. Design inefficacies in kernel and machine learning inference runtime have incurred large CPU idle epochs and correspondingly led to significant energy and memory inefficiency, which are crucial for the resource-constrained edge. Second, they lack support for Trusted Execution Environment (TEE). Designed to isolate and protect platform resources at the lowest level, a TEE (e.g. Arm TrustZone) executes security-sensitive code, oblivious to the OS. Yet, without mature filesystems or device drivers, the TEE inevitably relies on the OS by exposing the data and control path to the OS for execution on its behalf, creating security and privacy loopholes.

This dissertation shows that, by co-designing systems software with hardware and incorporating the app knowledge, it is possible to foster greater efficiency and security at the edge.

To this end, I present five systems in two parts. The first part addresses the efficiency problem. Starting with *Power Sandbox*, I endow the knowledge of energy consumption to apps at OS level, allowing them to reason about their own power and adapt towards greater efficiency accordingly. Then I will present *Transkernel* and *STI*, two systems that address the execution inefficiencies in kernel and machine learning runtime respectively. Through the two systems, I show that the key to greater efficiency is to eliminate CPU idling by specializing systems with respect to edge workloads and hardware. The second part introduces two systems *Driverlet* and *Enigma*, which enable TEE access to mature filesystems and complex devices for the first time. I will show that by intercepting at proper hardware/software boundaries, it is possible not only enable the practical use of TEE but also do so in a secure and private way. Together, the five systems compose a holistic tapestry of system designs towards a more efficient and secure edge.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Liwei Guo

Liwei Guo

This dissertation has been read and approved by the Examining Committee:

Dr. Felix Xiaozhu Lin, Advisor

Dr. Yangfeng Ji, Committee Chair

Dr. Mircea Stan

Dr. David Evans

Dr. Charlie Y. Hu

Accepted for the School of Engineering and Applied Science:

Jeniffer L. West, Dean, School of Engineering and Applied Science

December 2022

To my parents for their endless love and support.

Acknowledgments

My Ph.D. journey is a long venture both temporally and spatially, but I am grateful to the following people who have made it easy for me.

First and foremost, I owe my deepest gratitude to my advisor, Professor Felix Xiaozhu Lin. Being the Mastersmith and honing my research skills through the six-year apprenticeship is only the least he has done for me. He has been a caring brother, navigating me through my early tough years in the U.S., caring for my mental and physical wellness, and keeping me on my toes before I went astray. Moreover, by climbing the Himalayas of systems research with him, he has shown me the grandeur of computer systems, given me the great vision towards the future, and helped me develop a fine taste of doing systems research. I am forever grateful for his teaching, on which the eaglet shall finally soar.

My gratitude extends to my committee members, Professor Yangfeng Ji, Professor Mircea Stan, Professor David Evans of UVA and Professor Charlie Hu of Purdue. I thank them for their scrutiny and insightful feedback of the STI project, which helps immensely with its publication.

I spent the first four years of my Ph.D. at Purdue, where I have worked with many wonderful people. First, the crew members of Crossroads Systems Exploration Lab (XSEL): Hongyu Miao, Tiantu Xu, Heejin Park, and Shuang Zhai. The days of us hanging out and those Friday chitchats will be my most precious memory. I thank Mengwei Xu. During his one-year visit at XSEL, his knowledge in mobile intelligence brings a breath of fresh air to me and he has been a good gym partner who helps me improve a lot. I thank Yizhou Shan for deep conversations over OSes and kernels. I also thank my collaborators, Yi Qiao, Kaiyang Zhao, and Professor Yiying Zhang for helping the Transkernel and Enigma project.

During the remaining two years at UVA, I thank Meng Wang, who guided me around Charlottesville when I first arrived and had many technical discussions over security research with me afterwards. I am also fortunate to work with Felix's new batch of students: Wonkyo Choe, Rongxiang Wang, Xu Ouyang, Afsara Benazir, and Changhong Yang, who help me better prepare the final defense; I wish I could have stayed longer to pass along all my experiences in systems research.

I would also like to thank Renyu Guo (IU), Zhengyang Wang (Vanderbilt), Xiujia Yang (UIUC), and Jiacheng Li (Purdue), who were also Ph.D. students and made my journey not alone. Weekend gaming nights with them are important to my sanity, which refuel my energy and help me keep on grinding.

Last but not least, I am so grateful to my parents Hong Guo and Hua Xiong. They shape who I am today and always offer the endless and unconditional love and support to me throughout my long years of study. The journey would not have been possible without them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Emerging edge platforms are equipped with increasingly abundant and heterogeneous computing resources. Yet, the existing systems software is inadequate in fully utilizing them. To address the problem, my dissertation features five novel systems designs which help these edge platforms run more efficiently, securely and privately.

1.1 Background: workloads and hardware at the edge

Definition. This dissertation targets edge devices, which are low-cost embedded/mobile Arm SoCs running commodity, general purpose Linux kernels. Examples include smart speakers [2], cameras [3], watches [4], as well as phones [5]. An edge device may have multiple tenant apps but is entitled to only a single user or a small list of users. For instance, a mobile phone has many apps but is often used by a single user; a smart speaker runs only an intelligent personal assistant (IPA) app and is shared by household members.

Workloads. Sitting at the frontier of the data flow (hence the "edge"), edge devices are responsible for harvesting, processing, and persisting the data. Example workloads include:

• *Harvesting.* Deployed in the wild, an edge device periodically collects data from physical world, e.g. images, temperature, humidity. To save energy, the edge device often suspends between harvesting intervals and only resumes to harvest data from sensors.

• *Processing.* As soon as an edge device has harvested the data, it may process the data on-device. Doing so has two key benefits: 1) low latency, as it saves network roundtrips; 2) privacy, since no data leaks to the cloud. For example, the smart speaker and camera may execute machine learning inference on-device for speech recognition and object detection without transmitting any user audio/images to the cloud.

• *Persisting.* An edge device often needs to persist the harvested data and processed results in local storage in case of power loss; for reliability, it relies on mature filesystems to manage the data blocks.

Hardware. Driven by aforementioned workloads, edge devices are not only equipped with rich, multi-instance, IO devices such as MMC, USB, Wi-Fi NICs, and CSI cameras, but also are equipped with abundant and heterogeneous computing resources. For instance, they often have microcontroller-like peripheral cores (e.g. Cortex M3 [6]) which run alongside their Arm CPUs; with a much simpler micro-architecture, the heterogeneous cores achieve far better energy efficiency (e.g. 100× smaller idle power) than CPUs, making them an ideal place to run infrequent sensor data harvesting tasks, e.g. step counting. To process data (e.g. by executing ML inferences), they are often armed with various accelerators (e.g. DSPs, GPUs, NPUs), which execute math operators fast and efficiently by exploiting high data parallelism. For security, they employ Arm TrustZone as the Trusted Execution Environment (TEE) [7] which partitions the SoCs' resources among the secure and normal world, running in a time-sharing fashion. By design, the secure world isolates security and privacy-sensitive execution (e.g. sensor data acquisition) from the normal world OS (e.g. Linux). Figure 1.1 shows the software/hardware stack of such a heterogeneous edge platform.

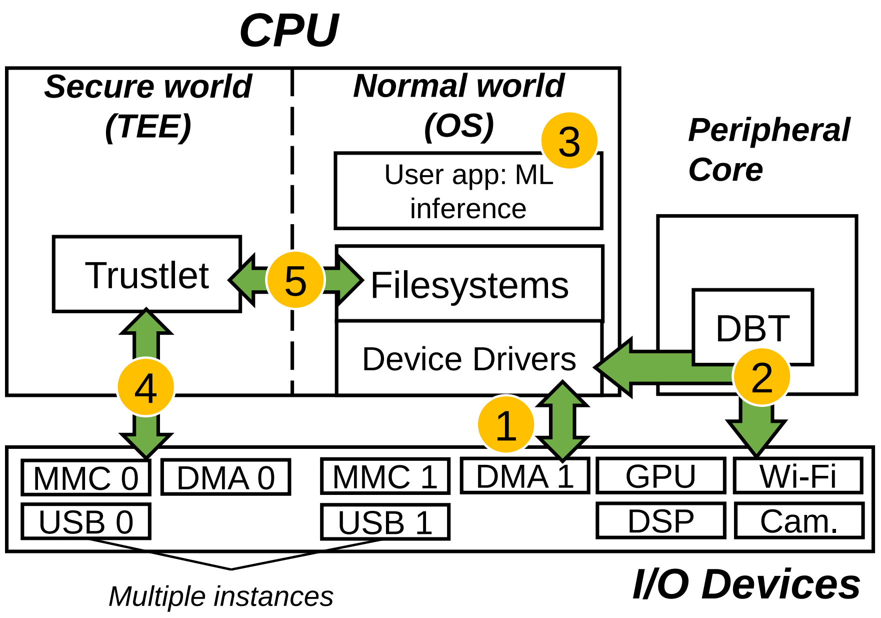


Figure 1.1: The five systems this dissertation presents towards a more efficient and secure edge platform. The figure shows their corresponding positions in the software/hardware stack. ① : Psbox (Chapter 2), ② : Transkernel (Chapter 3), ③ : STI (Chapter 4), ④ : Driverlet (Chapter 5), ⑤ : Enigma (Chapter 6)

1.2 Challenges to systems software (OS and Runtime)

Despite the rich (sometimes over-provisioned) hardware resources, existing systems software on edge platforms fails to fully utilize them in their workloads and faces two key inadequacies.

First, poor energy efficiency. As edge platforms are often battery-powered, energy efficiency is of their paramount importance. Yet, even with accelerators and heterogeneous cores, they still suffer from energy drain and have poor energy efficiency, as demonstrated by [8, 9, 10]. We discover two key bottlenecks in existing systems design:

• **Lack of energy accounting facility.** Existing systems provide no reliable, accurate energy accounting facilities. As a result, there is no means for apps running on the edge platform to tell their actual power consumption and adapt towards better efficiency, which is a long desired goal [11].

• **Large CPU idle epochs.** CPU execution is not efficient, either. In kernel execution, there exist many small, yet accumulatively large CPU idle epochs [12]; they are due to waiting for voltage ramp up and power state transitions, which are bound by physical factors. In ML inference, the existing runtime designs model loading and computation in separation; the result is, to execute an inference task, a CPU often times must wait IO (i.e. model loading) to complete before proceeding to computation, incurring long delays [13] and energy waste.

Second, inadequacy in using TEE. As an isolated execution environment, TrustZone TEE and its runtime kernel (e.g. OPTEE-OS [14]) resembles much of a baremetal environment, lacking essential services such as filesystems and device drivers. To use it, apps of the TEE (i.e. trustlets) must forward the file calls and IO job requests to a normal world OS and rely on the software stack of a commodity kernel. This creates two security risks which break the integrity and confidentiality goals of TrustZone TEE:

• **Possible data leak and tampering.** During execution of forwarded requests, the data passes through the OS. For devices without end-to-end encryption, this leads to possible data leak and tampering: a malicious OS (including its drivers) may peek into the data content, tamper with it, and feed the trustlets with tampered data.

• **External IO as a side channel.** As forwarding must send the arguments of a request in the clear (e.g. file offsets, read/write sizes), they can be used to infer the input data of the trustlet, creating a side channel. For instance, by inferring the accessed file offsets of a SQLite query, the attacker learns the query content [15].

1.3 Dissertation overview

1.3.1 Dissertation statement

Towards an efficient and secure edge, our overarching approach is to *co-design systems software with hardware and workloads*. The approach instantiates two key visions:

• **First, specialize for beaten path.** As described earlier, the workloads at edge are often driven by missions and executed repetitively; furthermore, its small set of users induces less dynamism to the device. For instance, a smart camera periodically captures image frames and a smart speaker keeps listening to activation keywords so it may execute speech recognition tasks in a tight loop. As such, there exist *beaten paths* among the workloads at the edge device. They are recurring, stable, and lean code paths, i.e. having similar call graphs and function dependencies, and are only a tiny fraction of their original codebase. One key insight is to specialize systems software for such beaten paths. Doing so has two key benefits: 1) it allows the system to co-design software with hardware at a finer granularity, thus improving efficiency; 2) it enables code reuse at a low cost, i.e. it only needs to reason about a small fraction of code. The benefits are crucial for an edge device, which is resource-constrained and craves for better efficiency.

• **Second, co-schedule hardware resources with app in the loop.** Unlike hardware on the cloud which is primarily designed for high throughput, hardware resources at the edge are more diverse and bring forth new tradeoffs, sometime unorthodox ones, which are often dependent on workloads and apps. For instance, a micro-controller like peripheral core is two orders of magnitude more efficient than a CPU when idling but runs one order of magnitude slower [16]; placing arbitrary app/workloads on a peripheral core only incurs efficiency loss instead of gain. To fully harness them towards our goals of efficiency and security, our key insight is to let systems software orchestrate hardware with app in the loop. We hence enable two-way interactions. First, the system provides APIs to apps for exposing necessary hardware details (e.g. power) so that apps may adapt accordingly. Second, the app supplies adaption decisions/workload characteristics (e.g. which parts of a file will be needed first), so that the system may prioritize and optimize for them. With such knowledge and interactions, the systems software works closer to hardware by selectively placing workloads on it, e.g. it decides *what* to place for efficiency and *what not* for security.

1.3.2 Contributions

This dissertation presents five novel systems which work in harmony under the aforementioned principle to address the above two key challenges.

This section overviews them by answering three key questions for each system:

- Why is the system important to the dissertation & what problem(s) it has addressed?
- What are the key innovations & results?
- What are the contributions to the field?

① **Power Sandbox** [17] is an OS principal to allow co-running apps to observe their individual power consumption at fine-granularity. It enables power awareness for applications, facilitating them to adapt to runtime environment towards better energy efficiency.

• **Why important?** Power awareness is important towards a more efficient edge platform; by knowing its own power consumption, an app can dynamically adjust its runtime behavior (e.g. throttle data transmission bandwidth, vary streaming fidelity), thus achieving better energy efficiency. However, in the existing approach an OS often meters system power and divides it among apps. Since the impacts of concurrent apps on system power are entangled, this approach not only makes it difficult to reason about power and but also results in power side channels, a serious vulnerability.

• **Key innovations.** A Power Sandbox (psbox) enables one app to observe the fine-grained power consumption of itself running in its vertical slice of the hardware/software stack. The observed power is insulated from the impacts of other apps. We support the psbox through two new kernel extensions: resource ballooning and power state virtualization. The former grants a psbox exclusive use of resource partitions at fine spatial or temporal granularities; it meters and reports the power of resource balloons. The latter virtualizes hardware power states for every single psbox upon exit and re-entry, eliminating power side channel.

• **Key results.** We implement psbox for a variety of major hardware components, including CPU, GPU, DSP, and WiFi interface on two embedded platforms. Psbox keeps an app's power observations highly consistent even when the app co-runs with other different apps. Across these runs, the app's energy, as observed by the app itself, differs by less than 5%; by contrast, energy shares reported by a prior approach differ by up to 60%.

• **What matters to the field?** Power sandbox is the first work that recognizes the power entanglement problem, which makes prior energy accounting approaches inadequate; it is the first kernel mechanism to address the problem and empowers apps with their power consumption.

② **Transkernel** [16] is an OS structure which offloads the unmodified kernel driver suspend/resume workloads of the CPU onto a peripheral core via dynamic binary translation. By doing so, a transkernel eliminates tedious and energy inefficient idling on CPUs, thus improving the energy efficiency of an edge platform.

• **Why important?** A major energy drain of smart devices is ephemeral tasks driven by background activities. To execute such a task, the OS kernel wakes up the platform beforehand and puts it back to sleep afterward. Such kernel executions are inefficient as they mismatch typical CPU hardware. They are better off running on a low-power, microcontroller-like core, i.e., peripheral core, relieving CPU from the inefficiency.

• **Key innovations.** The Transkernel OS structure has a set of novel designs. It translates stateful kernel execution through cross-ISA, dynamic binary translation (DBT); it emulates a small set of stateless kernel services behind a narrow, stable binary interface; it specializes for hot paths; it exploits ISA similarities for lowering DBT cost.

• **Key results.** In 10K SLoC, we implemented a transkernel prototype called ARK atop an ARM SoC. ARK offers complete support for device suspend/resume in Linux, capable of executing diverse drivers that implement rich functionalities (e.g., DMA and firmware loading) and invoke sophisticated kernel services (e.g., scheduling and IRQ handling). As compared to native kernel execution, transkernel only incurs 2.7× overhead, 5.2× lower than a baseline of off-the-shelf DBT. It reduces system energy by 34% under real-world usage.

• **What matters to the field?** The transkernel represents a new OS design point for harnessing heterogeneous SoCs. It contributes a key insight: while cross-ISA DBT is typically used under the assumption of efficiency loss, it can enable efficiency gain, even on off-the-shelf hardware.

③ **STI** [18] is an NLP inference engine which unifies model loading and execution via an elastic pipeline. With STI, an edge platform executes inference requests in short delays with minimum memory consumption, and is more efficient.

• **Why important?** Natural Language Processing (NLP) inference is seeing increasing adoption by mobile applications, where on-device inference is desirable for crucially preserving user data privacy and avoiding network roundtrips. Yet, the unprecedented size of an NLP model stresses both latency and memory, creating a tension between the two key resources of a mobile device. To meet a target latency, holding the whole model in memory launches execution as soon as possible but increases one app's memory footprints by several times, limiting its benefits to only a few inference before being recycled by mobile memory management and resulting in poor memory efficiency. On the other hand, loading the model from storage on demand incurs a few seconds long disk IO, far exceeding the delay range satisfying to a user; pipelining layerwise model loading and execution does not hide IO either, due to the large skewness between IO and computation delays.

• **Key innovations.** STI contributes two novel techniques. First, model sharding. STI manages model parameters as independently tunable *shards*, and profiles their importance to accuracy. Second, elastic pipeline planning with a preload buffer. STI instantiates an IO/compute pipeline and uses a small buffer for preload shards to bootstrap execution without stalling in early stages; it judiciously selects, tunes, and assembles shards per their importance for resource-elastic execution, which maximizes inference accuracy.

• **Key results.** Atop two commodity SoCs, we build STI in 1K SLOC and evaluate it against a wide range of NLP tasks, under a practical range of target latencies, and on both CPU and GPU. We demonstrate that, STI delivers high accuracies with 1–2 orders of magnitude lower memory, outperforming competitive baselines.

• **What matters to the field?** STI is the first inference engine that co-designs model loading and execution. Built on the key idea of maximizing IO/compute resource utilization on the most important parts of a model, STI reconciles the latency/memory tension of an edge platform.

④ **Driverlet** [19] is a driver model for deriving device drivers (e.g. MMC, USB storage, and CSI camera) for TrustZone from mature commodity Linux kernels via record and replay. By fulling confining device access inside TrustZone, driverlets ensures the data integrity and confidentiality, improving the security of an edge platform.

• **Why important?** Designed for IO-rich client devices, TrustZone features secure IO, allowing TrustZone apps (trustlets) to access IO devices without being known or tampered by the OS. However, Secure IO remains largely untapped today due to the difficulty in implementing device drivers for TrustZone.

• **Key innovations.** The Driverlet contributes a key insight: instead of reusing drivers *code*, we may reuse driver/device *interactions* from mature Linux driver via record and replay. Ahead of time, developers exercise a full driver and record the driver/device interactions; the processed recordings, called interaction templates, are replayed in the TEE at run time to access IO devices. The interaction template ensures faithful reproduction of recorded IO jobs (albeit on new IO data); it accepts dynamic input values; it tolerates nondeterministic device behaviors

• **Key results.** We build driverlets for three devices MMC, USB Mass, and a CSI Camera on RaspberryPi 3. We show driverlets are easy to build and have sufficient performance to trustlets: it only takes a few days to build a driverlet while porting/implementing a driver takes at least months; with driverlets, trustlets can execute over 100 SQLite queries per second (1.4× slower than a full-fledged native driver) and capture images at 2.1 FPS from a CSI camera (2.7× slower).

• **What matters to the field?** Driverlets fix the key missing link for secure IO, and for the first time open a door for trustlets to access complex yet essential devices.

⑤ **Enigma** [20] is a deception-based defense in TrustZone, which injects sybil file activities as the cover of actual file activities. By doing so, it mitigates the file activity side channels which can be used to infer app secrets of TrustZone, protecting the privacy of an edge platform.

Why important? As TrustZone TEE lacks filesystem implementations, the in-TEE apps (trustlets) often invoke external filesystems of a normal world OS for file services (e.g. persist the collected sensor data, access user credentials). While filedata can be encrypted, the file activities, including file operation types (e.g. read/write), sizes/offsets, and access occurrence (e.g. "the trustlet just created a file") must be sent in the clear. From the received file activities, the OS can infer a truslet's secrets such as input data, compromising the trustlet's privacy.

Key innovations. Enigma contributes three new designs. (1) To make the deception credible, the TEE generates sybil calls by replaying file calls from the TEE code under protection. (2) To make sybil activities cheap, the TEE requests the OS to run K filesystem images simultaneously. Concealing the disk, the TEE backs only one image with the actual disk while backing other images by only storing their metadata. (3) To protect filesystem image identities, the TEE shuffles the images frequently, preventing the OS from observing any image for long.

Key results. We build Enigma on Raspberry Pi 3, which works with unmodified EXT4 and F2FS shipped with Linux. We show that Enigma can concurrently run as many as 50 filesystem images with 37% disk overhead (less than 1% of disk overhead per image). Compared to common obfuscation for hiding addresses in a flat space, Enigma hides file activities with richer semantics. Its cost is lower by one order of magnitude while achieving the same level of probabilistic security guarantees.

What matters to the field? Enigma is the first system to guard the external IO of TrustZone. Its deception approach opens the door for a TEE to external untrusted OS services.

1.3.3 Dissertation organization

Guided by the two main challenges (§ 1.2), this dissertation is organized in two parts. The first part – *Harnessing hardware heterogeneity for efficiency* presents three works, Power Sandbox (Chapter 2), Transkernel (Chapter 3), and STI (Chapter 4), which re-defines power awareness for apps, improves kernel and user app efficiency on an edge platform, respectively. The second part – *Fostering security and privacy* presents Driverlet (Chapter 5), Enigma (Chapter 6) which are crucially tied to TrustZone as well security/privacy of an edge platform. Finally, Chapter 7 concludes the dissertation and discusses design hints to future edge systems.

Part I

Harnessing hardware heterogeneity for efficiency

Chapter 2

Power Sandbox: Power Awareness Redefined

2.1 Introduction

The quest for app power awareness[1] has lasted over a decade [11]: an app, as one or a group of user processes, demands to observe its power consumption online, in order to adapt its behaviors accordingly to lower power or higher efficiency. Traditionally, an operating system (OS) supports app power awareness through a two-step approach at run time as shown in Figure 2.1(a). First, the OS meters system power by either consulting a power model [21, 22, 23, 24, 25, 26, 27, 28] or performing *in situ* direct measurement [11, 29, 30, 31, 32]. Second, it divides the metered power into per-app shares, based on certain heuristics chosen at the OS development time.

Despite recent advances in fine-grained power metering [32, 31], the above approach suffers from two key inadequacies.

(1) **Reasoning difficulty**: it fails to provide power observations that are easy for apps to reason about and act upon.

(2) **Security vulnerability**: it creates power side channels [33], allowing attackers to learn a victim app's security-sensitive behaviors.

The latter inadequacy is already shown by prior work [34, 35, 36] and will be further demonstrated in this paper (§2.2.5): from its observed GPU power, an attacker app can infer what website a co-running

[1]Power awareness and energy awareness are often used interchangeably in prior work. To highlight power knowledge at fine temporal granularity, we use power awareness in this paper unless stated otherwise.

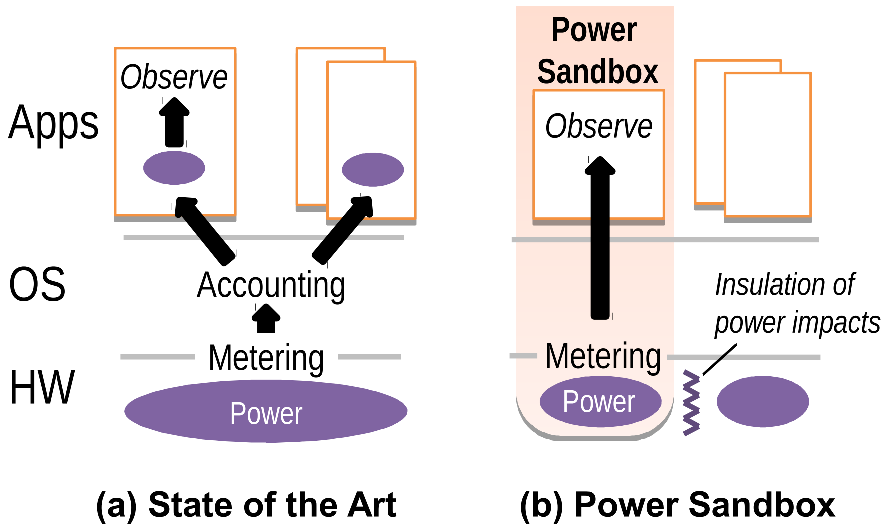


Figure 2.1: An overview of `psbox`.

victim browser is visiting. Such inference's success rate is 6× higher than random guess. Fine-grained power metering further *exacerbates* this vulnerability.

In summary, the two inadequacies are becoming the major obstacles towards app power awareness.

Our key observation is that the metered system power contains *entangled* impacts from concurrent apps, and the impacts cannot be separated cleanly. Such power entanglement is rooted in work-conserving OSs that aggressively multiplex apps on hardware resources. Unfortunately, the existing approach to app power awareness copes with power entanglement *reactively* at best without attempting to eliminate it.

To this end, we advocate a fresh perspective on OS support for power awareness, as illustrated in Figure 2.1(b). First, the OS supports any app to observe the power of the app running in its *vertical environment* (i.e., its vertical slice of the software/hardware stack) and hence prompts the app to suit the vertical environment. Furthermore, the OS insulates the app's power observation from the impacts of other apps.

Following this perspective, we propose a new abstraction called power sandbox, or `psbox` for short. A `psbox` allows the enclosed app to observe the collective power of the app itself and its vertical environment at fine temporal granularities. In this observation, the only possible contributions of concurrent apps are periods of idle power. The OS enforces `psbox` as the only way for apps to observe power: one app may enter or leave its `psbox` freely, but is only allowed to observe power when it is in the `psbox`. Free of power entanglement, the resultant power observation is not only amenable to reasoning but also minimizing power side channels. We stress that power sandbox *insulates* app power impacts but does not *isolate* their executions: all apps, inside a `psbox` or not, share the same system image as usual.

To support `psbox`, we have addressed two primary challenges:

Enforcing psbox boundaries We make the OS kernel respect psbox boundaries in resource multiplexing. The kernel does so with two extensions: i) it grants a psbox exclusive use of resource partitions at fine spatial or temporal granularities, called *resource balloons*; it meters and reports the power of resource balloons; ii) it virtualizes hardware power states for every single psbox.

Confining performance loss to sandboxed apps As other mechanisms for online resource monitoring [37, 38], psbox comes with runtime costs, which is mainly due to lost sharing opportunities. In response, a core mechanism of psbox is to confine the costs to the sandboxed apps and therefore ensures performance fairness among all the apps, regardless of their usage of psbox. This mechanism is both powerful and critical: assuming two apps co-running on a multicore equally share the CPU time and one app enters its psbox, the unsandboxed app continues to enjoy its original share normally – half of the total CPU time, despite the reduction in combined CPU utilization.

The OS kernel confines performance loss with two techniques. It tracks the lost sharing opportunities and fully charges the loss to the sandboxed app, disadvantaging this app in future resource competitions. It encapsulates resource balloons as normal scheduling entities and therefore reuses most of the existing kernel infrastructure for scheduling.

We intend psbox to be a "pay-as-you-go" service for apps: apps use psbox to periodically sample power or to selectively monitor power of key execution phases. Based on their power observation, apps make power-aware decisions, which remain valid even after they leave psbox. In most of their lifetime, they run outside of psbox without overhead.

Atop a recent Linux kernel and two embedded platforms, we implement psbox for a variety of major hardware components, including CPU, GPU, DSP, and WiFi interface. psbox keeps an app's power observations highly consistent even when the app co-runs with other different apps. Across these runs, the app's energy, as observed by the app itself, differs by less than 5%; by contrast, energy shares reported by a prior approach differ by up to 60%. In a benchmark of three co-running computer vision apps, the use of psbox by one app leads to 10% total throughput loss. The confinement of performance loss is robust: in a test with extremely high resource contention, despite the throughput of the sandboxed app dropping by 4×, the other co-running app only experiences 1% throughput loss.

Based on psbox, we present an end-to-end use case. We build a virtual reality app (in 2K SLoC) that periodically observes its power and dynamically trades its fidelity level for lower power, demonstrating how psbox facilitates the construction of power-aware apps.

This paper has made the following contributions:

- We present an analysis of existing approaches to app power awareness, demonstrate the inadequacies, and identify the cause as power entanglement. In response, we present a novel OS principal called power sandbox (`psbox`) that supports an app to observe the power of itself and its vertical environment.

- We enforce `psbox` and confine its performance cost to the sandboxed app. We do so through a suite of techniques: resource ballooning, power state virtualization, and tracking/charging the lost sharing opportunities.

- On top of a recent Linux kernel, we implement `psbox` for CPU, GPU, DSP, and WiFi interface. Our evaluation shows that `psbox` reliably insulates power impacts, incurs minor cost, maintains fairness, and facilitates the construction of power-aware apps.

The full source code of `psbox` is available at:

`http://xsel.rocks/p/psbox`

2.2 A Case for A New OS Principal

We next analyze the design space of supporting app power awareness. First, we distill the essential power knowledge needed by apps (§2.2.1). Next, we examine the classic two-step approach, showing that while metering is becoming accurate and efficient (§2.2.2), accounting encounters a fundamental difficulty which we dub power entanglement (§2.2.3 – §2.2.5). To address the difficulty, we advocate eliminating power entanglement and empowering apps to observe exactly what they need to know. This motivates a new OS principal (§2.2.6).

2.2.1 Power awareness: what matters to apps?

We first examine what power knowledge has been required by existing app adaptation strategies. Figure 2.2 illustrates the key concepts of app power-aware adaptation.

App cares about its own power impact By design, most adaptation strategies focus on optimizing one app's behaviors. By exploiting the app's domain knowledge, these strategies reduce the app's power impact, which will be translated to a similar reduction in the system power or energy. Often, apps demand to know their power impacts at fine temporal granularities in order to map the power to short-lived software activities [39, 32, 27, 25].

This app-centered approach is extensively taken by prior work: an app optimizes its own code execution efficiency [32, 39, 40, 41, 42], reduces the power impact of its own I/O activities [43, 44, 45], or does both simultaneously [46, 47].

App adapts to suit its vertical environment As illustrated in Figure 2.2, a vertical environment incorporates hardware conditions, system software configurations, user preferences, etc.

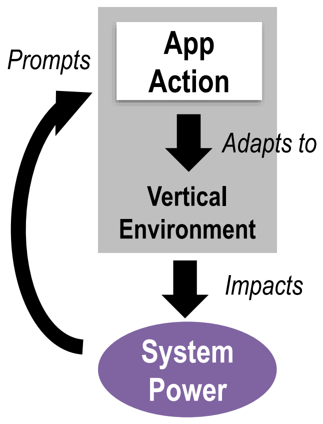


Figure 2.2: Concepts in app power-aware adaptation.

For higher power efficiency, prior systems adapt to various factors of a vertical environment. Code generators adapt to CPU microarchitectures [39]. Mobile/cloud offloading [40, 41] and mobile data compression [46] adapt to the comparative efficiency of CPU and wireless link. Content fidelity [11, 48, 49] and algorithm accuracy [50] adapt to user preferences. Network transfer scheduler adapts to network conditions [43, 44, 45] or app preference [51]. Web page or game rendering adapts to user perception [42, 52]. As such, power-aware adaptation decisions inherently depend on the app's vertical environment.

By contrast, app-centered adaptation rarely considers "horizontal" factors such as peer app activities, which would require apps to have not only deep knowledge of each other but also mutual trust. As a result, power-saving opportunities from horizontal cooperation (e.g., app co-execution [53], cooperative I/O [54], request piggybacking [55]) are more limited, and are often exploited at the OS level. These are complementary to the app-centered adaptation under discussion.

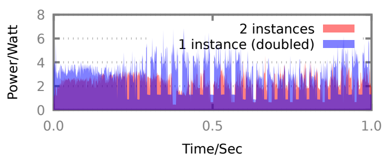
Comparative power drives actions To make an adaptation decision, an app often chooses one action out of multiple candidates by comparing their power impacts. In existing power-aware systems, these alternative actions include program partitioning plans [28, 40, 41], code generation strategies [39], middleware configurations [49], graphics rendering strategies [52], network transfer plans [43, 44, 45], hardware component combinations [47], and compression algorithms [46].

Summary: essential power knowledge We summarize the power knowledge that is essential to app adaptation as follows:

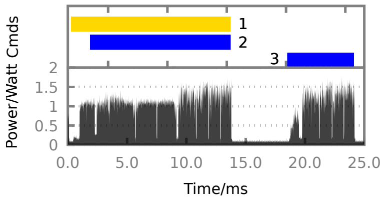
1. An app demands to observe power consumption of itself and its vertical environment at fine temporal granularity. It is often indifferent to the power impacts of peer apps.

2. An app must be able to compare the above power observations quantitatively.

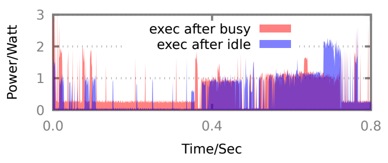
Unfortunately, this essential power knowledge mismatches what apps are learning from the current approach to power awareness. Next, we examine this approach, in particular its two key steps.



(a) Total CPU power of two co-running process instances, one on each core, compared to 2× power of one instance running alone. Hardware: 2×core Cortex-A15



(b) A sequence of three GPU commands (top) and the total GPU power (bottom). Commands of the same type have the same color. Hardware: PowerVR SGX544MP



(c) Comparison of CPU power of the same app when it runs after a CPU idle period and when it runs after a busy period.

Figure 2.3: Examples of power entanglement.

2.2.2 Fine-grained power metering is getting easier

System-level power metering[2] used to be the major challenge towards power awareness. While most prior work metered power using models [21, 22, 23, 24, 25, 26, 27, 28], such modeling for modern hardware is increasingly difficult, due to processor heterogeneity, variation in fabrication [56], and changing operating conditions [57].

Fortunately, direct measurement, the alternative metering method, starts to show high promise. Besides the known benefits of high rate (>10KHz) and accuracy (in mW) [57], recent work demonstrates that direct measurement can be efficient and therefore *in situ*, by offloading periodic power sampling and pre-processing to low-power microcontrollers [31, 32]. Fine-grained, inexpensive power metering enables characterization of short-lived software activities, and is likely to become a common feature of future hardware platforms. We will discuss this in detail in §2.8.1.

2.2.3 Accounting is hard due to power entanglement

Even though system power can be metered at a high resolution, attributing it to separate apps encounters a fundamental difficulty:

Power entanglement: In a work-conserving OS that aggressively multiplexes apps on hardware, concurrent apps impact the hardware power simultaneously, and the impacts become inseparable.

We identify three major causes for power entanglement:

- *Spatial concurrency in hardware* Multiple apps concurrently use disjoint hardware resources for which power can only be metered as a whole. Note that such power metering scopes are often hardware design choices. We show this with a simple experiment in Figure 2.3(a). On a dual-core CPU with one power rail, we measure the whole CPU power, and compare i) only running one process on core 0 to

[2]In this paper, we use "metering" to refer to both physically measuring energy and inferring energy through software models.

ii) additionally running a second instance of the same process on core 1. As shown in the figure, one cannot simply extrapolate the former run's power, e.g., by doubling it, to get that of the latter run. This is because in the latter run, the power impacts of two active CPU cores are entangled, as has also been confirmed by prior work [53].

- *Blurry request boundary* Many hardware components, notably accelerators and I/O, accept requests from CPU and execute the requests asynchronously. Since CPU lacks visibility into the execution durations of in-flight requests, it cannot differentiate their power impacts. In Figure 2.3(b), we show the durations of three consequent GPU commands and the GPU power. The commands' durations are to the best of CPU's knowledge. Each duration starts when the command leaves the OS and enters the GPU, and ends when the OS is notified command completion by a GPU interrupt. Although we expect that the power of command 2 is similar to that of command 3 (they are of the same type), command 2 significantly overlaps with command 1 in time and their power impacts are hence entangled. The OS is incapable of separating power of these two commands.

- *Lingering power state* Software workloads may prompt changes in the hardware power state, which will affect the power of subsequent workloads. In Figure 2.3(c), we compare the CPU power when one app runs in two different scenarios: running after the CPU has been idle for a while; running right after the completion of another busy workload. The latter scenario incurs noticeably different power, as the CPU clock rate raises prior to the app execution. Similar effects exist in transmission power of wireless interfaces.

Power entanglement exists no matter *how* power is metered, either through modeling or direct measurement. In particular, modeling suffers from all the causes above, as most existing modeling techniques infer system-level power from *aggregated* hardware activities, e.g. total LLC misses read from performance counters [27]. High-rate direct measurement does not help either, since the above causes prevent obtained power samples from being attributed to apps, as we will demonstrate in evaluation (§2.6).

Existing approaches are inadequate Existing accounting mechanisms cope with power entanglement *reactively* at best. They divide system power among apps using a variety of heuristics: even splitting [58], attributing each app's marginal contribution [23], attributing based on app hardware utilization [59], or attributing to the app that uses the hardware most recently [25].

These heuristics are useful for *system-level* energy accounting, in that they encapsulate the beliefs or policies of the OS designers. However, they are unable to address the aforementioned major obstacles in *per-app* power awareness, since no accounting heuristics can eliminate power entanglement that has *already* occurred.

2.2.4 Power entanglement creates reasoning difficulty

Existing accounting mechanisms provide per-app power shares that are difficult for apps to reason about or reproduce. For instance, merely based on its power share, one app can hardly tell why one network transmission consumes more energy than others of the same length (which could be because the OS charged the WiFi tail energy to this particular transmission [25, 26]); or why multiple invocations of the same function show much different power behaviors (which could be because varying workloads ran concurrently on other CPU cores).

One may suggest that besides dividing the system power, OS should open up its accounting internals to apps, e.g., publishing the hardware usage of concurrent apps and the accounting heuristics used by the OS. This will create more problems. i) Besides reasoning about power, app developers now need to reason about power accounting heuristics. As the heuristics become non-trivial (e.g. based on cooperative game theory [23]), app development soon becomes a daunting task. ii) Revealing apps' hardware usage to each other may create security vulnerabilities.

2.2.5 Power entanglement creates security vulnerability

Dividing system power among apps may reveal their power behaviors to each other. When the apps are mutually distrusted, this creates a known vulnerability called power side channels [33]: by observing the power of a victim app, an attacker app may learn the victim's security-sensitive behaviors, such as encryption and authentication procedures [60, 61, 35], GPS usage [34], or GUI state [35].

We next demonstrate that power entanglement can be exploited through power side channels, showing GPU power leaks a browser's deep information – which website it is visiting. We co-run two apps: a browser (victim) is scripted to open the Alexa top10 websites; an attacker app, while executing light GPU workloads as camouflage, attempts to infer what website the browser is opening. We train the attacker *once* with the GPU power traces collected when the browser runs alone, labeled by website URLs. In subsequent runs, the attacker infers the websites based the similarity between its known and observed GPU power activities. The similarity is measured with DTW, a well-known algorithm for time-series analysis [62].

Our results show that the attacker's success rate of inference is 60%, 6× higher than random guess. This is because different web pages tend to generate different GPU workloads, and hence unique power signatures.

2.2.6 Design choices

We advocate an OS principal for any power-aware app to observe the collective power of the app itself and its vertical environment. Specifically, the OS should achieve three objectives:

1. *Insulate app power observation* The OS shields an app's power observation from the impacts of other apps, and hence eliminates power entanglement for this observation. The OS does so by adjusting resource multiplexing.

2. *Preserve vertical environment* The OS keeps an app vertical environment unchanged, whether the app is using the OS service for observing power or not. This enables apps to make valid adaptation decisions based on their insulated power observations.

3. *Track and charge cost* The OS charges any overhead or lost multiplexing opportunity in insulating power observations to the requesting app. This ensures fairness among all apps despite their different usages of the service.

Following these choices, we introduce a new OS principal called *power sandbox*, or `psbox`, as will be presented below.

2.3 System Overview

`psbox` is an OS principal enclosing one power-aware app, i.e., one or a group of user processes. It is the only way for any app to observe power. More specifically, a `psbox` exposes an interface of *virtual power meter* to the enclosed app, from which the app may read real-time power consumption incurred by the app and its vertical environment. In this observed power, the only possible contributions of concurrent apps are periods of idle power.

```
// Create a power sandbox
box=psbox_create(HW_CPU /* optional */);
psbox_enter(box);
// Continuous collection of power samples
psbox_sample(box, &buf, NUM_SAMPLES);
// One-time query of energy
energy = psbox_read(box);
psbox_leave(box);
```

Listing 2.1: The `psbox` User API

Intended usage of `psbox` Since a `psbox`'s overhead is charged to the sandboxed app, we expect apps to use `psbox` as a "pay as you go" service. They use `psbox` to *periodically* sample power, or *selectively* monitor power during interesting execution phases, and leave `psbox` for full-speed execution. An app makes power-aware decisions according to its `psbox`'s virtual power meter. After the app leaves the `psbox`, its decisions remain valid, since the OS preserves the app's vertical environment (§2.2.1). The app only pays the price of `psbox` during a small fraction of its execution time.

We would like to stress this "pay as you go" power sandboxing is complementary to, and may coexist with, the OS mechanisms that optimize multiplexing of *power-unaware* apps for combined efficiency [53, 63, 64].

The app interface Apps access `psbox` through the API summarized in Listing 2.1. An app creates a `psbox` and binds it to a set of hardware components of which power is reported (line 2). The granularity of hardware sets is determined by the possible power metering scopes as supported by hardware. For example, the hardware can be a subset of CPU cores sharing one measurable power rail [65]. During execution, the app is at liberty to enter or leave the `psbox` (line 3 and 8).

When it is in `psbox`, the app may query the `psbox`'s virtual power meter. Similar to accessing CPU performance events [66, 67], the app may collect power samples in a user-provided buffer (line 5) or poll to get the accumulated energy (line 7). Unlike existing CPU events (including the power events [68]), all `psbox` power readings are timestamped. These timestamps come from a standard clock that apps can access through the `clock_gettime()` syscall. This allows apps to readily map power readings to software activities at fine granularities. Depending on metering methods, the timestamp resolution can be as high as 10 μs, as will described Section 2.5.

Kernel enforces `psbox` boundaries The kernel eliminates power entanglement for a `psbox`. To do so, the kernel allocates spatial and temporal partitions of hardware resources at fine granularities, and grants *exclusive* use of them to the `psbox`. We term these partitions resource balloons, which are exemplified by a set of CPU cores and a time slice of the WiFi interface. Having established the boundaries for resource balloons, the kernel meters the corresponding hardware power, through either direct measurement or modeling (§2.2). The kernel then reveals the metered hardware power to the `psbox`'s virtual power meter.

Kernel confines performance loss A `psbox` incurs performance overhead. Most notably, the exclusive use of resource balloons likely leads to hardware under-utilization. The kernel tackles the overhead in two ways. On one hand, the kernel reduces the overhead by keeping resource balloons small, as will be shown in Section 2.6. More importantly, the kernel confines the overhead to the sandboxed app and minimizes the impact on apps outside the `psbox`. To do so, the kernel tracks the *lost sharing opportunity* due to resource ballooning, bills it to the sandboxed app, and properly disadvantages the sandboxed app in future competitions for accessing the hardware.

2.4 Kernel Support

To support `psbox`, we face a twist of two challenges: i) eliminating power entanglement (§2.2.3) by changing how the kernel multiplexes concurrent apps on hardware; ii) integrating the changes into mature kernel mechanisms to avoid disruptive modifications. To address the first challenge, we present a model for extending kernel drivers; to address the second challenge, we describe how to apply the model to the kernel subsystems that manage major hardware components. For brevity, the remainder of this paper refers to these kernel subsystems as *drivers* in general.

2.4.1 The driver model

We propose two lightweight extensions to existing drivers.

Resource ballooning Resource multiplexing must respect `psbox` boundaries. More specifically, the kernel must confine spatial concurrency and asynchronous requests, two major causes of power entanglement (§2.2.3). To this end, we retrofit the concept of memory ballooning for virtual machines [69]. The kernel allocates fine-grained resource partitions, called *resource balloons*, and makes them exclusive to a `psbox`. The kernel schedules resources balloons together with other normal apps, enforces balloon boundaries, and meters the power of resource balloons for the `psbox`.

We next describe two types of balloons. In the discussion, we use `psbox`$\langle \boxed{\text{App}}, hw \rangle$ to denote a `psbox` bound to hardware hw and enclosing an app $\boxed{App}$. We use $\overline{App}$ to refer to all other apps outside the `psbox`.

- *Spatial balloon* is for confining spatial concurrency on OS-schedulable, preemptable resources, most notably CPU cores. It prevents $\boxed{App}$ and $\overline{App}$ from using hw simultaneously. To do so, when granting the access of hw to $\boxed{App}$, the OS schedules in a spatial balloon that occupies all the resources in hw, which effectively exclude $\overline{App}$ from hw.

- *Temporal balloon* is for confining request asynchrony on accelerators and I/O devices. It prevents $\boxed{App}$ and $\overline{App}$ from having in-flight requests submitted to hw simultaneously. To do so, when granting $\boxed{App}$ the access to hw, the OS schedules in a temporal balloon, a time slice during which the OS only dispatches the requests from $\boxed{App}$ to hw. At the start and end of the temporal balloon, the OS drains in-flight requests by holding back new requests until hw completes the existing ones.

A key advantage of resource balloons is they appear as normal scheduling entities to the existing kernel infrastructure. Hence, they keep most of the latter oblivious and therefore unmodified. i) The kernel's existing accounting mechanism does not differentiate the portion of hw used by $\boxed{App}$ from the portion intentionally kept idle by the balloons, e.g. unused CPU cores or stalled GPU cycles. The kernel simply bills all the resource occupied by the balloons to $\boxed{App}$. ii) The kernel's existing schedulers, e.g. for CPU or for network packets, still enjoy full freedom of choice: they are at liberty to decide whether and when to schedule a balloon on hw, and may freely multiplex $\overline{App}$ on hw without constraints.

Figure 2.7 in evaluation shows resource balloons in action.

Power state virtualization Enclosed in a `psbox`$\langle\boxed{\text{App}}, hw\rangle$, $\boxed{App}$ should neither observe any lingering power state (§2.2.3) on hw nor leave any residual state after using hw. To this end, the OS keeps a virtual copy of the power state of hw for each `psbox` (and a separate copy for $\overline{App}$). Upon scheduling in a resource balloon on hw, the OS puts hw in the power state in which the `psbox` left hw previously; when scheduling out the resource balloon, the OS extracts the hardware power state and saves it for the `psbox`.

To make this idea practical, we put hardware power states into two categories, depending on the costs of the related state transitions, and treat them differently:

- *Off/suspended states*, in which devices lose power or remain in deep sleep. Examples include CPU deep sleep that retains no cache content, or GPS cold start without any locked satellite. Exiting these power states often requires expensive hardware operations, e.g. device initialization. Once a device exists such an off/suspended state, it often remains in an operating state for a long period, as described below.
- *Operating/idle states* are rough equivalents of P and C states in ACPI [70], which control performance settings of a working device or power saving of an idle device. A device can switch among these states at low cost and with low delay (often sub-milliseconds). Examples include CPU frequencies and WiFi transmission power levels.

The kernel virtualizes *operating/idle* states and reports the corresponding hardware power to `psbox`es. By contrast, it neither virtualizes *off/suspended* states nor reveals the power pertaining to these states. The rationales are as follows. First, reconstructing off/suspended states for each `psbox` can be prohibitively expensive, e.g. it requires to cold restart a GPS device for each new `psbox`. Furthermore, it is unsafe to reveal unvirtualized off/suspend hardware states to apps, which would allow a malicious app to infer the device usage, e.g. whether other apps have just used GPS for localization, through power side channels (§2.2.5). Hence, for the durations when hw is off/suspended, the kernel simply feeds `psbox` with samples of hw's idle power. To $\boxed{App}$, hw appears idle.

2.4.2 Applying the driver model

According to our model, a driver takes on two new responsibilities for `psbox`:

1. Enforcing resource balloon boundaries, including virtualizing power states;

2. Tracking lost opportunities of resource sharing and counting them against $\boxed{App}$.

Beyond these two, balloon scheduling is handled by existing kernel mechanisms transparently.

Multicore CPU

We build spatial balloons into the CPU scheduler. A typical multicore CPU scheduler runs multiple instances, one for each core and managing a runqueue of local tasks (processes or threads). To choose the next running task, an instance picks the one with the best scheduling credit. Scheduling credits are often computed from tasks' recent CPU usage. For scalability, scheduler instances rarely communicate.

To enforce spatial balloons for $\texttt{psbox}\langle\boxed{\text{App}}, hw\rangle$, the CPU scheduler coschedules tasks of $\boxed{App}$ on all the cores of *hw*. If the runnable tasks in $\boxed{App}$ are fewer than the cores, the scheduler runs dummy tasks on the remaining cores to force them idle.

To do this, an existing multicore scheduler faces twofold challenges. First, it needs to decide when to start and end a coscheduling period across a set of cores. However, in current designs each scheduler instance schedules its local tasks independently. Second, according to CPU cycles spent in coscheduling, the scheduler needs to discount scheduling credits, and hence ensure fairness between $\boxed{App}$ and $\overline{App}$ across all the cores. However, in current designs an instance focuses on maintaining fairness among its local tasks.

While the idea of coscheduling is long known [71], the above challenges were still considered unaddressed on multicore, especially the fairness concern [72]. To address the challenges, we introduce a new notion of *scheduling loan* with three key ideas: i) we allow a scheduler instance to pick a task T for execution even if T does not have the best scheduling credit among all the local runnable tasks; ii) in order to be picked, T must get a loan to triumph other runnable tasks and pay back the loan with its future credits; iii) all tasks in $\boxed{App}$ share their scheduling loans.

Our augmented multicore scheduler works as follows.

- *Scheduling entities*: Similar to a Linux `cgroup`, a `psbox` has a set of scheduling entities $\{E\}$, one entity on each core. An entity E_i encompasses all tasks in $\boxed{App}$ on core i and keeps a collective scheduling credit. The kernel schedules E_i together with other normal tasks.

1. *Schedule in*: Same as in current designs, the scheduler instance on core i picks E_i when E_i has the best scheduling credit. The instance further picks a task within E_i to run.

2. *Task shootdown*: The scheduler instance thus requests all other cores to schedule in their corresponding entities in $\{E\}$. It does so by sending inter-processor interrupts to all other cores. Upon request, the scheduler on a remote core j picks E_j: it calculates Δ_j, the initial loan of E_j, as the difference between

E_j's current scheduling credit and that of the most favorable task on core j (which would otherwise run). After shootdown, all tasks in $\overline{\mathit{App}}$ are off CPU and a coscheduling period for $\boxed{\mathit{App}}$ starts.

3. *Running & loan update*: During coscheduling, scheduler instances bill local CPU cycles to the corresponding entities in $\{E\}$. When any scheduler instance, e.g., the one on core i, is invoked for rescheduling, it takes the chance to calculate the extra loan needed by E_i to warrant E_i's continue use of core i, and add this new loan to Δ_i.

4. *Schedule out*: The coscheduling of $\boxed{\mathit{App}}$ continues until none of $\{E\}$ has the best credit on their corresponding cores, i.e., they all need extra loans to continue. At that time, the scheduler simultaneously schedules out all E_i from all the cores, by performing another shootdown.

5. *Loan redistribution & repayment*: When scheduling E_i out, a current scheduler design will adjust E_i's credits based on the time E_i just runs. We further make $\boxed{\mathit{App}}$ pay back the loans that have accumulated during the preceding coscheduling period. To provide long-term fairness over all the cores, all entities in $\{E\}$ evenly split their total loans. The scheduler redistributes the loans within $\{E\}$, which will disadvantage $\boxed{\mathit{App}}$ in future scheduling.

Accelerators

Accelerators, such as GPU and DSP, execute commands offloaded from the CPU. The lowest CPU/accelerator interface is often a shared command queue. To exploit hardware parallelism, the command queue is asynchronous: CPU may dispatch multiple commands to the queue, and will be notified by the accelerator on the completion of these commands.

In multiplexing apps on an accelerator, the corresponding driver schedules their commands. The driver picks one app's pending commands for dispatch, based on the scheduling credits of all apps, e.g., their recent accelerator usages, and the driver's scheduling policy. To support `psbox`, we bake temporal balloons (§ 2.4.1) in the driver. We augment how the driver switches among commands of different apps and bills the accelerator usage; meanwhile, we keep any scheduling policy intact. In a nutshell, i) the augmented driver treats $\boxed{\mathit{App}}$ as a single app in scheduling; ii) the driver avoids dispatching commands of $\boxed{\mathit{App}}$ as long as any commands from $\overline{\mathit{App}}$ are outstanding; iii) the driver bills any resultant lost opportunity in utilizing the accelerator to $\boxed{\mathit{App}}$; iv) the driver further virtualizes the accelerator's operating frequency, its most important power state, for each `psbox`.

We next describe how the driver schedules in and out a temporal balloon.

1. *Drain others*: When the driver's scheduling policy decides to dispatch commands for $\boxed{App}$, the driver buffers all subsequent requests (from both $\boxed{App}$ and $\overline{App}$) until the accelerator hardware notifies the completion of all existing commands. During this phase, the driver bills the unutilized portion of the accelerator (e.g., idle DSP cores) to $\boxed{App}$ as if the portion was actually used by $\boxed{App}$.

2. *Flush psbox*: After draining outstanding commands, the driver sends out any buffered command for $\boxed{App}$, which may have accumulated during phase 1.

3. *Serve psbox*: The driver directly dispatches all the subsequent requests from $\boxed{App}$ to the accelerator while buffering the ones from $\overline{App}$.

4. *Drain psbox*: When the driver's scheduling policy decides that $\overline{App}$ deserves the access of accelerator, it drains any outstanding commands from $\boxed{App}$ in a way similar to phase 1. Over the course of phase 2–4, the driver bills the usage of entire accelerator to $\boxed{App}$.

5. *Flush others*: The driver sends out any buffered commands from $\overline{App}$, which may have accumulated in phase 4, in their queueing order. Thereafter, it buffers all subsequent commands from $\boxed{App}$ while dispatching ones from $\overline{App}$ directly.

The above design integrates well with existing schedulers, yet are not tied to any specific definition of fairness or scheduling policy. A challenge to demonstrating this, however, is that many production accelerator drivers use simple scheduling policies, e.g., round-robin dispatch, which do not guarantee fairness. In our implementation described in Section 2.5, we have built fair queueing schedulers as baseline designs for GPU and DSP on our test platform, and augment the schedulers for supporting `psbox`.

Wireless interfaces

Wireless network interfaces (NICs) such as WiFi interface, are asynchronous by nature. Often, apps trap into the kernel to deposit their packets into their corresponding kernel buffers; the driver incorporates a packet scheduler to dispatch these packets into a unified transmission queue, from which the driver will send packets to the NIC in order. The packet scheduler determines scheduling credits for apps based their total sent bytes; it ensures fairness through its queueing discipline, e.g., the Linux `fq_codel`.

We tap into the packet scheduler to realize temporal balloons for NICs. We realize packet draining phases similar to accelerators as described above, while holding back packets in per-socket buffers instead of a global queue. To better assess lost sharing opportunities, the packet scheduler inspects packets that are buffered due to temporal balloons. It identifies any buffered packets that *could have* been dispatched without the balloons. Based on the total bytes in these packets, the driver discounts the scheduling credit of $\boxed{App}$ as a penalty for the lost opportunities.

A particular challenge is making packet reception respect `psbox` boundaries (§2.4.1). To achieve this, the NIC should *defer* receiving the packets that are not destined to the current temporal balloon, a function unsupported by commodity wireless NICs. Because of this, our current implementation is limited in insulating power impacts of receiving different packets. Yet, we have observed that such reception deferral can be achieved by exploiting virtual MAC addresses, an emerging feature of recent WiFi NICs [73, 74, 75]: the driver creates one virtual MAC for each `psbox` and switches among virtual MACs as it switches among temporal balloons.

Wireless NICs often have non-trivial power state that must be virtualized. Fortunately, modern WiFi NICs [76] often expose the control of power states to the OS. Hence, we augment the WiFi NIC driver to virtualize power states including transmission modes and power saving timer, and drive an independent state machine for each `psbox`. We recognize that cellular (4G) NICs have uncontrollable power states [77] which we will discuss in Section 2.7.

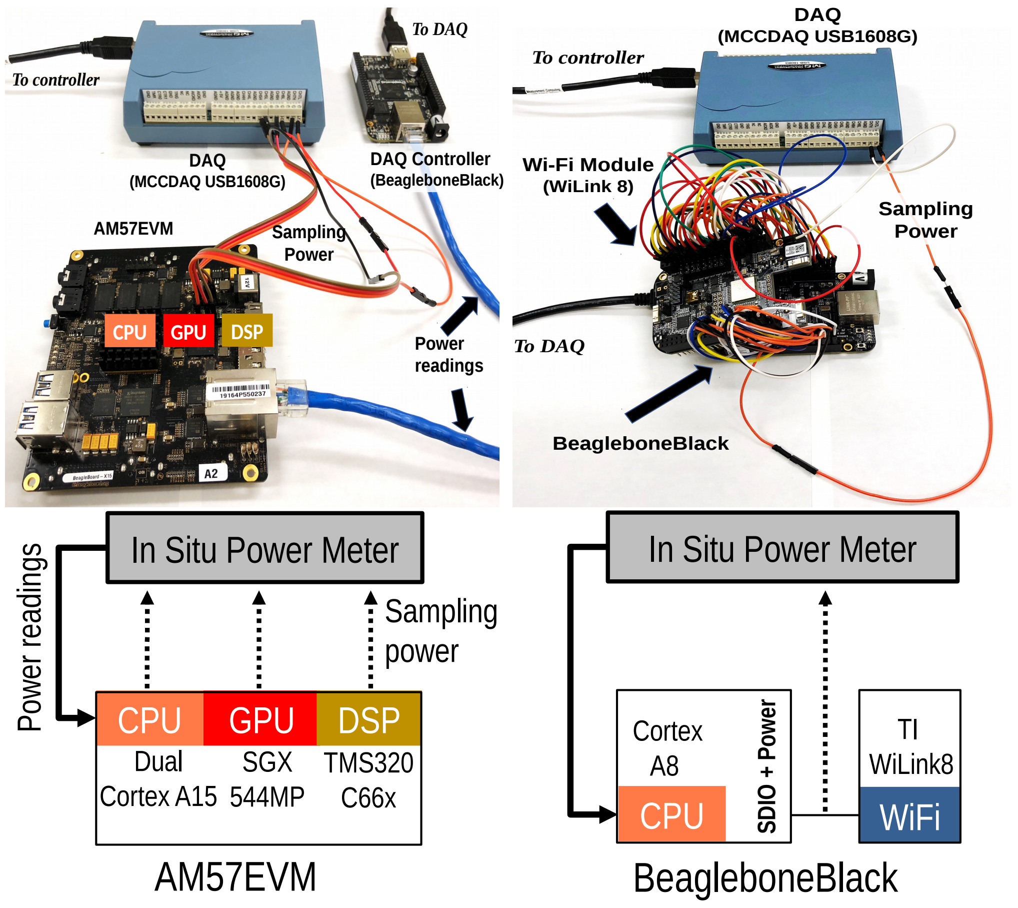
2.5 Implementation

We have built `psbox` into the Linux kernel 4.4 with about 2250 SLoC. We have assembled two hardware prototypes capable of measuring each of the major hardware components *in situ* and separately, as shown in Figure 2.4. The power sampling is as fast as 100KHz. Besides acquiring power samples, the power meter and the CPU synchronize their respective clocks to align power samples with software activities. It is worth noting that the purpose of our hardware prototypes is for evaluating `psbox`; they are not intended to be free-roaming devices as other systems [32, 31].

CPU We build `psbox` into the Linux completely fair scheduler (CFS) [79]. Although a CFS instance is able to schedule a process group (cgroup) as one scheduling entity, it does not coordinate multiple scheduler instances. We encapsulate each power sandbox in a Linux cgroup, and coordinate the tasks within through IPI.

GPU We implement `psbox` for PowerVR SGX544, a mobile GPU on the platform in Figure 2.4(a). Due to diversity of modern GPUs, we further evaluate `psbox` atop Qualcomm Adreno420 on Nexus 6. The two GPUs belong to different families, and have very different hardware/software stacks.

For both GPUs, we tap into their GPU command queues to implement fair schedulers in the spirit of the Linux CFS [79]: our scheduler tracks per-app virtual GPU runtime and favors GPU commands from the app that has the minimal virtual GPU runtime. Atop the schedulers, the drivers enforce temporal balloon



(a) Platform for testing psbox on CPU, GPU, and DSP

(b) Platform for testing psbox on WiFi interface

Figure 2.4: Prototype hardware platforms used in `psbox` evaluation. *In situ*, *per component* power metering (through four distinct power rails) is built atop a Cortex-A8 controlling MCCDAQ USB1608G [78] sampling at 100KHz. Time synchronization is over GPIO (not shown). In (b), the Beaglebone Black acts as both the target system and the DAQ controller.

boundaries differently, based on their existing structures: since SGX544 directly dispatches commands from syscall contexts to GPU, the driver buffers app *locking requests*; by contrast, since Adreno330 buffers GPU commands in per-process queues before dispatching them, the driver buffers *commands* from apps.

DSP We implement `psbox` for TI c66x, a popular multicore DSP that supports OpenCL. During execution, CPU dispatches DSP commands, e.g., task execution or cache flush, via a kernel-managed command queue. Similar to GPU, we enforce resource partitions atop a fair scheduler along the command queue. The driver further inspects DSP commands for tracking their dispatch and completion time.

WiFi We build `psbox` for the TI WiLink8 NIC with a `wl1837` chip as shown in Figure 2.4(b). The chip accepts packets and commands from CPU over SDIO, and runs its own firmware to implement MAC layer and below.

	Benchmark	Description
CPU	bodytrack	A vision program tracking human body move (P)
	calib3d	Camera calibration and 3D reconstruction (O)
	dedup	Compressing data stream with deduplication (P)
GPU	browser	A webkit browser opening a Google homepage (T)
	magic	Rendering a “magic lantern” scene at 60fps (V)
	cube	Rendering a rotating cube scene at 60fps (Q)
	triangle	A synthetic app drawing 100k triangles /sec offscreen
DSP	sgemm	Single-precision matrix-multiplication (T)
	dgemm	Double-precision matrix-multiplication (T)
	monte	Monte Carlo simulation. (T)
WiFi	browser	A Links browser opening a Yahoo homepage
	scp	Transmitting a 50MB data file over ssh
	wget	Transmitting a 50MB data file over http

Figure 2.5: Benchmark apps used in evaluation. P-PARSEC 3; O-OpenCV 3.1; T-TI am57 SDK; V-PowerVR SDK; Q-Qt SDK

We build temporal resource partitions into the Linux’s fair packet scheduler and virtualize the NIC power state in the driver. Despite the NIC’s support of multiple MACs, when we switch MAC at run time the NIC resets and loses its association with base station. Therefore, the lack of true MAC virtualization defeats our effort in insulating energy impacts of receiving different packets, as described in Section 2.4.2.

2.6 Evaluation

We evaluate the drivers augmented for `psbox` reported in Section 2.5 using benchmark apps summarized in Table 2.5. The evaluation answers the following questions:

§2.6.1 Does `psbox` eliminate power entanglement?

§2.6.2 How does `psbox` impact app performance?

§2.6.3 Does `psbox` confine throughput loss to sandboxed apps?

§2.6.4 Does `psbox` facilitate building power-aware apps?

2.6.1 Elimination of power entanglement

Methodology To test each driver, we run a set of scenarios as shown in Figure 2.6. Designating a benchmark app *App* to be power-aware, we first run *App* alone and then co-run it with other apps. For co-running scenarios, we compare `psbox` to an existing kernel-level accounting mechanism [26] without `psbox`. This prior mechanism derives *App*’s power by dividing each system power sample among co-running apps based on their hardware usages in each power sampling interval. Note that we implement this prior mechanism favorably by

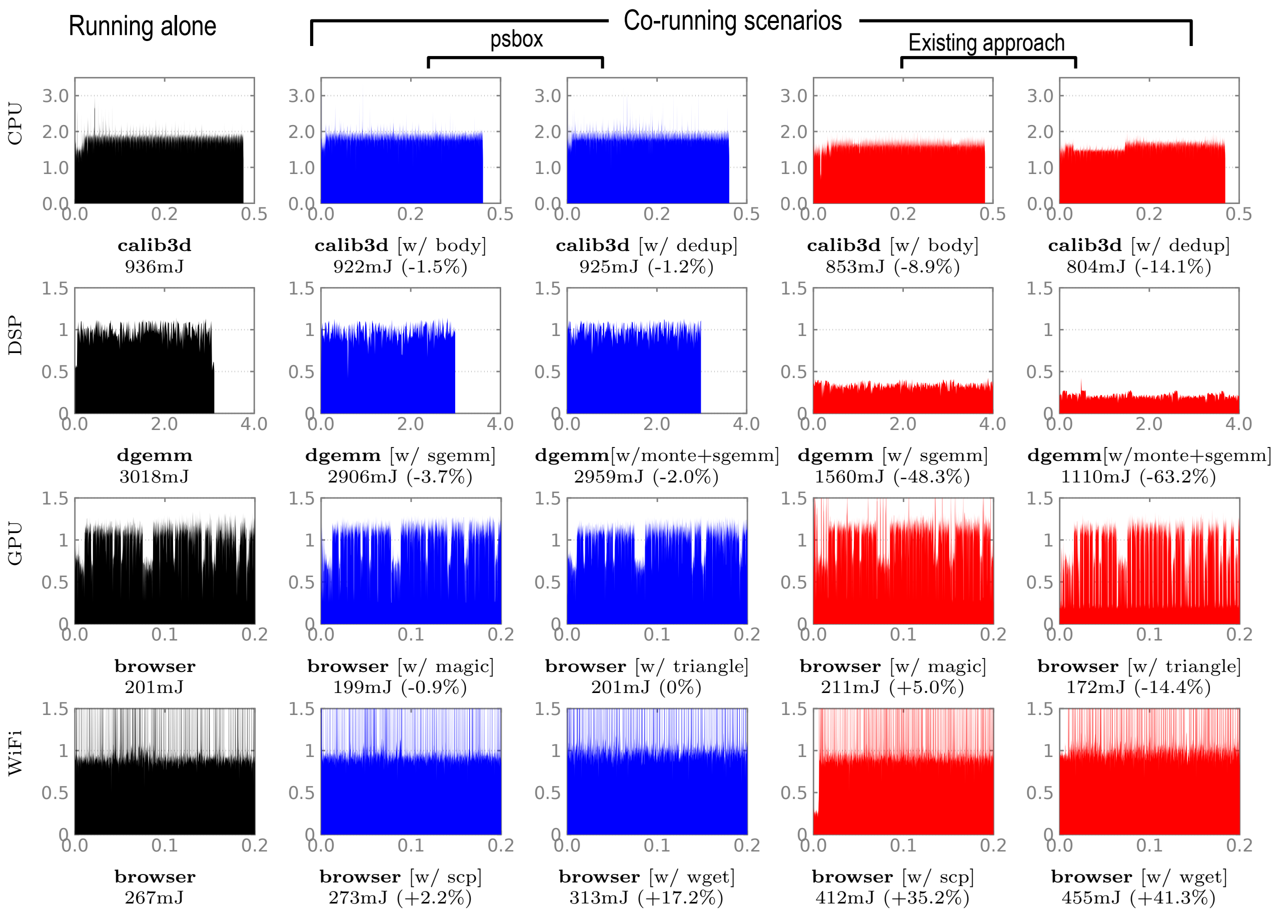
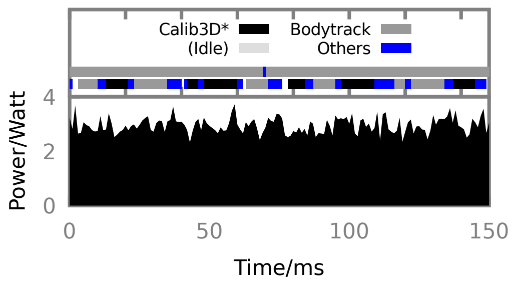


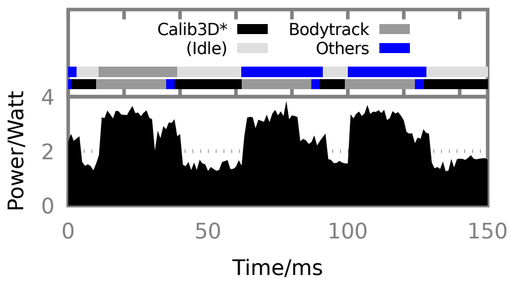
Figure 2.6: **Power of the benchmark scenarios**. In all plots, x-axis: Time/Sec; y-axis: Power/Watt. In each row: even as an app co-runs with different apps (column 2–5), `psbox` provides it with consistent power observations (column 2 & 3), which are close to the power of the app running alone (column 1). This contrasts to the power attributed to the app by an existing accounting approach (column 4 & 5). The numbers under each plot show the app's total energy and the difference compared to the energy when the app runs alone. Some plots cannot display full length of power activities due to space limit.

tracking hardware usage at the lowest software level and at very fine granularities (10μs, 10$\times$ smaller than prior work [32, 31]).

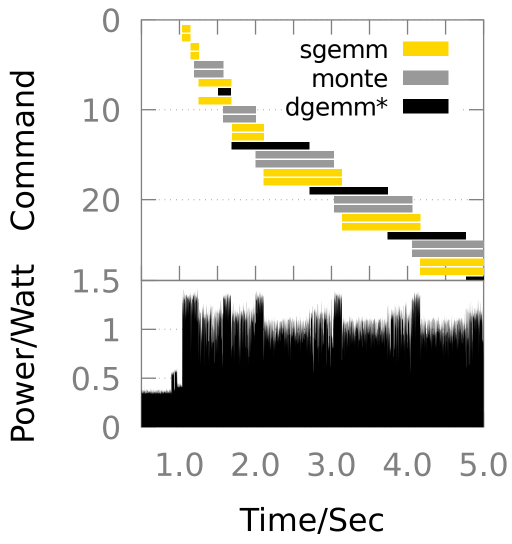
Our experiments demonstrate that `psbox` achieves its primary goal of eliminating power entanglement. As shown in the figure, no matter whether *App* is executed alone or co-executed with different apps, `psbox` keeps *App*'s power observations highly consistent, e.g., preserving significant power spikes and dips. By contrast, the power shares produced by the prior mechanism exhibit significant variations. The power differences are reflected in that of the accumulated energy: while the energy values reported by `psbox` are less than 5% within each other in most scenario sets, that of the prior approach can be as high as 60%. This also supports our argument in Section 2.2.3: existing accounting approaches are fundamentally inadequate, despite of the high metering rate. Note that `psbox` does not seek absolute *reproducibility* of power observations, which



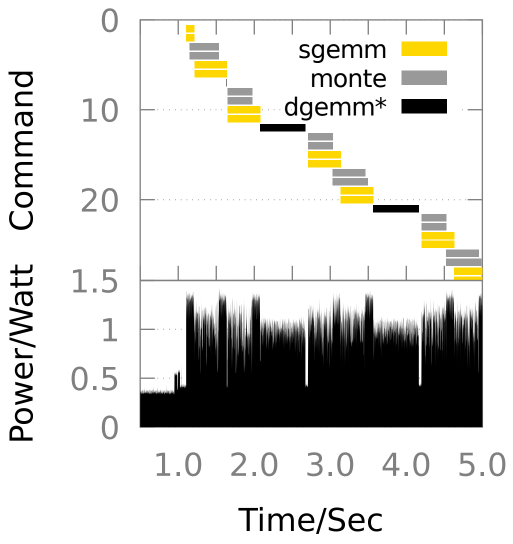
(a) Dual-core CPU w/o `psbox`



(b) Dual-core CPU w/ `psbox` and spatial balloons for calib3d*



(c) DSP w/o `psbox`. Commands overlap in time freely.



(d) DSP w/ `psbox` and temporal balloons for dgemm*

Figure 2.7: Resource multiplexing and the resultant system power, before and after one app* enters `psbox`. (a)(b): CPU schedule and power. When Calib3D runs, the system power consumption is lower because Calib3D's `psbox` forces the other CPU core to stay idle. (c)(d): DSP commands and power.

is difficult, if not impossible, on commodity computers. This is because OS resource scheduling and app behaviors are not guaranteed to be the same across different runs.

We further show the details of resource multiplexing, without and with `psbox`. As shown in Figure 2.7, `psbox` creates spatial and temporal balloons on CPU and DSP, respectively, and hence makes resource multiplexing respect the `psbox` boundaries. Outside of these balloons, the kernel multiplexes other apps freely as usual.

2.6.2 Performance impact

Latency increase All apps in the system may experience extra latency in some of their hardware access, if the hardware access happens to trigger resource balloon switch. Our implementation keeps the extra latency relatively low. Throughout our benchmark scenarios, the CPU scheduling latency is increased by tens of μs

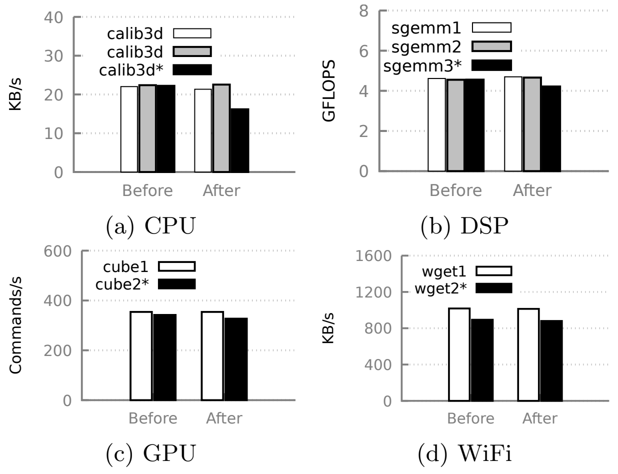


Figure 2.8: Throughputs of co-running app instances, before and after one instance (marked with *) enters `psbox`.

for task shootdown; the command dispatch latencies for GPU and DSP are increased by 1.8 ms and 100 ms on average, respectively.

The increased latency for WiFi packet transmission can be long, sometimes hundreds of ms. We found this is likely due to internal notification batching by the firmware of the WiLink NIC on the platform in Figure 2.4(b). In addition, the platform's wimpy CPU also contributes to interrupt handling latency. The combined software and hardware behaviors prolong draining phases.

Throughput loss As mentioned in Section 2.3, the exclusion of resource balloons may lead to lost sharing opportunities, which will reduce the total throughput on hardware. In our experiments, the total throughput loss can be noticeable, ranging from 0.9% (WiFi) to 9.8% (CPU). In face of the hardware throughput loss, we next discuss how well `psbox` maintains fairness among apps.

2.6.3 Confinement of throughput loss

Our system maintains throughput fairness among apps which may have different usages of `psbox`. To ease the comparison of app throughput loss, we co-run multiple instances of the same app. We show the throughputs of all the apps in Figure 2.8. When one app enters its `psbox`, it is the only one experiencing throughput loss; in comparison, the throughputs of other co-executing apps remain largely unaffected despite the total throughput decrease. Note that this is achieved without changing existing scheduling policies. This validates our key design of fully charging lost sharing opportunities to the sandboxed app (§2.4.2). We further test the robustness of our fairness guarantee under extremely high resource contention: we test the GPU driver, by co-running *browser* (in `psbox`) with *triangle*, a synthetic, intensive benchmark. Our results show that while

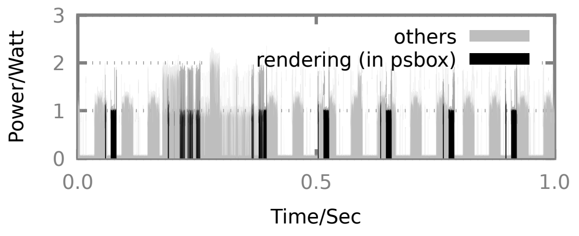


Figure 2.9: CPU power of a VR scenario. The rendering task enters `psbox` to observe its power and adapts accordingly

the GPU throughput of *browser* drops by 4× due to excessive draining time, that of triangle only decreases by 1%.

2.6.4 An end-to-end use case

We demonstrate the efficacy of `psbox` on a virtual reality (VR) scenario derived from a SDK demo (2K SLoC) [80]. The VR scenario lets a human user move her hand in order to control animated water waves. Two CPU tasks are running continuously. The *gesture* task processes video frames, identifies hand contours, and recognizes hand gestures. The *rendering* task translates the recognized gestures to wind directions, generates Phillips spectrum and 2D IFFT, and keeps refreshing a height map for animating the waves.

We, as app programmers, set to make *rendering* power-aware, so that it can trade the rendering fidelity (e.g. framerate, resolution) for lower power at run time. Without `psbox`, reasoning about the power of *rendering* is difficult due to power entanglement, as shown in Figure 2.9. To worsen the problem, the *gesture* task's workloads (and hence its power impacts) largely vary based on inputs, i.e., the number of contours in a frame. With `psbox`, the *rendering* task observes its power without the varying impacts of *gesture*. By adjusting the rendering fidelity based on its power observation in `psbox`, *rendering* achieves a wide range (8.9×) of power, from 90mW to 800mW.

Without `psbox` isolation, the *rendering* task will mistakenly take entangled power impacts into account. This incorrect power knowledge will mislead the app's power adaptation strategy, lowering energy efficiency or user experience. This VR scenario demonstrates the benefit from insulating power impacts.

2.7 Limitations & Discussions

Support `psbox` on extra hardware (1) **Display** may consume more than 50% of energy of a smartphone or tablet [81]. Fortunately, modern displays, notably OLED, are known free of power entanglement: each pixel contributes to the total power independently with little lingering power state [82]. Hence, OS may simply divide the display power among apps based the pixels produced by each app [25]. (2) **GPS** power is unaffected by concurrent uses once the device is operating. Therefore, the kernel can safely reveal GPS hardware power, except when the GPS is in off/suspended state, to individual `psbox`es. This avoids expensive power state virtualization as described in Section 2.4.1. (3) **Cellular interface** While temporal balloons for cellular interface (4G) can be constructed in a way similar to WiFi NICs (§2.4.2), a unique challenge is for the kernel to virtualize a cellular interface's power state [26]. In practice, the state transitions of a cellular interface are not controllable by the OS, but by the cellular standard that must be agreed with cellular towers. To this end, `psbox` will be made feasible on cellular interfaces through future hardware support. (4) **DRAM** consume 5% – 25% of system energy [81, 83]. Given that DRAM power is often metered at the level of DIMM [84] or controller [68], it is possible to realize `psbox` on DRAM through temporal balloons, However, it is challenging to track app DRAM usage and ensure fairness, for which the OS may need to consult hardware performance counters.

Userspace OS daemon Our current implementation focuses on kernel drivers. In other systems especially Android, multiplexing of app requests also happens in user-level daemons. It is possible to build `psbox` into these daemons by making their request multiplexing respect `psbox` boundaries.

Power-aware entities other than apps Some scenarios define alternative entities for power awareness, e.g. a user request served by multiple processes or even machines [85, 27]. `psbox` may enclose these entities in addition to an app. To do support this, each involved process or machine, as points of multiplexing, must be augmented to respect `psbox` boundaries.

Alternative OS mechanisms for supporting `psbox` Besides our Linux-based instantiation of `psbox` (§2.4.2), there are existing OS mechanisms that are absent in the mainline Linux yet suiting the need of enforcing `psbox`. First, scheduler activations [86] help move much of the CPU scheduling logic for `psbox` to user space. With such a mechanism, an app in its `psbox` spawns dummy threads to occupy unused cores for enforcing the balloon boundary; as the app's actual threads suspend/resume, the kernel notifies the app through upcalls, which adjust the number of dummy threads accordingly. Second, gang scheduling [72], commonly seen in real-time kernels, directly supports executing all threads in a `psbox` (a gang) simultaneously and enforces mutual exclusion among gangs. Third, systems like Dune [87] creates *per-app* virtualized views of the baremetal CPU hardware. This idea can be further extended to create per-app views of baremetal I/O devices, e.g. WiFi NIC. The virtualization cost can be further reduced by only enforcing power insulation (as required by `psbox`) while eschewing strong state isolation.

2.8 Road to Existing Ecosystems

To bring psbox and the power awareness into today's mobile and embedded ecosystems, the major challenges are twofold: i) processing high-rate power data with low hardware cost and ii) reusing mature APIs. We next discuss how these can be achieved by leveraging the *existing* software/hardware support for sensor data processing.

2.8.1 Hardware support

We next discuss how situ power metering (§ 2.2.2) can be realized atop existing hardware platforms with little addition.

Integrating with existing sensor hubs To harness rich sensors, most modern mobile/embedded devices incorporate sensor hubs, whose overall market is projected to exceed 2 billion units [88]. Sensors hubs are dedicated, extremely efficient processors for pre-processing sensor data, typically incarnated as Arm Cortex-M MCUs. As the volume of sensor hubs grows, their cost keeps decreasing: it is several US dollars per unit at the time of writing. They are penetrating most of the mobile/embedded SoC market.

By their design, sensor hubs directly suit pre-processing of power samples. A Cortex-M0 sensor hub clocked at 32 MHz consumes as low as 13 mW, and is capable of real-time processing of power data sampled at 1 KHz [89]. Such a sampling frequency already exceeds what is demanded by existing power-aware systems [32, 29, 11], and is able to differentiate microscopic power activities, e.g. scheduler context switch as shown in Figure 2.7.

Asymmetric cores We recognize that there exist mobile/embedded devices that do not have sensor hubs (yet). To increase efficiency of pre-processing power samples, they can leverage the lower-power cores in modern Arm architecture, e.g. big.LITTLE and DynamIQ [90]. The trend of increasing architectural asymmetry promises better processing efficiency.

Utilizing low-cost power sensors Modern mobile devices are already sensor-rich. For instance, the recent iPhone X has eight sensors of different types [91], ranging from the accelerometer to proximity sensor. Often, it is the types of sensors that differentiate mobile devices. While existing sensors are for *extrospection*, we believe it is equally valuable and feasible for the devices to additionally incorporate power sensors for *introspection*.

Power sensors can be very cost effective. The simplest power sensor can be a shunt resistor accompanied by an analog-to-digital converter (ADC); the latter can be further integrated into an on-chip I/O controller [65]. The combined cost is less than $1 [92]. Standalone current sensing ICs provide additional design convenience. At minor cost (around $1 per unit) [93], such an IC can be attached to a device's I2C bus with little extra hardware complexity. A typical current sensing IC [94] is capable of sampling three power rails at 500KHz simultaneously and returns digitalized power samples. They are already pervasive on experimental devices including Tegra X1 [95], X2 [96], Odroid XU3 [97].

2.8.2 Software support

To foster its adoption, `psbox` can further leverage the existing software infrastructure. This includes mature API frameworks and processing algorithms of sensor samples.

High-level sensor APIs Mobile OSes such as Android and iOS support tens of sensor types. They already offer mature APIs for apps to retrieve sensor data and subscribe to sensor events [98, 99]. The `psbox` native interface, as presented in Section 2.3, can be further wrapped under such APIs, adding a new "power" sensor type. For instance, through calling Android's `SensorManager.registerListener`, an app is able to retrieve power samples or register callbacks for "high power" events. This is exactly how today's apps monitor existing sensors such as accelerometers.

To cater to app-defined power events, existing sensor APIs can be further augmented with simple temporal predicates [100]. Through embedded scripting languages such as Lua or Javascript, the apps can specify events such as "frequent power spikes" or "power keeps increasing". The predicates are continuously evaluated over power samples by the OS or the sensor hub.

Sensor hub runtime As discussed before, processing of power samples can be offloaded to sensor hub hardware for efficient execution. Fortunately, there exist rich runtime software on sensor hubs that facilitates such offloading. First, existing commodity sensor hub runtimes, e.g. SenseMe [101], are already mature; they provide an arsenal of signal processing algorithms, e.g. denoising, that can pre-process power samples with high efficiency. Second, recent research has proposed a variety of techniques for simplifying new code development for sensor hubs. For instance, our work Reflex [102] creates a software distributed shared memory between CPU and sensor hubs; MobileHub [103] automatically learns sensor events and produces event detection code for sensor hubs; Sidewinder [104] supports composition of parameterized, pre-defined algorithms for sensor hubs. These rich techniques are applicable to development of power data processing algorithms for sensor hubs.

2.9 Related Work

Power metering Much work infers power from software-visible events, such as syscall activities [24, 25], kernel activities [26], hardware states [59, 105, 21, 27, 22, 58, 28]. They often construct linear models either during development [21, 27, 24, 25, 26, 59, 28] or at run time [22, 58, 105]. Although convenient, energy modeling is limited by complex hardware [57] and high variation in semiconductor process [56]. Intel RAPL [68] is a CPU feature: the firmware monitors hardware activities and infers power based on pre-defined models. Yet, lacking timestamps, RAPL power samples can hardly be mapped to software activities at fine time granularity [106, 107]. Direct measurement allows accurate power metering through external multimeters [11, 29], fine-grained hardware instrumentation [30], smart switching regulators [108], smart battery interfaces [31, 32], and specialized metering circuits [109, 110]. Regardless of metering approaches, power entanglement is inevitable as explained in Section 2.2, which necessitates power sandboxes.

Power accounting heuristics As mentioned in Section 2.2, prior work attributes power using various heuristics. Eprof [24, 25] attributes lingering tail power to the last triggering entity. HaPPy [111] splits hyperthreading CPU power based on per-thread aggregated cycles. Ghanei *et al.* [112] track asynchronous hardware use and evenly divides power among concurrent apps. Dong *et al.* [23] attribute energy based game theory. Power Containers [27] meters per core power, while evenly splitting the power of shared resources among active cores. Joulemeter [113] models per-VM power in the server by inferring system power from hardware activities reported by OS or performance counters. However, without eliminating power entanglement, they suffer from the inadequacies described in Section 2.2. It is also difficult to apply the performance counter-based approaches to many accelerators and I/Os that lack performance counters.

OS-level power management Power management has been a key OS responsibility. Odyssey [11, 29] enables the OS to guide apps for energy-aware adaptation. ECOSystem [21] and Cinder [114] present OS-level abstractions for energy. Koala [105] builds energy models in the kernel and sets performance/efficiency dynamically. Rao *et al.* [115] build a controller to balance performance loss and energy saving, based on application-specific data profiled offline. OS also manages power for accelerators [116] and I/O devices [117, 12].

However, none prior work presented virtualized power view to individual apps.

Power side-channel attacks Prior work exploits power side channels to steal private information on smartphones [35], recover cryptographic keys [60, 61], reveal mobile user geolocations [34], and leak information across virtual machines [36]. However, few systems prevent power side channels through active resource management as we do.

OS resource scheduling Several proposals on scheduling are in particular related to `psbox`. GPU scheduling has been advocated for long. TimeGraph [118] prioritizes and isolates performance of competing apps. PTask [119], Gdev [120], and Menychtas *et al.* support fair sharing of GPU. ShuffleDog [63] prioritizes UI tasks through resource scheduling. SmartIO [64] reduces app delay by prioritizing disk reads over writes. Energy discounted computing [53] co-schedules tasks to improve total system efficiency. Complementary to `psbox`, these scheduling proposals target performance or efficiency for power-unaware apps.

2.10 Conclusions

An app's power observation should be insulated from the impacts of concurrent apps. We introduce power sandbox, a new OS principal capturing the power of the enclosed app and its vertical environment. To support power sandbox, our key techniques are two: to allocate exclusive resource partitions at fine granularities and bill the lost sharing opportunities; to virtualize hardware power states. Our experience shows that power sandbox simplifies reasoning, eliminates security vulnerability, and still ensures fairness among apps.

Chapter 3

Transkernel: Bridging Monolithic Kernels to Peripheral Cores

3.1 Introduction

Driven by periodic or background activities, modern embedded platforms[1] often run a large number of ephemeral tasks. Example tasks include acquiring sensor readings, refreshing smart watch display [121], push notifications [122], and periodic data sync [123]. They drain a substantial fraction of battery, e.g., 30% for smartphones [124, 125] and smart watches [126], and almost the entire battery of smart things for surveillance [127]. To execute an ephemeral task, a commodity OS kernel, typically implemented in a monolithic fashion, drives the whole hardware platform out of deep sleep beforehand (i.e., "resume") and puts it back to deep sleep afterwards (i.e., "suspend"). During this process, the kernel consumes much more energy than the user code [121], up to 10× shown in recent work [122].

Why is the kernel so inefficient? Recent studies [128, 12, 121] show the bottlenecks as two kernel phases called *device suspend/resume* as illustrated in Figure 3.1. In the phases, the kernel operates a variety of IO devices (or *devices* for brevity). It invokes device drivers, cleans up pending IO tasks, and ensures devices to reach expected power states. The phases encompass concurrent execution of drivers, deferred functions, and hardware interrupts; they entail numerous CPU idle epochs; their optimization is proven difficult (§3.2) [12, 129, 130].

We deem that device suspend/resume mismatches CPU. It instead would be much more efficient on low-power, microcontroller-like cores, as exemplified by ARM Cortex-M. These cores are already incorporated

[1]This paper focuses on battery-powered computers such as smart wearables and smart things. They run commodity OSes such as Linux and Windows. We refer to them as embedded platforms for brevity.

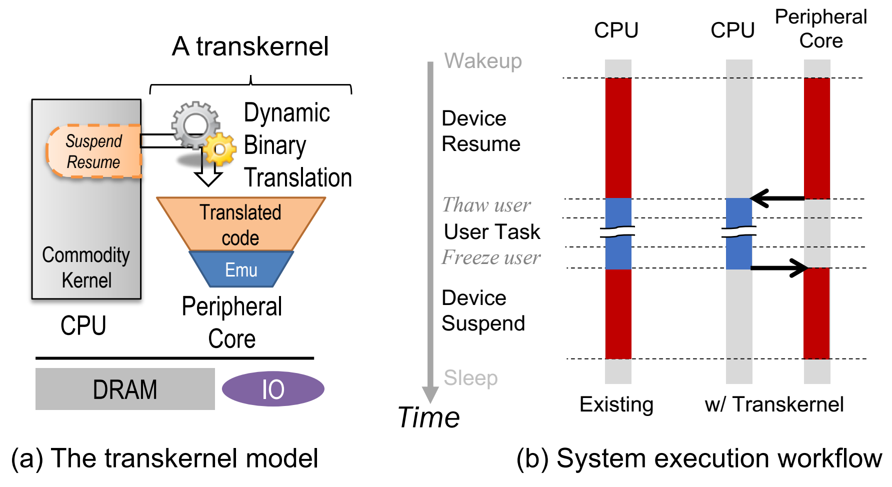


Figure 3.1: An overview of Transkernel.

as *peripheral cores* on a wide range of modern system-on-chips (SoCs) used in production such as Apple Watch [131] and Microsoft Azure Sphere [132]. On IO-intensive workloads, a peripheral core delivers is much more efficient than the CPU due to lower idle power and higher execution efficiency [102, 133, 134, 135]. Note that running *user code* (which often builds atop POSIX) on peripheral cores is a non-goal: on one hand, doing so would gain much less efficiency due to fewer idle epochs in user execution; on the other hand, doing so requires to support a much more complex POSIX environment on peripheral cores.

Offloading the execution of a commodity, monolithic kernel raises practical challenges, not only i) that the peripheral core has a different ISA and wimpy hardware but also ii) that the kernel is complex and rapidly evolving [136]. Many OS proposals address the former while being inadequate in addressing the latter [137, 138, 139, 140, 141]. For instance, one may refactor a monolithic kernel to span it over CPU and a peripheral core; the resultant kernel, however, depends on a wide binary interface (ABI) for synchronizing state between the two ISAs. This interface is brittle. As the upstream kernel evolves, maintaining *binary compatibility* across different ISAs inside the kernel itself soon becomes unsustainable. Instead, we argue for the code running on peripheral cores to enjoy firmware-level compatibility: developed and compiled *once*, it should work with *many* builds of the monolithic kernel – generated from different configurations and source versions.

Our response is a radical design called *transkernel*, a lightweight virtual executor empowering a peripheral core to run specific kernel phases – device suspend/resume. Figure 3.1 overviews the system architecture. A transkernel executes unmodified kernel binary through cross-ISA, dynamic binary translation (DBT), a technique previously regarded as expensive [137] and never tested on microcontroller-like cores to our knowledge. Underneath the translated code, a small set of emulated services act as lightweight, drop-in replacements for their counterparts in the monolithic kernel. Four principles make transkernel practical: i) translating stateful code while emulating stateless kernel services; ii) identifying a narrow, stable translation/emulation interface; iii) specializing for hot paths; iv) exploiting ISA similarities for DBT.

We demonstrate a transkernel prototype called ARK (**A**n a**R**m trans**K**ernel). Atop an ARM SoC, ARK runs on a Cortex-M3 peripheral core (with only 200 MHz clock and 32KB cache) alongside Linux running on a Cortex-A9 CPU. ARK transparently translates unmodified Linux kernel drivers and libraries. It depends on a binary interface consisting of only 12 Linux kernel functions and one kernel variable, which are stable for years. ARK offers complete support for device suspend/resume in Linux, capable of executing diverse drivers that implement rich functionalities (e.g., DMA and firmware loading) and invoke sophisticated kernel services (e.g., scheduling and IRQ handling). As compared to native kernel execution, ARK only incurs 2.7× overhead, 5.2× lower than a baseline of off-the-shelf DBT. ARK reduces system energy by 34%, resulting in tangible battery life extension under real-world usage.

We make the following contributions on OS and DBT:

• We present the transkernel model. In the design space of OSes for heterogeneous multi-processors, the transkernel represents a novel point: it combines DBT and emulation for bridging ISA gaps and for catering to core asymmetry, respectively.

• We present a transkernel implementation, ARK. Targeting Linux, ARK presents specific tradeoffs between kernel translation versus emulation; it identifies a narrow interface between the two; it contributes concrete realization for them.

• Crucial to the practicality of ARK, we present an inverse paradigm of cross-ISA DBT, in which a microcontroller-like core translates binary built for a full-fledged CPU. We contribute optimizations that systematically exploit ISA similarities. Our result demonstrates that while cross-ISA DBT is typically used under the assumption of efficiency *loss*, it can enable efficiency *gain*, even on off-the-shelf hardware.

The source code of ARK can be found at `http://xsel.rocks/p/transkernel`.

3.2 Motivations

We next discuss device suspend/resume, the major kernel bottleneck in ephemeral tasks, and that it can be mitigated by running on a peripheral core. We show difficulties in known approaches and accordingly motivate our design objectives.

3.2.1 Kernel in device suspend/resume

Expecting a long period of system inactivity, an OS kernel puts the whole platform into deep sleep: in brief, the kernel synchronizes file systems with storage, freezes all user tasks, turns off IO devices (i.e., device suspend), and finally powers off the CPU. To wake up from deep sleep, the kernel performs a mirrored procedure [142]. In a typical ephemeral task, the above kernel execution takes hundreds of milliseconds [143] while the user execution often takes tens of milliseconds [121]; the kernel execution often consumes several times more energy than the user execution [122].

Problem: device suspend/resume By profiling recent Linux on multiple embedded platforms, our pilot study [12] shows the aforementioned kernel execution is bottlenecked by device suspend/resume, in which the kernel cleans up pending IO tasks and manipulates device power states. The findings are as follows. i) ***Device suspend/resume is inefficient.*** It contributes 54% on average and up to 66% to the total kernel energy consumption. CPU idles frequently in numerous short epochs, typically in milliseconds. ii) ***Devices are diverse.*** On a platform, the kernel often suspends and resumes tens of different devices. Across platforms, the bottleneck devices are different. iii) ***Optimization is difficult.*** Device power state transitions are bound by slow hardware and low-speed buses, as well as physical factors (e.g., voltage ramp-up). While Linux already parallelizes power transitions with great efforts [129, 130], many power transitions must happen sequentially per *implicit* dependencies of power, voltage, and clock. As a result, CPU idle constitutes up to 68% of the device suspend/resume duration.

Challenge: Widespread, complex kernel code Device suspend/resume invokes multiple kernel layers [136, 144]. Specifically, it invokes functions in individual drivers (e.g., MMC controllers), driver libraries (e.g., the generic clock framework), kernel libraries (e.g., for radix trees), and kernel services (e.g., scheduler). In a recent Linux source tree (4.4), we find that over 1000 device drivers, which represent almost all driver classes, implement suspend/resume callbacks in 154K SLoC. These callbacks in turn invoke over 43K SLoC in driver libraries, 8K SLoC in kernel libraries, and 43K SLoC in kernel services. The execution is control-heavy, with dense branches and callbacks.

Opportunities We observe the following kernel behaviors in device suspend/resume. i) ***Low sensitivity to execution delay*** On embedded platforms, most ephemeral tasks are driven by background activities [122, 145, 124]. This contrasts to many servers for interactive user requests [146, 145]. ii) ***Hot kernel paths*** In successful suspend/resume, the kernel acquires all needed resources and encounters no failures [147]. Off the hot paths, the kernel handles rare events such as races between IO events, resource shortage, and hardware failures. These branches typically cancel the current suspend/resume attempt, perform diagnostics, and retry later. Unlike hot paths, they invoke very different kernel services, e.g., syslog. iii) ***Simple concurrency***

SoC	**Cores**	**ISAs**	**Shared DRAM?**	**Mapping kern mem?**	**Shared IRQ**
OMAP4460 [148] (2010)	A9+M3	v7a+v7m	Full	Yes. MPU	39/102
AM572x [149] (2014)	A15+M4	v7a+v7m	Full	Yes. MPU	32/92
i.MX6SX [150] (2015)	A9+M4	v7a+v7m	Full	Yes. MPU	85/87
i.MX7 [151] (2017)	A7+M4	v7a+v7m	Full	Yes. MPU	88/90
i.MX8M [152] (2018)	A53+M4	v8a+v7m	Full	Yes. MPU	88/88
MT3620 [132] (2018)*	A7+M4	v7a+v7m	Full	Likely. MPU	Likely most

Table 3.1: Transkernel hardware model fits many popular SoCs which are used in popular products such as Apple Watch and Azure Sphere. Section 3.7.5 discusses caveats. *: lack public technical details.

exists among the syscall path (which initiates suspend/resume), interrupt handlers, and deferred kernel work. The concurrency is for hardware asynchrony and kernel modularity rather than exploiting multicore parallelism.

Summary: design implications Device suspend/resume shall be treated systematically. We face challenges that the invoked kernel code is diverse, complex, and cross-layer; we see opportunities that allow focusing on hot kernel paths, specializing for simple concurrency, and gaining efficiency at the cost of increased execution time.

3.2.2 A peripheral core in a heterogeneous SoC

Hardware model We set to exploit peripheral cores already on modern SoCs. Hence, our software design only assumes the following hardware model which fits a number of popular SoCs as listed in Table 3.1.

1. ***Asymmetric processors***: In different coherence domains, the CPU and the peripheral core offer disparate performance/efficiency tradeoffs. The peripheral core has memory protection unit (MPU) but no MMU, incapable of running commodity OSes as-is.

2. ***Heterogeneous, yet similar ISAs***: The two processors have different ISAs, in which many instructions have similar *semantics*, as will be discussed below.

3. ***Loose coupling***: The two processors are located in separate power domains and can be turned on/off independently.

4. ***Shared platform resources***: Both processors share access to platform DRAM and IO devices. Specifically, the peripheral core, through its MPU, should map all the kernel code/data at identical virtual addresses as the CPU does. Both processors must be able to receive interrupts from the devices of interest, e.g., MMC; they may, however, see different interrupt line numbers of the same device.

How can peripheral cores save energy? They are known to deliver high efficiency for IO-heavy workloads [102, 134, 153, 135, 103]. Specifically, they benefit the kernel's device suspend/resume in the following ways. i) A peripheral core can operate while leaving the CPU offline. ii) The idle power of a peripheral core is often one order of magnitude lower [133, 154], minimizing system power during core idle periods. iii) Its simple microarchitecture suits kernel execution, whose irregular behaviors often see marginal benefits from powerful microarchitectures [155]. Note that a peripheral core offers much higher efficiency than a LITTLE core as in ARM big.LITTLE [156], which mandates a homogeneous ISA and tight core coupling. We will examine big.LITTLE in Section 3.7.

ISA similarity On an SoC we target, the CPU and the peripheral core have ISAs from the same family, e.g., ARM. The two ISAs often implement similar instruction *semantics* despite in different *encoding*. The common examples are SoCs integrating ARMv7a ISA and ARMv7m ISA [150, 151, 149, 132, 148]. Other families also provide ISAs amenable to same-SoC integration, e.g., NanoMIPS and MIPS32. We deem that the ISA similarities are *by choice*. i) For ISA designers, it is feasible to explore performance-efficiency tradeoffs within one ISA family, since the family choice is merely about instruction *syntax* rather than *semantics* [157]. ii) For SoC vendors, incorporating same-family ISAs on one chip simplifies software efforts [158], silicon design, and ISA licensing.

3.2.3 OS design space exploration

We set to realize heterogeneous execution for an *existing* monolithic kernel.

How about refactoring the kernel and cross-compiling statically? One may be tempted to modify a monolithic kernel (we use Linux as the example below) [133, 137] to be one unified source tree; the tree shall be cross-compiled into a kernel binary for CPU and a "peripheral kernel" for the peripheral core. This approach results in an OS structure shown in Figure 3.2(a). Its key drawback is the two interfaces that are difficult to implement and maintain, shown as 〰 in the figure.

① The interface between two heterogeneous ISAs, as needed for resolving inter-kernel data dependency. Through the interface, both kernels synchronize their kernel state, e.g., devices configurations, pending IO tasks, and locks, before and after the offloading. Built atop shared memory [133, 137, 160], the interface is essentially an agreement on thousands of shared Linux kernel data types, including their semantics and/or

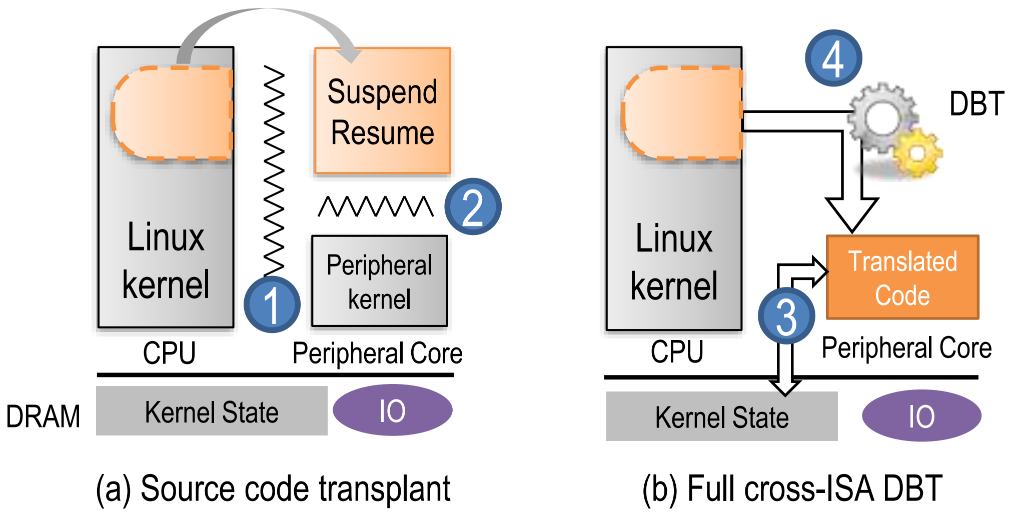
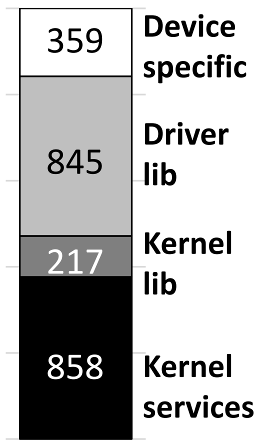


Figure 3.2: Alternative ways for offloading kernel phases.



(a) # of functions

From \ To	v2.6	v3.16	v4.4	v4.9	v4.17
v2.6 (Jan 2011)		155	196	194	213
		378	*385*	*384*	*395*
v3.16 (Aug 2014)	500		155	163	214
	717		*674*	*661*	*707*
v4.4 (Jan 2016)	640	216		55	159
	855	*780*		*721*	*828*
v4.9 (Dec 2016)	816	354	214		173
	938	*848*	*797*		*848*
v4.17 (Jul 2018)	1075	606	498	384	
	1111	*1043*	*1060*	*1015*	

(b) # of functions (upper) & types (lower) w/ changed ABI across kernel versions

Figure 3.3: Counts of Linux kernel functions referenced by device suspend/resume, showing (a) the functions are rich and diverse and (b) their ABI change is substantial over time. Exported functions only. Build config: omap2defconfig. ABI changes detected with ABI compliance checker [159].

memory layout. The agreement is brittle, as it is affected by ISA choices, kernel configurations, and kernel versions. Hence, keeping data types consistent across ISAs entails tedious tweak of kernel source and configurations [161, 160]. As Greg Kroah-Hartman puts, "you will go insane over time if you try to support this kind of release, I learned this the hard way a long time ago." [162]

② The interface between the transplant code and the peripheral kernel, as needed for resolving functional dependency. In principle, this interface is determined by the choice of transplant boundary. In prior work, the example choices include the interface of device-specific code [161, 160, 163], that of driver classes [164, 165], or that of driver libraries [133]. All these choices expose at least hundreds of Linux kernel functions on this interface, as summarized in Figure 3.3(a). This is due to Linux's diverse, sophisticated drivers. Implementing such an interface is daunting; maintaining it is even more difficult due to significant ABI changes [166] as shown in Figure 3.3(b).

In summary, all these difficulties root in the peripheral kernel's *deep dependency* on the Linux kernel. This is opposite to the common practice: heterogeneous cores to run their own "firmware" that has little dependency on the Linux kernel. This is sustainable because the firmware stays compatible with many builds of Linux.

How about virtual execution? Can we minimize the dependency? One radical idea would be for a peripheral core to run the Linux kernel through virtual execution, as shown in Figure 3.2(b). Powered by DBT, virtual execution allows a *host* processor (e.g., the peripheral core) to execute instructions in a foreign *guest* ISA (e.g., the CPU). Virtual execution is free of the above interface difficulties: the translated code precisely reproduces the kernel behaviors and directly operates the kernel state (③). The peripheral core interacts with Linux through a low-level, stable interface: the CPU's ISA (④).

The problem, however, is the high overhead of existing cross-ISA DBT [167]. It is further exacerbated by our *inverse* DBT paradigm: whereas existing cross-ISA DBT is engineered for a brawny host emulating a weaker guest (e.g., an x86 desktop emulating an ARM smartphone) [168, 169], our DBT host, a peripheral core, shall serve a full-fledged CPU. A port of popular DBT exhibits up to 25× slowdown as will be shown in §3.7. Such overhead would negate any efficiency promised by the hardware and result in overall efficiency *loss*. Furthermore, cross-ISA DBT for the *whole* Linux kernel is complex [170]. A peripheral core lacks necessary environment, e.g., multiple address spaces and POSIX, for developing and debugging such complex software.

3.2.4 Design objective

We therefore target threefold objective.

G1. Tractable engineering. We set to reuse much of the kernel source, in particular the drivers that are impractical to build anew. We target simple software for peripheral cores.

G2. Build once, work with many. One build of the peripheral core's software should work with a commodity kernel's binaries built from a wide range of configurations and source versions. This requires the former to interact with the latter through a stable, narrow ABI.

G3. Low overhead. The offloaded kernel phases should yield a tangible efficiency gain.

3.3 The Transkernel Model

Running on a peripheral core, a transkernel consists of two components: a DBT engine for translating and executing the unmodified kernel binary; a set of emulated, minimalistic kernel services that underpin the translated kernel code, as will be described in detail in Section 3.4. A concrete transkernel implementation targets a specific commodity kernel, e.g., Linux. A transkernel does not execute user code in ephemeral tasks as stated in Section 3.1.

The transkernel follows four principles:

1. Translating stateful code; emulating stateless services By *stateful code*, we refer to the offloaded code that must share states with the kernel execution on CPU. The stateful code includes device drivers, driver libraries, and a small set of kernel services. They cover the most diverse and widespread code in device suspend/resume (§3.2). By translating their binaries, the transkernel reuses the commodity kernel without maintaining wide, brittle ABIs. (objective G1, G2)

The transkernel emulates a tiny set of kernel services. We relax their semantics to be stateless, so that their states only live within one device suspend/resume phase. Being stateless, the emulated services do not need to synchronize states with the kernel on CPU over ABIs. (G2)

2. Identifying a narrow, stable translation/emulation ABI The ABI must be unaffected by kernel configurations and unchanged since long in the kernel evolution history. (G2)

3. Specializing for hot paths In the spirit of OS specialization [171, 172, 173], the transkernel only executes the hot path of device suspend/resume; in the rare events of executing off the hot path, it transparently falls back on CPU. The transkernel's emulated services seek *functional equivalence* and only implement features needed by the hot path; they do not precisely reproduce the kernel's behaviors. (G1)

4. Exploiting ISA similarities for DBT The transkernel departs from generic cross-DBT that bridges arbitrary guest/host pairs; it instead systematically exploits similarities in instructions semantics, register usage, and control flow transfer. This makes cross-ISA DBT affordable. (G3)

Limitations First, across ISAs of which instruction semantics are substantially different, e.g., ARM and x86, the transkernel may see diminishing or even no benefit. Second, the transkernel's longer delays (albeit lower energy) may misfit latency-sensitive contexts, e.g., for waking up platforms in response to user input. Our current prototype relies on heuristics to recognize such contexts and falls back on the CPU accordingly (Section 3.4).

In Section 3.4 below we describe how to apply the model to a concrete transkernel, in particular our translation/emulation decisions for major kernel services, and our choices of the emulation interface. We will

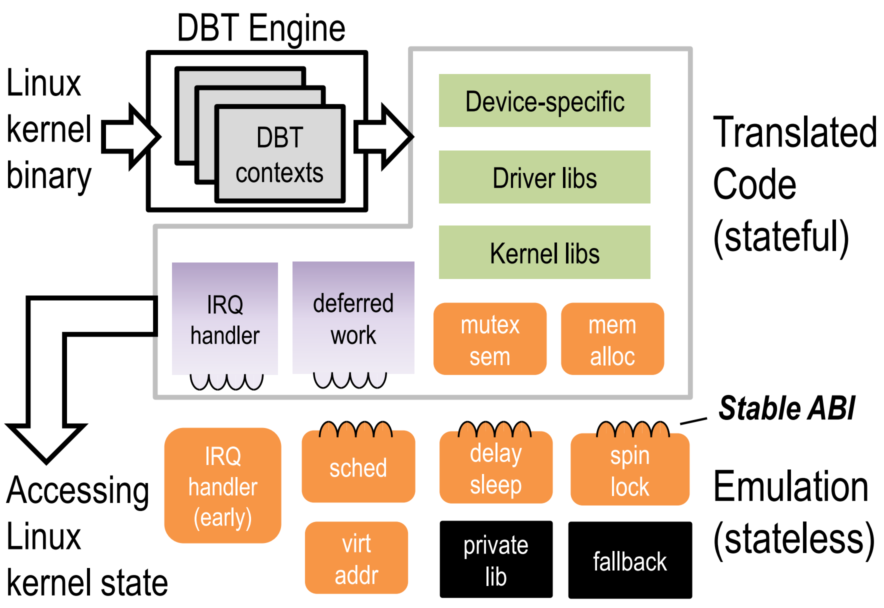


Figure 3.4: The ARK structure on a peripheral core.

describe DBT in Section 3.5.

3.4 ARK: An ARM Transkernel

Targeting an ARM SoC, we implement a transkernel called ARK. The SoC encompasses a popular combination of ISAs: ARMv7A for its CPU and ARMv7m for its peripheral core. The CPU runs Linux v4.4.

Offloading workflow ARK is shipped as a standalone binary for the peripheral core, accompanied by a small Linux kernel module for control transfer between CPU and the peripheral core. We refer to such control transfer as *handoff*. Prior to a device suspend phase, the kernel shuts down all but one CPU cores, passes control to the peripheral core, and shuts down the last CPU core. Then, ARK completes the device phase in order to suspend the entire platform. Device resume is normally executed by ARK on the peripheral core; in case of urgent wakeup events (e.g., a user unlocking a smart watch screen), the kernel resumes on CPU with native execution.

System structure As shown in Figure 3.4, ARK runs a DBT engine, its emulated kernel services, and a small library for managing the peripheral core's private hardware, e.g., interrupt controllers. The emulated services serves downcalls (∩∩∩) from the translated code and makes upcalls (∪∪∪) into the translated code. Table 3.2 summarizes the interfaces. Upon booting, ARK replicates Linux kernel's linear memory mappings for addressing kernel objects in shared memory [133, 160]. ARK maps I/O regions with MPU and time-multiplexes the regions on the MPU entries.

Kernel services	Implementations & reasons
Scheduler (§3.4.1)	Emulated. Reason: simple concurrency.
IRQ handler (§3.4.2)	Early stage emulated; then translated
HW IRQ controller (§3.4.2)	Emulated. Reason: core-specific
Deferred work (§3.4.3)	Translated. Reason: stateful
Spinlocks (§3.4.4)	Emulated. Reason: core-specific
Sleepable locks (§3.4.4)	Fast path translated. Reason: stateful
Slab/Buddy allocator (§3.4.5)	Fast path translated. Reason: stateful
Delay/wait/jiffies (§3.4.6)	Emulated. Reason: core-specific

jiffies	udelay()	msleep()	tasklet_schedule()	irq_thread()
ktime_get()	queue_work_on()	worker_thread()	run_local_timers()	
generic_handle_irq()	schedule()	async_schedule()*	do_softirq()*	

*=ABI unchanged since 2014 (v3.16); others unchanged since 2011 (v2.6).

Table 3.2: Top: Kernel services supported by ARK. Bottom: Linux kernel ABI (12 funcs+1 var) ARK depends on. ARK offers complete support for device suspend/resume in Linux.

To support concurrency in the offloaded kernel phases, ARK runs multiple DBT contexts. Each context has its own DBT state (e.g., virtual CPU registers and a stack), executing DBT and emulated services independently. Context switch is as cheap as updating the pointer to the DBT state.

ARK executes the hot paths. Upon entering cold branches pre-defined by us, e.g., kernel `WARN()`, ARK migrates all the DBT contexts of *translated* code back to the CPU and continues as *native* execution there (§3.6).

3.4.1 A Scheduler of DBT Contexts

ARK emulates a scheduler which shares no state, e.g., scheduling priorities or statistics, with the Linux scheduler on the CPU. Corresponding to the simple concurrency model of suspend/resume (§3.2), ARK eschews reproducing Linux's preemptive multithreading but instead maintains and switches among cooperative DBT contexts: one primary context for executing the syscall path of suspend/resume, one for executing IRQ handlers (§3.4.2), and multiple for deferred work (§3.4.3). Managing no more than tens of contexts, ARK uses simple, round-robin scheduling. It begins the execution in the syscall context; when the syscall context blocks (e.g., by calling `msleep()`), ARK switches to the next ready context to execute deferred functions until they finish or block. When an interrupt occurs, ARK switches to the IRQ context to execute the kernel interrupt handler (§3.4.2).

3.4.2 Interrupt and Exception Handling

During the offloaded device phase, all interrupts are routed to the peripheral core and handled by ARK.

Kernel interrupt handlers ARK emulates a short, early stage of interrupt handling while translating the kernel code for the remainder. This is because this early stage is ISA-specific (e.g., for manipulating the interrupt stack), on which the CPU (v7a) and the peripheral core (v7m) differ. Hence, the emulated services implement a v7m-specific routine and install it as the hardware interrupt handler. Once an interrupt happens, the routine is invoked to finish the v7m-specific task and make an upcall to the kernel's ISA-neutral interrupt handling routine (listed in Table 3.2), from where the ARK translates the kernel to finish handling the interrupt.

Hardware interrupt controller ARK emulates the CPU's hardware interrupt controller. This is needed as the two cores have separate, heterogeneous interrupt controllers. The CPU controller's registers are unmapped in the peripheral core; upon accessing them (e.g., for masking interrupt sources) the translated code triggers faults. ARK handles the faults and operates the peripheral core's controller accordingly.

Exception: unsupported We don't expect any exception in the offloaded kernel phases. In case exception happens, ARK uses its fallback mechanism (§3.6) to migrate back to CPU.

3.4.3 Deferred Work

Device drivers frequently schedule functions to be executed in the future. ARK translates the Linux services that schedule the deferred work as well as the actual execution of the deferred work. ARK chooses to translate such services because they must be *stateful*: the peripheral core may need to execute deferred work created on the CPU prior to the offloading, e.g., freeing pending WiFi packets; it may defer new work until after the completion of resume.

ARK maintains dedicated DBT contexts for executing the deferred work (Section 3.4.1). While the Linux kernel often executes deferred work in kernel threads (daemons), our insight is that deferred work is oblivious to its execution context (e.g., a real Linux thread or a DBT context in ARK). Beyond this, ARK only has to run the deferred work that may *sleep* with separate DBT contexts so that they do not block other deferred work. From these DBT contexts, ARK translates the main functions of the aforementioned kernel daemons, which retrieve and invoke the deferred work.

Threaded IRQ defers heavy-lifting IRQ work (i.e., deferred work) to a kernel thread which executes the work after the hardware IRQ is handled. A threaded IRQ handler may sleep. Therefore, ARK maintains per-IRQ DBT contexts for executing these handlers. Each context makes upcalls into `irq_thread()` (the main function of threaded irq daemon, listed in Table 3.2).

Tasklets, workitems, and timer callbacks The kernel code may dynamically submit short, non-sleepable functions (tasklets) or long, sleepable functions (workitems) for deferred execution. Kernel daemons (softirq and kworker) execute tasklets and workitems, respectively.

ARK creates one dedicated context for executing all non-sleepable tasklets and per-workqeueue contexts for executing workitems so that one workqueue will not block others. These contexts make upcalls to the main functions of the kernel daemons (`do_softirq()`, `worker_thread()`, and `run_local_timers()`), translating them for retrieving and executing deferred work.

3.4.4 Locking

Spinlocks ARK emulates spinlocks, because their implementation is core-specific and that ARK can safely assume all spinlocks are free at handoff points: as described in early Section 3.4, handoff happens between one CPU core and one peripheral core, which do not hold any spinlock; all other CPU cores are offline and cannot hold spinlocks. Hence, ARK emulates spinlock acquire/release by pausing/resuming interrupt handling. This is because ARK runs on one peripheral core and the only hardware concurrency comes from interrupts.

Sleepable locks ARK translates sleepable locks (e.g., mutex, semaphore) because these locks are stateful: for example, the kernel's clock framework may hold a mutex preventing suspend/resume from concurrently changing clock configuration [174]. Furthermore, mutex's seemingly simple interface (i.e., compare & exchange in fast path) has *unstable* ABI and therefore unsuitable for emulation: a mutex's reference count type changes from `int` to `long` (v4.10), breaking the ABI compatibility. The translated operations on sleepable locks may invoke spinlocks or the scheduler, e.g., when updating reference counts or putting the caller to sleep, for which the translated execution makes downcalls to the emulated services.In practice, no sleepable lock is held prior to system suspend.

3.4.5 Memory Allocation

The device phase frequently requests dynamic memory, often at granularities of tens to hundreds of bytes. By Linux design, such requests are served by the kernel slab allocator backed by a buddy system for page allocation (fast path); when the physical pages runs low, the kernel may trigger swapping or kill user processes (slow path).

ARK provides memory allocation as a stateful service. It translates the kernel code for the fast path, including the slab allocator and the buddy system. In the case that the allocation enters the slow path (e.g., due to low physical memory), ARK aborts offloading; fortunately, our stress test suggests such cases to be extremely rare, as will be reported in Section 3.7. With a stateful allocator, the offloaded execution can free dynamic memory allocated during the kernel execution on CPU, and vice versa. Compare to prior work that instantiates per-kernel allocators with split physical memory [133], ARK reduces memory fragmentation and avoids tracking *which* processor should free *what* dynamic memory pieces. Our experience in Section 3.7 show that ARK is able to handle intensive memory allocation/free requests such as in loading firmware to a WiFi NIC.

3.4.6 Delays & Timekeeping

Delays ARK emulates `udelay()` and `msleep()` for busy waiting and sleeping. ARK converts the expected wait time to the hardware timer cycles on the peripheral core. ARK implements `msleep()` by pausing scheduling the caller context.

jiffies The Linux kernel periodically updates jiffies, a global integer, as a low-overhead measure of elapsed time. By consulting the peripheral core's hardware timer, ARK directly updates the jiffies. It is thus the only shared variable on the kernel ABI that ARK depends (all others are functions).

3.5 The Cross-ISA DBT Engine

A Cross-ISA DBT Primer DBT, among its other uses [175, 176, 177], is a known technique allowing a *host* processor to execute instructions in a foreign *guest* ISA. In such cross-ISA DBT, the host processor runs a program called DBT engine. At run time, the engine reads in guest instructions, translates them to host instructions based on the engine's built-in *translation rules*, and executes these host instructions. The engine translates guest instructions in the unit of translation block – a sequence (typically tens) of guest instructions that has one entry and one or more exits. After translating a block, the engine saves the resultant host instructions to its *code cache* in the host memory, so that future execution of this translated block can be directed to the code cache.

Design overview We build ARK atop QEMU [170], a popular, opensource cross-ISA DBT engine. ARK inherits QEMU's infrastructure but departs from its generic design which translates between arbitrary ISAs. ARK targets two well-known DBT optimizations: i) to emit as few host instructions as possible; ii) to exit from the code cache to the DBT engine as rarely as possible. We exploit the following similarities between the CPU's and the peripheral core's ISAs (ARMv7a & ARMv7m):

1. Most v7a instructions have v7m counterparts with identical or similar semantics, albeit in different encoding. (§3.5.1)
2. Both ISAs have the same general purpose registers. The condition flags in both ISAs have same semantics. (§3.5.2)
3. Both ISAs use program counter (PC), link register (LR), and stack pointer (SP) in the same way. (§3.5.3)

Beyond the similarities, the two ISAs have important discrepancies. Below, we describe our exploitation of the ISA similarities and our treatment for *caveats*.

3.5.1 Exploiting Similar Instruction Semantics

	Category	Cnt	v7m
w/ CNTPRT	Identity	447	1
	Side effect	52	3-5
	Const constraints	22	2-5
	Shift modes	10	2
w/o counterparts		27	2-5
Total (v7a)		558	

Table 3.3: Translation rules for v7a instructions. Column 3: the number of v7m instructions emitted for one v7a instruction

We devise translation rules with a principled approach by parsing a machine-readable, formal ISA specification recently published by ARM [178]. Our overall guideline is to map each v7a instruction to one v7m instruction that has identical or similar semantics. We call them *counterpart* instructions. For a counterpart instruction with similar (yet non-identical) semantics, ARK emits a few "amendment" v7m instructions to make up for the semantic gap. The resultant translation rules are based on individual guest instructions, different from translation rules based on one or more translation blocks commonly seen in cross-ISA DBT [179]. This is because semantics similarities allows identity translation for most guest instructions. Amendment instructions are oblivious to interrupts/exceptions: as stated in §3.4.2, ARK defers IRQ handling to translation block boundary and expects no exceptions.

Table 3.3 summarizes ARK's translation rules for all 558 v7a instructions. Among them, 80% can be translated with identity rules, for which ARK only needs to convert instruction encoding at run time. 15% of v7a instructions have v7m counterparts but may require amendment instructions, which fortunately fall into a few categories: i) ***Side effects.*** After load/store, v7a instructions may additionally update memory content or register values (shown in Table 3.4, G1). ARK emits amendment instructions to emulate the extra side effect (H3). ii) ***Constraints on constants.*** The range of constants that can be encoded in a v7m instruction is often narrower (Table 3.4, G2). In such cases, the amendment instructions load the constant to a scratch register, operate it, and emulate any side effects (e.g., index update) the guest instruction may have. iii) ***Richer shift modes.*** v7a instructions support richer shift modes and larger shift ranges than their v7m counterparts. This is exemplified by Table 3.4 G1, where a v7m instruction cannot perform LSR (logic shift right) inline as its v7a counterpart. Similar to above, the amendment instructions perform shift on the operand in a scratch register.

Beyond the above, only 27 v7a instructions have no v7m counterparts, for which we manually devise translation rules.

ARMv7a	ARMv7m (by ARK)
G1: `ldr r0, [r1], r2, lsr #4`	**H1:** `ldr.w r0, [r1]` **H2:** `lsr.w t0, r2, 0x4` **H3:** `add.w r1, r1, t0`
G2: `adds r0, r1, 0x80000001`	**H4:** `mov.w t0, 0xc0` **H5:** `ror.w t0, t0, 0x7` **H6:** `adds.w r0, r1, t0`
G3: `sub r0, r1, r2`	**H7:** `sub.w r0, r1, r2`

Table 3.4: Sample translation by ARK. By contrast, our baseline QEMU port translates G1–G3 to **27** v7m instructions

In summary, through systematic exploitation of similar instruction semantics, ARK emits compact host code at run time. In the example shown in Table 3.4, three v7a instructions are translated into seven v7m instructions by ARK, while to 27 instructions by our QEMU baseline.

3.5.2 Passthrough of CPU registers

General purpose registers Both the guest (v7a) and the host (v7m) have the same set (13) of general-purpose registers. In allocating registers of a host instruction, ARK follows guest register allocation with best efforts (e.g., one-to-one mapping in best case, as in Table 3.4, G1). ARK emits much fewer host instructions than QEMU, which emulates all guest registers in host memory with load /store.

Caveats fixed The amendment host instructions operate scratch registers as exemplified by `t0` in Table 3.4, H2-H6. However, the wimpy host faces higher register pressure, as it (v7m) has no more registers than the brawny guest (v7a). To spill some registers to memory while still reusing the guest's register allocation, we make the following tradeoff: we designate one host register as the *dedicated* scratch register, and emulates its guest counterpart register in memory. We pick the *least* used one in the guest binary as the dedicated scratch register, which is experimentally determined as R10 by analyzing kernel binary. We find most amendment instructions are satisfied by *one* scratch register; in rare cases when extra scratch registers are needed, ARK follows a common design to allocate dead registers and spill unused ones to memory.

Condition flags Both the guest and the host ISAs involve five hardware condition flags (e.g., zero and carry) with identical semantics; fortunately, most guest (v7a) instructions and their host (v7m) counterparts have identical behaviors in testing/setting flags per the ISA specifications [178]. ARK hence directly emits instructions to manipulate the host's corresponding flags. Such flag passthrough especially benefits control-heavy suspend/resume, which contains extensive conditional branches (§3.2); we study its benefits quantitatively in §3.7.3.

Caveats fixed Amendment host instructions may affect the hardware condition flags unexpectedly. For amendment instructions (notably comparison and testing) that *must* update the flags as mandated by ISA, ARK emits two host instructions to save/restore the flags in a scratch register around the execution of these amendment instructions.

3.5.3 Control Transfer and Stack Manipulation

Function call/return Both guest (v7a) and host (v7m) use PC (program counter) and LR (link register) to maintain the control flow. QEMU emulates guest PC and LR in host memory. As a result, the return address, loaded from stack or the emulated LR, points to a guest address (i.e., kernel address). Each function return hence causes the DBT to step in and look up the corresponding code cache address. This overhead is magnified in the control-heavy device phase.

By contrast, ARK never emits host code to emulate the guest (i.e., kernel) PC or LR. For each kernel function call, ARK saves the return addresses within *code cache* on stack or in LR; for each kernel function return, ARK loads the return address (which points to code cache) to hardware PC from the stack or the hardware LR. By doing so, ARK no longer participates in all function returns. Our optimization is inspired by same-ISA DBT [180].

Stack and SP QEMU emulates the guest (i.e., kernel) stack and SP with a host array and a variable. Each guest push/pop translates to multiple host instructions updating the stack array and the emulated SP. This is costly, as suspend/resume frequently makes function calls and operates stack heavily.

ARK avoids such expensive stack emulation by emitting host push/pop instructions to directly operate the guest stack *in place*. This is possible because ARK emulates the Linux kernel's virtual address space (§3.4). ARK also ensures the host code generate the same stack frames as the guest would do by making amendment instructions avoid using stack, which would introduce extra stack contents. In addition, this further facilitates the migration in abort (§3.6).

Caveats fixed i) As the host saves on the guest stack the code cache addresses, which are meaningless to the guest CPU, upon migrating from the peripheral core (host) to the CPU (guest), the DBT rewrites all code cache addresses on stack with their corresponding guest addresses. ii) guest push/pop instruction may involve emulated registers (i.e., scratch register). ARK must emit multiple host instructions to correctly synchronize the emulated registers in memory.

3.6 Translated $\longrightarrow$ Native Fallback

As described in Section 3.3, when going off the hot paths, ARK migrates the kernel phase back to the CPU and continues as native execution, analogous to virtual-to-physical migration of VMs [181]. Migrating one DBT context is natural, as ARK passes through most CPU registers and uses the kernel stack in place (§3.5.3). Yet, to migrate *all* active DBT contexts, ARK address the following unique challenges.

Migrate DBT contexts for deferred work After fallback, all blocked workitems should continue their execution on the CPU. Unfortunately, their enclosing DBT contexts do not have counterparts in the Linux kernel. To solve this issue, we again exploit the insight that the workitems are oblivious to their execution contexts. Upon migration, the Linux kernel creates temporary kernel threads as "receivers" for blocked workitems to execute in. Once the migrated workitems complete, the receiver threads terminate.

Migrate DBT context for interrupt If fallback happens inside an ISA-neutral interrupt handler (translated), the remainder of the handler should migrate to the CPU. This challenge, again, is that ARK's interrupt context has no counterpart on the CPU: the interrupt never occurs to the CPU. ARK addresses this by *rethrowing* the interrupt as an IPI (inter-processor interrupt) from the peripheral core to the CPU; the Linux kernel uses the IPI context as the receiver for the migrated interrupt handler to finish execution.

Section 3.7 will evaluate the fallback frequency and cost.

3.7 Evaluation

We seek to answer the following questions:

1. Does ARK incur tractable engineering efforts? (§3.7.2)
2. Is ARK correct and low-overhead? (§3.7.3)
3. Does ARK yield energy efficiency benefit? What are the major factors impacting the benefit? (§3.7.4)

3.7.1 Methodology

Test Platform We evaluate ARK on OMAP4460, an ARM-based SoC [148] as summarized in Table 3.6. We chose this SoC mainly for its good documentation and longtime kernel support (since 2.6.11), which allows our study of kernel ABI over a long timespan in Section 3.2. As Cortex-M3 on the platform is incapable of DVFS, for fair comparison, we run both cores at their highest clock rates. Note that OMAP4460 is not completely aligned with our hardware model, for which we apply workarounds as will be discussed in Section 3.7.5.

Benchmark setup We benchmark ARK on the whole suspend/resume kernel phases. We run a user program as the test harness that periodically kicks ARK for suspend/resume; the generated kernel workloads are the same as in all ephemeral tasks. Our benchmark is macro: it exercise extensive drivers and services, during which ARK translates and executes over 200 million instructions.

The benchmark operates nine devices for suspend/resume. 1. **SD card**: SanDisk Ultra 16GB SDHC1 Class 10 card; 2. **Flash drive**: a generic drive connected via USB; 3. **MMC controller**: on-chip OMAP HSMMC host controller; 4. **USB controller**: on-chip OMAP HS multiport USB host controller; 5. **Regulator**: TWL6030 power management IC connected via I2C; 6. **Keyboard**: Dell KB212-B keyboard connected via USB; 7. **Camera**: Logitech c270 connected via USB; 8. **Bluetooth NIC**: an adapter with Broadcom BCM20702 chipset connected via USB; 9. **WiFi NIC**: TI WL1251 module. The kernel invokes sophisticated drivers, thoroughly exercising various services including deferred work (2–4,6–8), slab/buddy allocator (1–4,6–9), softirq (9), DMA (2,6–9), threaded IRQ (1,5,9), and firmware upload (9).

We measure device suspend/resume executed by ARK on Cortex-M3 and report the measured results. We compare ARK to native Linux execution on Cortex-A9. We further compare to a baseline ARK version: its DBT is a straightforward v7m port of QEMU that misses optimizations described in Section 3.5. We report measurements taken with warm DBT code cache, as this reflects the real-world scenario where device suspend/resume is frequently exercised.

3.7.2 Analysis of engineering efforts

ARK eliminates source refactoring of the Linux kernel (§3.2.3). As shown in Table 3.5, ARK transparently reuses substantial kernel code (15K SLoC in our test), most of which are drivers and their libraries. We stress that ARK, as a driver-agnostic effort, not only enables reuse of the drivers under test but also other drivers in the ARMv7 Linux kernel.

Existing code (unchanged)	
Translated	15K SLoC
Substituted w/ emu	25K SLoC
New implementation	
DBT	9K SLoC
Emulation	1K SLoC

Table 3.5: Source code of ARK.

Table 3.5 also shows that ARK requires modest efforts in developing new software for the peripheral core. The 9K new SLoC for DBT is low as compared to commodity DBT (e.g., QEMU has 2M SLoC). ARK implements emulation in as low as 1K SLoC and in return avoids translating generic, sophisticated Linux kernel services [180, 182]. The result validates our principle of specializing these emulated services.

ARK meets our goal of "build once, run with many". We verify that the ARK binary works with a variety of kernel configuration variants (including `defconfig-omap4` and `yes-to-all`) of Linux 4.4. We also verify that ARK works with a wide range of Linux versions, from version 3.16 (2014) to 4.20 (most recent at the time of writing). This is because ARK only depends on a narrow ABI shown in Table 3.2, which has not changed since Linux 3.16.

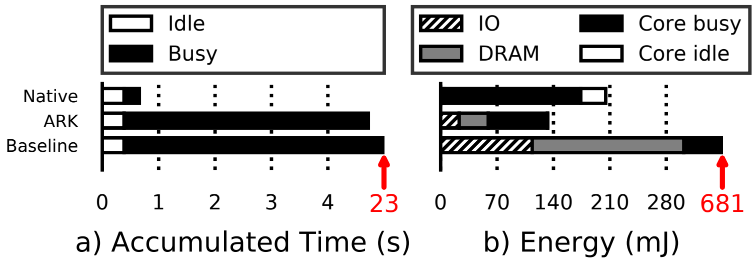


Figure 3.5: Execution time and energy in device suspend/resume. ARK substantially reduces the energy.

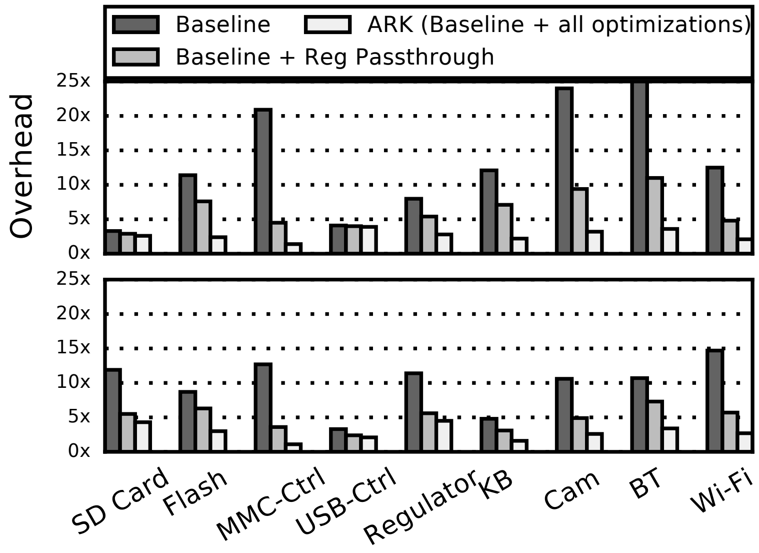


Figure 3.6: Busy execution overhead for devices under test (top: suspend; bottom: resume). Our DBT optimizations reduce the overhead by up to one order of magnitude

3.7.3 Measured execution characteristics

ARK's correctness Formally, we derive translation rules from the specification of ARM ISA [178]; experimentally, we validate ARK by comparing its execution results side-by-side with native execution and examining the translated code with the native kernel binary. Over 200 million executed instructions, we verify that ARK's translation preserves kernel's semantics and presents consistent execution results.

Core activity We trace core states during ARK execution. Figure 3.5 (a) shows the breakdown of execution time. Compared to the native execution on CPU, ARK shows the same amount of accumulated idle time but much longer (16×) busy time. The reasons are Cortex-M3's much lower clock rate (1/6 of the A9's clock rate) and ARK's execution overhead. Despite the extended busy time, ARK still yields energy benefit, as we will show below.

Memory activity We collect DRAM activities by sampling the hardware counters of the SoC's DDR controller. We observed that ARK on Cortex-M3 generates much higher average DRAM utilization (32 MB/s

	CPU	Peripheral core
Core	Cortex A9@1.2GHz	Cortex M3@200MHz
Cache	L1:64KB + L2:1MB	L1:32KB
Typical busy/idle power	630mW/80mW	17mW/1mW

Table 3.6: The test platform - OMAP4460 on a Pandaboard

read and 2MB/s write) than the native execution on A9 (only 8MB/s read and 4MB/s write). We attribute such thrashing to M3's small (32KB) last-level cache (LLC). Throughout the test, the ARK emitted and executed around 230KB host instructions, which far exceeds the LLC capacity and likely causes thrashing. By contrast, Cortex-A9 has a much larger LLC (1MB), which absorbs most of the kernel memory access. The memory activity has a strong energy impact, as will be shown below.

Busy execution overhead Our measurement shows that ARK incurs low overhead in busy kernel execution, which includes both DBT and emulation. We report the overhead as the ratio between ARK's cycle count on Cortex-M3 to the Linux's cycle count on A9. Note that an M3 cycle is 6× longer than A9 due to different clock rates.

Overall, the execution overhead is 2.7× on average (suspend: 2.9×; resume: 2.6×). Of individual drivers, the execution overhead ranges from 1.1× to 4.5× as shown in Figure 3.6. Our DBT optimizations (§3.5) have strong impact on lowering the overhead. Lacking them, our baseline design incurs a 13.9× overhead on average, 5.2× higher than ARK. We examined how our optimizations contribute to the gap: register passthrough (§3.5.2) reduces the baseline's overhead by 2.5× to 5.5×. Remaining optimizations (e.g., control transfer) collectively reduce the overhead by additional 2×. Our optimizations are less effective on drivers with very dense control transfer (e.g., USB) due to high DBT cost.

Emulated services Our profiling shows that ARK's emulated services incur low overhead. Overall, the emulated services only contribute 1% of total busy execution. i) The early, core-specific interrupt handling (§3.4.2) takes 3.9K Cortex-M3 cycles, only 1.5–2× more cycles than the native execution on A9. ii) Emulated workqueues (§3.4.3) incurs a typical queueing delay of tens of thousands M3 cycles. The delay is longer than the native execution but does not break the deferred execution semantics.

Fallback frequency & cost We stress test ARK by repeating the benchmark for 1000 runs. Throughout the 1000 runs, ARK encounters only four cases when the execution goes off the hot path, all of which caused by the WiFi hardware failing to respond to resume commands; it is likely due to an existing glitch in WiFi firmware. In such a case, ARK migrates execution by spending around 20 us on rewriting code cache addresses on stack (§3.5.3), 17 us to flush Cortex-M3's cache, and 2 us to wake up the CPU through an IPI.

3.7.4 Energy benefits

Methodology We study system-level energy and in particular how it is affected by ARK's its extended execution time. We include energy of both cores, DRAM, and IO.

We measure power of cores by sampling the inductors on the power rails for the CPU and the peripheral core. As the board lacks measurement points for DRAM power [183], we model DRAM power as a function of DRAM power state and read/write activities, with Micron's official power model for LPDDR2 [184]. The system energy of ARK is given by:

$$E_{ARK} = \underbrace{E_{core}}_{\text{Measured}} + T_{idle} \cdot \underbrace{(P_{mem\_sr} + P_{io})}_{Modeled} + T_{busy} \cdot \underbrace{(P_{mem} + P_{io})}_{Modeled}$$

Here, E_{core} is the measured core energy. All Ts are measured execution time. Ps are power consumptions for DRAM and IO: P_{mem} is DRAM's active power derived from measured DRAM activities as described in Section 3.7.3; P_{mem_sr} is DRAM's self-refresh power, 1.3mW according to the Micron model; P_{io} is the average IO power which we estimate as 5mW based on prior work [117]. Note that during suspend/resume, IO devices no longer actively perform work, thus consuming much less power.

Energy saving ARK consumes 66% energy (a reduction of 34%) of the native execution, despite its longer execution time. The energy breakdown in Figure 3.5(b) shows the benefit comes from two portions: i) in busy execution, ARK's energy efficiency is 23% higher than the native execution due to low overhead (on average 2.7×); ii) during system idle, ARK reduces system energy to a negligible portion, as the peripheral core's idle power is only 1.25% of the CPU's. Figure 3.5(b) highlights the significance of our DBT optimizations: the baseline, like ARK, benefits from lower idle power as well; however its high execution overhead ultimately leads to 5.1× energy compared to the native execution. Interestingly, ARK consumes *more* DRAM energy than the native execution. We deem the cause as Cortex-M3's tiny LLC (32KB) as describe earlier. Our result suggests that the current size is suboptimal for the offloaded kernel execution.

What-if analysis How sensitive is ARK's energy saving to two major factors: the DBT overhead (ARK's behavior) and the processor core usage (Linux's behavior)? To answer the question, we estimate the *what-if* energy consumption by using the power model as described above. The analysis results in Figure 3.7 show two findings. i) ARK's energy benefit will be more pronounced with lower core usage (i.e., longer core idle), because ARK's efficiency advantage over native execution is higher during core idle. ii) ARK's energy benefit critically depends on DBT. When the DBT overhead (on x-axis) drops to below 3.5×, ARK saves energy even for 100% busy execution; when the overhead exceeds 5.2×, ARK wastes energy even for 20% busy execution, the lowest core usage observed on embedded platforms in prior work [12].

Qualitative comparison with big.LITTLE We estimate ARK saves tangible energy compared to a

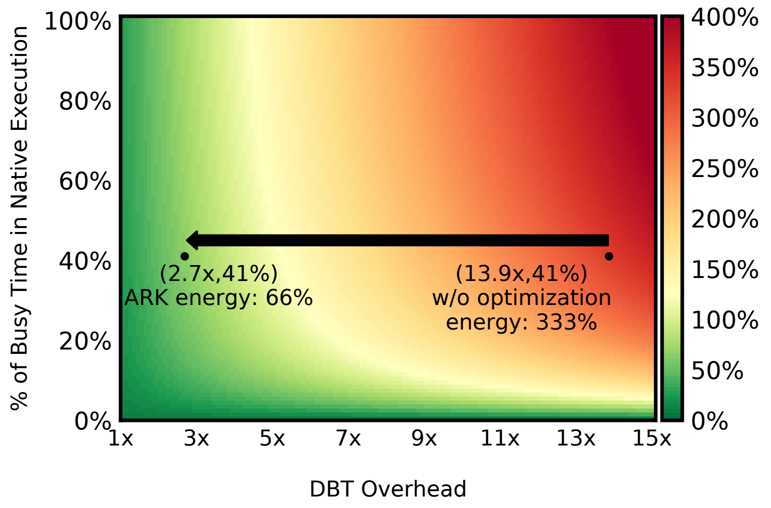


Figure 3.7: System energy consumption (inc. cores, DRAM, and IO) of ARK relative to native execution (100%), under different DBT overheads (x-axis) and processor core usage (y-axis). ARK's low energy hinges on low DBT overhead.

LITTLE core. We use parameters based on recent big.LITTLE characterizations [185, 186]: compared to the big (i.e., CPU on our platform), a LITTLE core has 40 mW idle power [187] and offers 1.3× energy efficiency at 70% clock rate [188]. We favorably assume LITTLE's DRAM utilization is as low as the big, while in reality the utilization should be higher due to LITTLE's smaller LLC. Even with this favorable assumption for LITTLE and unfavorable, tiny LLC for ARK, LITTLE consumes 77% energy of native execution, more than ARK (51%–66%), mainly because LITTLE's idle power is 40× of Cortex-M3. Furthermore, ARK's advantage will be even more pronounced with a proper LLC as discussed earlier.

Battery life extension Based on ARK's energy reduction in device suspend/resume, we project the battery life extension for ephemeral tasks reported in prior work [122]. When the ephemeral tasks are executed at 5-second intervals and the native device suspend/resume consumes 90% system energy in a wakeup cycle, ARK extends the battery life by 18% (4.3 hours per day); with 30-second task intervals and a 50% energy consumption percentage, ARK extends the battery life by 7% (1.6 hours per day). This extension is tangible compared to complementary approaches in prior work [122, 117].

3.7.5 Discussions

Workarounds for OMAP4460 While OMAP4460 mostly matches our hardware model as summarized in Table 3.1, for minor mismatch we apply the following workarounds. ***Memory mapping*** Our hardware model (§3.2.2) mandates that the peripheral core should address the entire kernel memory. Yet, Cortex-M3, according to ARM's hardware specification [6], is only able to address memory in certain range (up to `0xE0000000`), which unfortunately does not encompass the Linux kernel's default address range. As a workaround, we

ARMv8	ARMv7m (by ARK, ideally)

G1:
 	
`ldrb w2, [x22, #1059]`
 	
`ldrb w1, [x0, #160]`
	H1:
	`(emulate x22+#1059 in addr1)`
	`ldrb r2, [addr1]`
	`(emulate x0+#160 in addr2)`
	`ldrb r1, [addr2]`

G2: `cmp w2, w1`	**H2:** `cmp r2, r1`
G3: `beq mmc_select_bus_width+0x160`	**H3:** `beq mmc_select_bus_width+0x160`

Table 3.7: Ideal AARCH64 translation by ARK for `mmc_compare_ext_csds()` in Linux v4.4. While identity mapping still exists (G2→H2), software emulation can diminish ARK's benefits (G1→H1).

configure the Linux kernel source, shifting its address range to be addressable by Cortex-M3. ***Interrupt handling*** While our hardware model mandates that both processors should receive all interrupts, OMAP4460 only routes a subset of them (39/102) to Cortex-M3, leaving out IO devices such as certain GPIO pins. These IO devices hence are unsupported by the ARK prototype and are not tested in our evaluation.

Recommendation to SoC architects To make SoCs friendly to a transkernel, architects may consider: i) routing all interrupts to CPU and the peripheral core, ideally with the identical interrupt line numbers; ii) making the peripheral core capable of addressing the whole memory address space; iii) enlarging the peripheral core's LLC size modestly. We expect a careful increase (e.g., to 64 KB or 128 KB) will significantly reduce DRAM power at a moderate overhead in the core power.

Applicability to a pair of 64-bit/32-bit ISAs While today's smart devices often use ARMv7 CPUs, emerging ARM SoCs start to combine 64-bit CPUs (ARMv8) with 32-bit peripheral core (ARMv7m), as listed in Table 3.1. On one hand, transkernel's idea of exploiting ISA similarity still applies, as exemplified by G2→H2 in Table 3.7; on the other hand, its DBT overhead may increase significantly for the following reasons. Compared to the 32-bit ISA, the 64-bit ISA has richer instruction semantics, more general purpose registers, and a much larger address space. As a result, ARK cannot pass through 64-bit CPU registers but instead have to emulate them in memory; ARK must translate the guest's 64-bit memory addresses to 32-bit addresses supported by the host (Table 3.7 G1→H1), e.g., by keeping consistent two sets of page tables, for 64-bit and 32-bit virtual address spaces, respectively; with large physical memory (¿4GB), even this technique will not work because the peripheral core's page tables are incapable of mapping the extra physical memory.

3.8 Related Work

OS for heterogeneous cores A multikernel OS runs its kernels on individual processors. A number of such OSes are designed anew with a strong distributed system flavor. They define explicit message interfaces among kernels [139, 189, 190]; some additionally exploit managed languages/compilers to generate such interfaces [138]. Unlike them, transkernel targets spanning an *existing* monolithic kernel and therefore adopts DBT to address the resultant interface challenge.

OSes like Popcorn [137] and K2 [133] seek to present a single Linux image over heterogeneous processors. For sharing kernel state across ISAs, they rely on manual source tweaks or hand-crafted communication. They face the interface difficulty as described in §3.2.3.

Prior systems distribute OS functions over CPU and accelerators [191, 192]. The accelerators cannot operate autonomously, which is however required by device suspend/resume. Prior systems offload apps from a smartphone (weak) to cloud servers (strong) for efficiency [40, 41]. Unlike them, transkernel offloads kernel workloads from a strong processor to a weak peripheral core on the same chip.

DBT DBT has been used for system emulation [170] and binary instrumentation [180, 177, 176, 182]; DeVuyst et al. [193] uses DBT to speed up process migration. Related to transkernel, prior systems run translated user programs atop an emulated syscall interface [170, 194, 169]. Unlike them, transkernel translates kernel code and emulates a narrow interface *inside* the kernel. Prior systems use DBT to run binaries in commodity ISAs (e.g., x86) on specialized VLIW cores and hence gain efficiency [195, 196, 197, 198]. None runs on microcontrollers to our knowledge. transkernel demonstrates that DBT can gain efficiency even on off-the-shelf cores. Existing DBT engines leverage ISA similarities, e.g., between aarch32 and aarch64 [168, 199]. They still fall into the classic DBT paradigm, where the host ISA is brawny and the guest ISA is wimpy (i.e., lower register pressure). With an inverse DBT paradigm, ARK addresses very different challenges. Much work is done on optimizing DBT translation rules, using optimizers [200, 201] or machine learning[179]. Compared to them, ARK leverages ISA similarities and hence reuses code optimization already in guest code by guest compilers.

Kernels and drivers The transkernel is inspired by the POSIX emulator [202] however is different as it emulates kernel ABIs. Prior kernel studies show rapid evolution of the Linux kernel and the interfaces between kernel layers are unstable [136, 203]. This observation motivates transkernel. Extensive work transplants device drivers to a separate core [160], user space [161], or a separate VM [204]. However, the transplant code cannot operate independent of the kernel, whereas transkernel must execute autonomously.

Encapsulating the NetBSD kernel subsystems (e.g., drivers) behind stable interfaces respected by developers, rump kernel [205] seeks to enable their reuse in foreign environments, e.g., hypervisors. The transkernel targets a different goal: spanning a live Linux kernel instance over heterogeneous processors. Applying Rump kernel's approach to Linux is difficult, as Linux intentionally rejects interface stability for drivers [162].

Suspend/resume's inefficiency raises attention for cloud servers [145, 206] and mobile [122]. Drowsy [122] mitigates inefficiency by reducing the devices involved in suspend/resume through user/kernel co-design; Xi *et* *al.* propose to reorder devices to resume [206]. While acknowledging the value of such kernel optimizations, we believe ARK is a key complement that works on unmodified binaries. ARK can co-exist with the mentioned optimizations in the same kernel. PowerNap [145] takes a hardware approach to speed up suspend/resume for servers. It does not treat kernel execution for operating diverse IO on embedded platforms. Kernels may put idle devices to low power at runtime [117], complementary to suspend/resume that ensures all devices are off.

3.9 Conclusions

We present transkernel, a new executor model for a peripheral core to execute a commodity kernel's phases, notably device suspend/resume. The transkernel executes the kernel binary through cross-ISA DBT. It translates stateful code while emulating stateless services; it picks a stable ABI for emulation; it specializes for hot paths; it exploits ISA similarities for DBT. We experimentally demonstrate that the approach is feasible and beneficial. The transkernel represents a new OS design point for harnessing heterogeneous SoCs.

Chapter 4

Turbocharge Interactive NLP at the Edge

4.1 Introduction

Natural Language Processing (NLP) is seeing increasing adoption by mobile applications [207]. For instance, a note-taking app allows users to verbally query for old notes and dictate new notes. Under the hood, the app invokes an NLP model in order to infer on user input. It is often desirable to execute NLP inference *on device*, which crucially preserves user data privacy and eliminates long network trips to the cloud [208, 209].

NLP inference stresses mobile devices on two aspects. (1) Impromptu user engagements. Each engagement comprises a few turns [210]; users expect short delays of no more than several hundred ms each turn [211], often mandated as target latencies [208]. (2) Large model size. Designed to be over-parameterized [212, 213], today's NLP models are hundred MBs each [214, 215, 216], much larger than most vision models [217, 218]. As common practice, separate NLP model instances are fine-tuned for tasks and topics, e.g. one instance for sentiment classification [219] and one for sequence tagging [220], which further increase the total parameter size on a mobile device.

How to execute NLP models? There are a few common approaches (Figure 4.1). (1) *Hold in memory*: preloading a model before user engagement or making the model linger in memory after engagement. The efficacy is limited: a model in memory increases one app's memory footprint (often less than 100MB [221, 222]) by a few times, making the app a highly likely victim of the mobile OS's low memory killer [223]; as user engagements are bursty and each consists of as few as 1-3 model executions [210], a lingering model likely benefits no more than 2 executions before its large memory is reclaimed by the OS; since user engagements

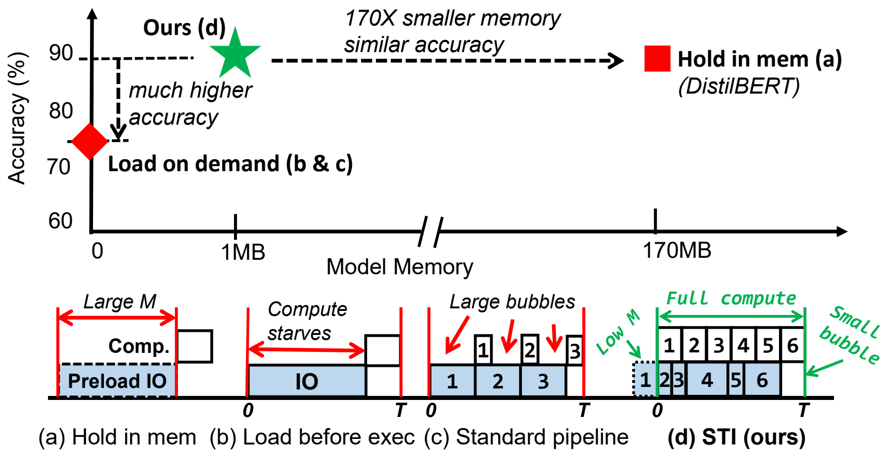


Figure 4.1: Comparison of model execution methods. Our method achieves high accuracy at low memory cost. T: target latency. M: model memory for Transformer weights.

are impromptu [224, 225], predicting when to preload/unload models is challenging. (2) *Load on demand.* The problem is the long IO delays for loading a NLP model. For instance, DistilBERT, a popular model optimized for mobile, takes 2.1 seconds to load its 170 MB parameters as we measured, far exceeding user desirable latencies of several hundred ms. To hide IO delays, one may stream model parameters from storage to memory during computation: execute model layer k while loading parameters for layer $k+1$. While such a IO/compute pipeline was known in ML [226, 227], directly applying it to NLP inference is ineffective: the core parts of NLP models such as attention has a skewed IO/compute ratio due to low arithmetic intensity [228]. As a result, most of the time (¿72%) the computation is stalling.

These approaches suffer from common drawbacks: (1) key resources – memory for preload and IO/compute for model execution – are managed in isolation and lack coordination; (2) obliviousness to a model's parameter importance, i.e. which parameters matter more to model accuracy. Hence, the preload buffer unnecessarily holds parameters that could have been streamed in parallel to execution; IO unnecessarily loads parameters that the computation cannot consume within the target latency. The results are memory waste, frequent pipeline stalls, and inferior model accuracy due to low FLOPs.

Our design We present an engine called STI. Addressing the drawbacks above, STI integrates on-demand model loading with lightweight preload, getting the best of both approaches.

(1) A model as resource-elastic shards. The engine preprocesses an N-layer model: partitioning each layer into M shards; compressing each shard as K fidelity versions, each version with a different parameter bitwidth. The engine therefore stores the $N \times M \times K$ shard versions on flash. At run time, the engine assembles a *submodel* of its choice: a subset of n layers ($n <= N$); m shards ($m <= M$) from each selected layer; a fidelity version for each selected shard. *Any such submodel can yield meaningful inference results*, albeit with different

accuracies and resource costs. Our model sharding is a new combination of existing ML techniques [229, 230].

In this way, the engine can dynamically vary a model's total execution time, adjust IO/compute ratios for individual shards, and allocate IO bandwidth by prioritizing important shards.

(2) Preload shards for warming up pipeline. The engine maintains a small buffer of preload shards, adjusting the size to available memory. Instead of trying to hold the entire model, it selectively holds shards from a model's bottom layers (closer to input). Upon user engagement, the engine can start executing the early stage of a pipeline with much of the parameters already loaded, which otherwise would have to stall for IO.

(3) A joint planner for memory, IO, and computation. The engine's planner selects shards and their versions to preload and to execute. Its goal is to compose a submodel that simultaneously meets the target latency, minimizes pipeline stalling, and maximizes accuracy.

Towards this goal, our ideas are (1) set layerwise IO budgets according to layerwise computation delays and (2) allocate IO budgets according to shard importance. To plan, STI first decides a submodel that can be computed under the target latency. The engine then sets *accumulated IO budgets* (AIBs) at each layer to be the computation delays of all prior layers; it further treats the available memory for preload shards as *bonus* IO budgets to all layers. Having set the budgets, the engine iterates over all shards, allocating extra bitwidths to loading important shards and hence debiting IO budgets of respective layers. The engine preloads the first k shards in the layer order that maximize the usage of preload memory size $|S|$ but not exceeding $|S|$.

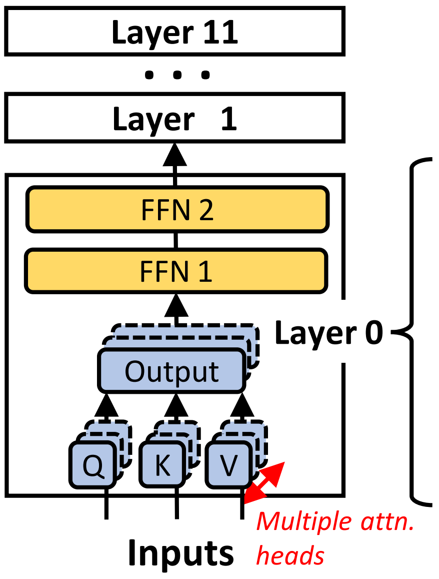
Results We implement STI atop PyTorch and demonstrate it on mobile CPU and GPU of two embedded platforms. On a diverse set of NLP tasks, STI meet target latencies of a few hundred ms while yielding accuracy comparable to the state of the art. We compare STI against competitive baselines enhanced with recent ML techniques [229, 230] as illustrated in Figure 4.1. Compared to holding a model in memory, STI reduces parameter memory by 1-2 orders of magnitude to 1–5MB, while only seeing accuracy drop no more than 0.1 percentage points; compared to existing execution pipelines, STI increases accuracy by 5.9-54.1 percentage points as its elastic pipeline maximizes both compute and IO utilization.

Contributions The paper makes the following contributions:

- Model sharding, allowing the engine to fine control an NLP model's total computation time and finetune each shard's IO time according to resource constraints and shard importance.

- A pipeline with high IO/compute utilization: a small preload buffer for warming up the pipeline; elastic IO and computation jointly tuned to minimize pipeline bubbles and maximize model accuracy.

- A two-stage planner for the pipeline: picking a submodel, tracking layerwise IO budgets, and prioritizing importance shards in resource allocation.



Param. breakdown (one BERT layer)

Module	# of param.
FFN2	2.36M
FFN1	2.36M
Output	590K
Value	590K
Key	590K
Query	590K
Total #	7.08M

Figure 4.2: (Left) The BERT model comprising transformer layers and (Right) the number of 32-bit floating point parameters within a layer [231].

4.2 Motivations

4.2.1 Transformer on mobile devices

A primer on transformer Figure 4.2 shows the architecture of Transformer [231], the modern NN developed for NLP tasks. Compared with traditional NNs (e.g. LSTM [232]), it features a unique Multi-Headed Attention (MHA) mechanism. MHA extracts features at sequence dimension by modeling pairwise word interactions through many *attention heads* (typically 12), which are backed by three fully-connected (i.e. linear) layers, namely Query (Q), Key (K), Value (V). Given an input, each attention head independently contributes an attention score as one representation of the feature space. Scores across attention heads are concatenated via a linear output layer (O) and then projected into higher feature dimensions by two linear layers in the point-wise Feed-Forward Network (FFN) module.

Due to the large number of fully connected layers, a transformer based model contains over 100 million parameters. As a result, a typical pretrained model is of a few hundred MBs. For instance, BERT [214] as one of the most popular model is over 400MB large.

Resource demands (1) *Low latencies.* Prior studies show that users expect mobile devices to respond in several hundred milliseconds, and their satisfaction quickly drops as latency grows beyond around 400ms [233]. (2) *Large model parameters.* The scale of NLP parameters is unprecedented for on-device machine learning. Even DistilBERT [215] optimized for mobile has nearly 200MB of parameters, contrasting to popular vision models which are as small as a few MBs [217, 218]. Such numerous parameters stress both memory capacity and IO for loading them.

Besides parameters, model execution also allocates memory for intermediate results. Yet, such data has short lifespans and does entail loading from storage. Hence, it can be served with a relatively small working buffer sufficient to hold a model tile (often a few MBs); the size do not grow with the model size. We therefore do not optimize for it.

4.2.2 Transformers challenge existing paradigms

Existing paradigms are inadequate, as shown in Figure 4.1.

First, hold in memory. An app may keep model files lingering in memory or even *pin* them; thus, the model can start execution anytime without IO delays. For how long the app holds the model depends on its prediction of future user engagements.

The major drawback is that an in-memory model will take hundreds of MBs of memory, bloating an app's memory footprint which is often less than 100 MBs [221, 222]. When an app's memory footprint is much larger than its peers, it becomes a highly likely victim of mobile memory management, which aggressively kills memory-hungry apps [221]. Once killed, the app has to reload the model for the next engagement. Furthermore, precise prediction of user engagement is difficult, as mobile apps often exhibit sporadic and ad hoc usage [8, 211]. To exacerbate the problem, co-running apps may invoke separate models for their respective tasks, e.g. for sentiment analysis and for next-word prediction.

Second, load before execute. As the default approach by popular ML frameworks [234, 235]: upon user engagement, the app sequentially loads the model and executes it. As we measured on a modern hexa-core Arm board (see Table 4.2), it takes 3.6 seconds to execute DistilBERT, among which 3.1 seconds are for loading the whole 240 MB model file. Prior work observed similar symptoms of slow start of model inference [236, 226].

Third, pipelined load/execution. To hide IO delays, one may leverage layerwise execution of ML models [237, 238] and overlap the layer loading IO and execution [226, 227]. This approach is barely effective for on-device NLP due to the high skewness between IO delays and computation delays. As we measured, a layer in DistilBERT requires 339 ms for parameter load while only 95 ms to compute. The root causes are (1) low arithmetic intensity in Transformer's attention modules [239] and (2) mobile device's efficiency-optimized flash, which limits the rate of streaming parameters from storage to memory. As a result, the pipeline is filled with bubbles and the computation stalls most of the time at each model layer.

Section 4.7 will compare our system against these approaches.

4.2.3 Model compression is inadequate

For efficient NLP inference, a popular category of techniques is model compression, including pruning networks (e.g. layers [215] and attention heads [240]), reducing feature dimensions [241], and sharing weights across layers [242]. A notable example is DistilBERT [215]: through distilling knowledge, it prunes half of BERT's layers, shrinking the model by 2×.

Still, model compression *alone* is inadequate. (1) While one may compress a model to be sufficiently small (e.g. ∼10MBs [243]) so that the load delay or the memory footprint is no longer a concern, the resultant accuracy is inferior, often unusable [244]. (2) The execution pipeline's bubbles still exist: compression often scales model compute and parameters *in tandem*, without correcting the computation/IO skewness. Hence, compute is still being wasted. (3) Most compression schemes lack flexibility as needed to accommodate diverse mobile CPU, GPU, and IO speeds. They either fix a compression ratio or require model re-training to adjust the ratios, which must done by the cloud for each mobile device.

Section 4.7 will evaluate the impact of model compression.

4.3 Design overview

4.3.1 The system model

STI incarnates as an library linked to individual apps. For complete NLP experience, we assume that the app incorporates other components such as automatic speech recognition (ASR), word embedding, and speech synthesis [245, 246, 247, 248]. As they often run much faster than model execution and are orthogonal to STI, this paper does not optimize for them.

STI loads and executes a model by layer: it loads one layer (comprising multiple shards) as a single IO job, decompresses all the shards in memory, and computes with the layer as a single compute job. IO and compute jobs of different layers can overlap. STI does not use smaller grains (e.g. load/execute each *shard*) as they leave the IO and GPU bandwidth underutilized, resulting in inferior performance.

STI allocates two types of memory buffers.

- *Preload buffer* holds shards preloaded selectively. STI keeps the buffer as long as the app is alive. STI can dynamically change the buffer size as demanded by the app or the OS.
- *Working buffer* holds a layer's worth of intermediate results and uncompressed parameters. The buffer is temporary, allocated before each execution and freed afterward. The buffer size is largely constant, not growing with the model size; it is not a focus of STI.

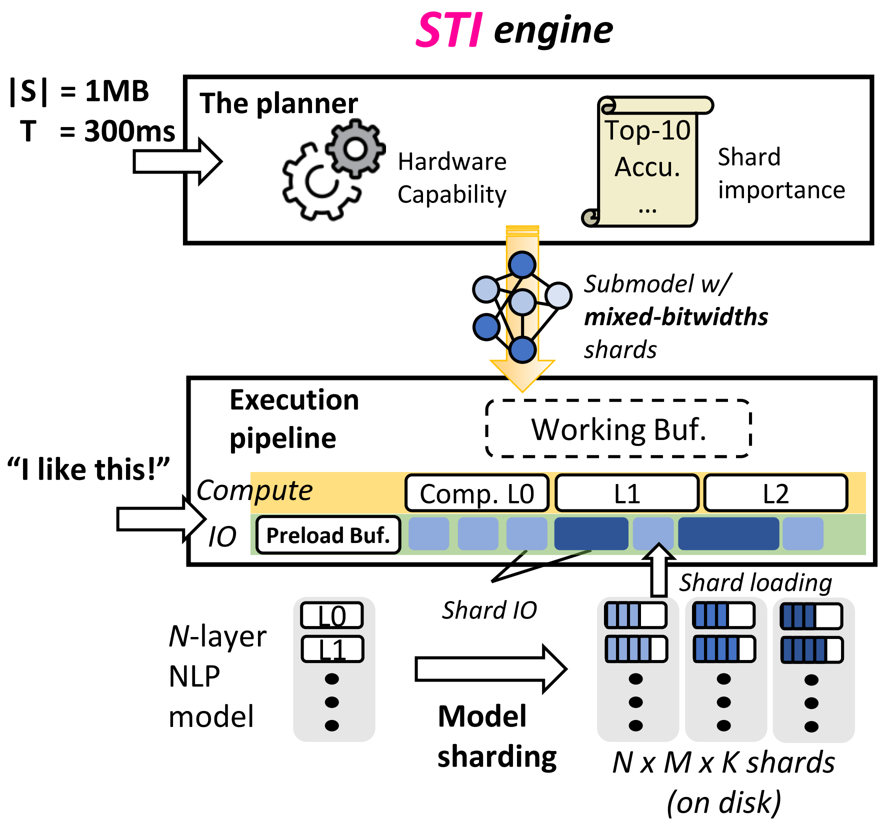


Figure 4.3: System architecture of **S**peedy **T**ransformer **I**nference (STI) and workflow.

4.3.2 The operation

The STI architecture is shown in Figure 4.3. STI preprocesses a given language model (e.g. DistilBERT finetuned for sentiment analysis): decomposing the model into shards and profiling shard importance (Section 4.5). As a one-time, per-model effort, the preprocessing is expected to be done in the cloud prior to model deployment to mobile devices; as preprocessing only requires lightweight model transformation (as opposed to expensive re-training [249]), it can be done on device as needed. The resultant model shards are stored alongside apps.

STI profiles each device's hardware once. The goal is to measure IO and computation delays in executing a language model; the profiling results serve as the basis for pipeline planning. To do so, STI loads and executes a Transformer layer in different bitwidths.

As an app launches, STI is initialized as part of the app. The app specifies which NLP model(s) it expects to execute, as well as the corresponding target latencies Ts and preload buffer sizes $|S|$s. Later, the app can update Ts and $|S|$s at any time. For each expected model, STI plans a separate execution pipeline with separate preload model shards. STI plans a pipeline once and executes it repeatedly. Replanning is necessary only when a model's T or $|S|$ is changed by the app or OS.

Upon user engagement, STI executes a pipeline for the requested model. Since planning is already done beforehand, STI simply loads and executes the shards that have been selected in planning.

4.3.3 Example execution scenarios

One-shot execution In this scenario, a user engagement consists of one turn, executing the model once. With preloaded shards, STI executes the pipeline without stalling in bottom layers close to input. STI uses the working buffer during the execution and frees it right after. Throughout the execution, the content of preload buffer is unchanged.

A few back-to-back executions One engagement may comprise multiple executions (often no more than 3) [210]. The scenarios is similar to above except for the opportunity of caching already loaded shards between executions. To this end, the app may request to enlarge the preload buffer so it selectively caches the loaded shards. In subsequent executions, STI no longer reloads these shards; its planner redistributes the freed IO bandwidth to other shards (Section 4.5), loading their higher-fidelity versions for better accuracy. After the series of executions, the app may choose to keep the additional cached shards as permitted by the OS or simply discard them.

4.3.4 Applicability

STI supports Transformer-based models [250, 229, 249]. This paper focuses on classification tasks (BERT and its variants), which underpin today's on-device NLP. Although STI's key ideas apply to *generative* models such as GPT-2 [216], their wide adoption on mobile (in lieu of template-based responses [251]) is yet to be seen; we consider them as future work.

STI keeps a model's execution time under a target latency T. However, it *alone* is insufficient to keep the *total wall-clock time* under T. Such a guarantee would require additional OS support, e.g. real-time scheduling. Towards such a guarantee, STI lays the foundation.

STI expects a small preload buffer. It can, however, work without such a buffer (i.e. "cold start" every time), for which its elastic sharding and pipeline still offer significant benefits as we will show in Section 4.7.

On future hardware/workloads, we expect STI's benefit to be more pronounced: mobile compute continues to scale (due to advances in technology nodes and accelerators); users expect results in higher accuracy; NLP models are becoming larger. All these lead to higher computation/IO skewness, necessitating an elastic pipeline of loading and execution.

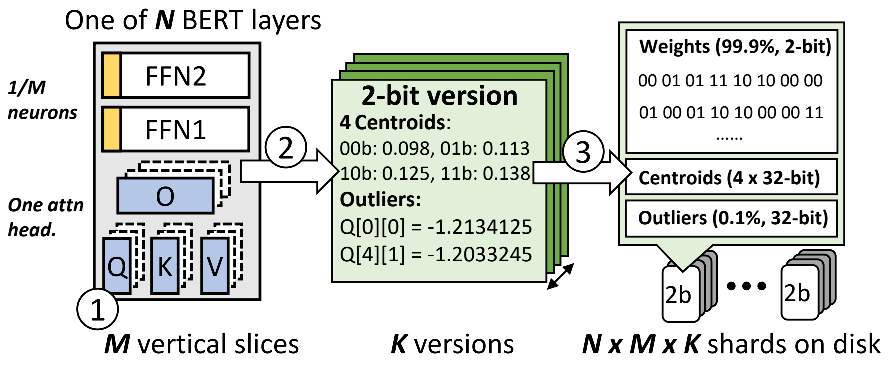


Figure 4.4: Instantiating $N \times M \times K$ model shards on disk. The example shows a 2-bit shard. 99.9% of its weights are represented by 2-bit indexes pointing to 2^2 centroids; the rest 0.1% outliers are preserved as-is.

4.4 Elastic model sharding

4.4.1 Key challenges

We solve a key challenge: how to partition the model into individual shards? Set to enable the resource elasticity of a model (i.e. depths/widths/fidelity), the shards must meet the following criteria:

• *Elastic execution.* Shards must preserve the same expressiveness of the attention mechanism and can execute partially to produce meaningful results.

• *Tunable IO.* The IO delays of shards must be tunable to accommodate IO/compute capability of different hardware (e.g. due to diverse CPU/GPUs and DVFS).

4.4.2 Instantiating model shards on disk

To address the challenges, our key idea is to combine two machine learning techniques – dynamic transformer [229, 250] and dictionary-based quantization [252], in a novel way. We next describe details.

First, vertical partitioning per layer The system adopts a pretrained transformer model, which has already been fine-tuned on a downstream task.

For each of the N layers, the system partitions it into M vertical slices, as shown in Figure 4.4 (①). By construction, each vertical slice is independent, constituting one attention head plus $1/M$ of FFN neurons of the layer; the partitioning is similar to a dynamic transformer [250, 229]. Table 4.1 shows the weight compositions of a vertical slice. Each cell of the table describes the dimension of the weight matrix, where d is hidden state size, M is the number of attention heads, and d_{ff} is the number of FFN neurons; a shard is therefore one of the M equal slices of a layer. Doing so warrants model shards the same capability to extract linguistic features from inputs, as done by the attention mechanism: of an individual shard, its attention head obtains one independent representation of input tokens, which is projected into a higher feature dimension by

	Attn (Q,K,V,O)	FFN1	FFN2
Transformer Layer	$d \times d$	$d_{ff} \times d$	$d \times d_{ff}$
Shard (vertical slice)	$d \times \frac{d}{M}$	$\frac{d_{ff}}{M} \times d$	$d \times \frac{d_{ff}}{M}$

Table 4.1: The weight composition of a shard. M is number of attention heads. The M shards equally slices a transformer layer, where each shard of the layer can be uniquely identified by its vertical slice index $i = 0 \dots M-1$.

FFN neurons [253, 254]; jointly, multiple shards attend to information from different representation subspace at different positions [231]. Therefore, an arbitrary subset of shards of a layer can be executed and still give meaningful results.

STI uses the *submodel* to describe the transformer model on shards, e.g. a $n \times m$ submodel comprises n layers, each layer having m shards. The number m is the same across all layers, as mandated by the transformer architecture [231], which specifies each layer must have the same width (i.e. number of shards m) for aligning input/output features between layers. Although it is possible for a shard to use 0s as placeholder weights, STI expects all m shards to have concrete weights for a good accuracy.

Second, quantization per shard The system compresses each of the $N \times M$ shards into K bitwidths versions (e.g. $K = 2 \dots 6$). STI is the first to bring quantization to *shard* granularity, whereas prior work only explores layer granularity [255, 256, 257]. Doing so reduces IO/compute skewness and facilitates elastic IO, allowing STI to prioritize IO resources at a much finer granularity, e.g. by allocating higher bitwidths to more important shards, and catering to IO/compute capability of diverse devices.

To compress, STI uses Gaussian outlier-aware quantization [230]. The key idea is to represent the vast majority of weights (e.g. 99.9%) which follow a Gaussin distribution using 2^k floating point numbers called *centroids*, hence compressing the original weights into k-bit indexes pointing to centroids. For the very few *outliers* (e.g. 0.1%) which do not follow the Gaussian distribution, it preserves their weights as-is. The process is shown in Figure 4.4 (②).

We choose it for two main reasons. 1) It provides good compatibility between shards of different bitwidths, allowing STI to tune their bitwidth individually per their importance and to assemble a *mixed-bitwidth* submodel. This is due to its lossy compression nature – shards still preserve the original distribution of layer weights, albeit in different fidelities. Hence they can work with each other seamlessly. 2) It does not need to fine-tune a model or require additional hardware support. The quantization analyzes the weight distribution of the pretrained model and is not specific to network structures; it hence does not require fine-tuning, as opposed to fixed-point quantization [258, 243]. The resultant *mixed-bitwidth* submodel also differs from a traditional *mixed-precision* network [257, 256, 255], which requires DSP extensions for executing integer operations efficiently; the extensions are often exclusive to microcontrollers on ARM devices, e.g.

Cortex-M4 [259].

Quantized shards are not meant to be used as-is. Prior to use, STI must decompress them, which is a mirror process of compression, by substituting dictionary indexes with floating point centroids and outliers. Therefore model shards quantization reduces IO but not computation (FLOPs) as the inference still executes on floating point numbers.

Third, storing shards per version STI stores each shard of every bitwidth on disk, in total $N \times M \times K$ shards (e.g. N=M=12, K=$2 \ldots 6, 32$, where 32 is the uncompressed, full fidelity). Each shard contains a weight matrix of the same dimensions listed in Table 4.1. Instead of original FP32 weights, the weight matrix now stores K-bit indexes, which reduces its file size by $32/K\times$. Additional to the weight matrix, it stores centroids and outliers as dictionaries to look up during decompression, as illustrated by Figure 4.4 (③). To load, it refers to individual on-disk shards by their original layer/vertical slice indexes and bitwidths.

4.5 Pipeline planning

4.5.1 Overview

Planning goals Towards maximizing the accuracy under a target latency T, STI plans for two goals:

• *First, minimize pipeline bubbles.* STI attempts to utilize both IO and computation as much as possible: by keeping IO always busy, it loads higher-bitwidth shards to improve submodel fidelity; by maxing out computation (FLOPs), it drives the inference towards a higher accuracy.

• *Second, prioritize bitwidths on important shards.* As transformer parameters exhibit clear redundancy, STI allocates IO bandwidths with respect to shard importance, i.e. a shard is more important if it contributes more significantly to accuracy when being executed in higher bitwidths.

Two-stage planning Towards the goals, STI conducts a two-stage planning: 1) Compute planning. Based on measured computation time of a layer, it proposes the largest submodel R' bound by T, which has the maximum FLOPs. 2) IO planning. It first assigns each layer an *accumulated IO budget* (AIB) for tracking layerwise IO resources. To allocate and saturate the IO resources, STI attempts to consume each layer's AIB. Starting from most important shards, STI assigns them a higher bitwidth, e.g. 6-bit; it does so iteratively for less important shards, until no AIB is left for each layer. We next describe details.

4.5.2 Prerequisite: offline profiling

The following measurements are done ahead of time, off the inference execution path.

Hardware capability STI measures the following hardware capabilities of a mobile device at installation time.

• IO delay $T_{io}(k)$ as a function of bitwidth k. STI measures the average disk access delay for loading one shard in k bitwidth, where $k = 2 \ldots 6, 32$. It only has to measure one shard per bitwidth because all others have same amount of parameters.

• Computation delay $T_{comp}(l, m, \mathit{freq})$ as a function of l, the input sentence length, m, the number of shards per layer (e.g. $m = 3 \ldots 12$), and *freq* as the current operating frequency of CPU/GPU. It fixes l to be commonly used input lengths after padding (e.g. $l = 128$). It does a dry run for each (l, m, freq) tuple on one transformer layer. It measures the average execution delay as the decompression delay of m shards in 6-bitwidth and the execution delay of the transformer layer composed by the m shards. Although the decompression delay is strictly dependent on the shard bitwidth, the delay differences between individual bitwidths are negligible in practice, e.g. $< 1ms$; measuring 6-bitwidth shards further provides an upper bound for decompression delays, ensuring STI always stays under the target latency.

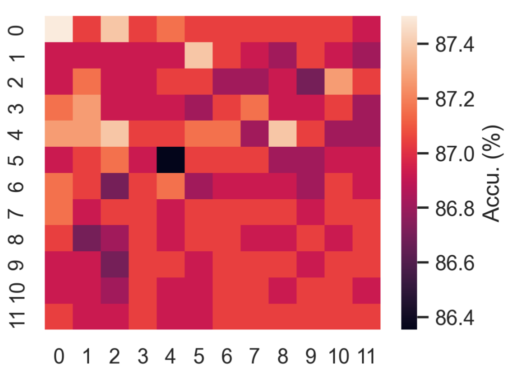
The delays can be recorded offline and replayed at run time because they are data-independent [19, 260] and are shown deterministic [261], w.r.t. the parameters $k, l, m,$ and *freq*.

Shard importance Intuitively, important shards have greater impacts on accuracy. Formally, STI deems a shard more important than another if the shard increases the model accuracy more significantly *as* they have higher fidelities. Specifically, STI profiles shard importance as follows. It first sets the full 12x12 model (i.e. with 144 shards) to the lowest bitwidth (i.e. 2-bit), enumerates through each shard, and increases the shard bitwidth to the highest (i.e. 32-bit); for each enumeration, it runs the resultant model on a dev set and profiles its accuracy. The profiling therefore produces a table (e.g. with $12 \times 12 = 144$ entries), whose each entry records the model accuracy when the individual shard is at the highest bitwidth. STI then sorts the table by model accuracy and obtains the list of ranked shard importance.

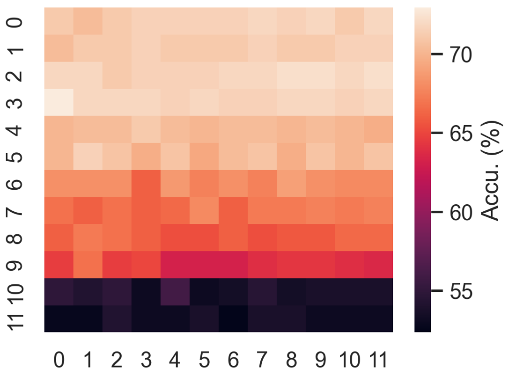
Notably, the profiling needs to be done for individual fine-tuned models, which differ in weight distribution. Figure 4.5a and 4.5b shows the example of profiling results for models used in SST-2 and RTE respectively. As can be seen, shards of different models exemplify dissimilar importance distributions. For instance, important shards distribute more evenly throughout the layers of SST-2 model yet they are much more concentrated on bottom layers (i.e. layer 0-5) of RTE model.

4.5.3 Compute planning

Given a target latency T, STI proposes a submodel sized by $n \times m$ for the incoming inference, which maximizes FLOPs.



(a) SST-2



(b) RTE

Figure 4.5: Example shard profiles on SST-2 and RTE show distinct importance distribution. Each cell at (x, y) marks a shard; the lighter its color is, the more important the shard is. Y-axis: transformer layer index, X-axis: vertical slice index.

Key ideas In searching for the submodel size, STI follows two principles: 1) whenever possible, it always picks the submodel with *most* number of shards, i.e. $n \times m$ is maximized; 2) when two candidate submodels have similar number of shards, it prefers the deeper one, i.e. the candidate with a larger n. As the transformer attention heads within the same layer are known to be redundant [212], it is wiser to incorporate more layers.

To infer (n, m), STI enumerates through all possible pairs using the profiled $T_{comp}(l, m, freq)$; the enumeration process has a constant complexity and is efficient. Since all inputs can be padded to a constant length (e.g. $l = 128$), and $freq$ is often at peak during active inference, STI only needs to enumerate in total 144 pairs in practice. For each T, the enumeration therefore deterministically gives a submodel of $(n \times m)$ which is both largest and deepest.

4.5.4 IO planning

In this stage, STI selects the bitwidths for individual shards of the $(n \times m)$ submodel. Without stalling the pipeline, it seeks those that maximize accuracy.

Problem formulation

Given the deadline T, $n \times m$ submodel R determined by compute planning, and the preload buffer S, STI plans for a shard configuration S' to load during computation, s.t. 1) loading S' does not stall the pipeline, and 2) $R = S + S'$ achieves maximum accuracy.

Accumulated IO budgets

To ensure the planning S' does not stall the pipline, STI uses *Accumulated IO Budgets* (AIBs) to track fine-grained, per-layer available IO .

Key ideas To quantify AIBs, our observation is that the pipeline does not stall *iff* before executing one layer, all shards of the current **and** prior layers are already loaded. We hence define AIBs as follows:

Definition (Accumulated IO Budgets)**.** *The AIB(k) of* k^{th} *layer is the available IO time the layer can leverage to load all shards from* $0 \ldots k$ *layers, written as* $AIB(k) = AIB(k-1) + T_{comp}(k-1)$*, where* $T_{comp}(k-1)$ *is the computation delay of the* $(k-1)^{th}$ *layer.*

The recursive definition (i.e. hence *accumulated*) encodes the data dependency between pipeline layers: each layer crucially depends on previous layers' available IO budgets and computation delays for overlapping the loading of its own shards. As of the very first layer, its AIB is set as the IO delay to fill the preload buffer S, considered as "bonus IO". For instance, the AIB of the second layer is the AIB *plus* the computation delay of the first layer, i.e. $AIB(1) = AIB(0) + T_{comp}(0)$. With the above definition, STI checks AIBs of all layers: as long as they are non-negative, STI knows each layer still has IO time remaining and the pipeline does not stall, and deems the planning *valid*.

How to use Upon each planning, STI initializes AIBs for all layers as follows. It first sets AIB(0) to be the IO delay to fill the preload buffer as described before. Next, STI sets subsequent AIBs recursively using the above definition, e.g. $AIB(1) = AIB(0) + T_{comp}, AIB(2) = AIB(0) + 2 \times T_{comp}, AIB(3) = AIB(0) + 3 \times T_{comp}$. Note that since layers have an identical structure, STI uses a constant T_{comp} across all layers.

When STI selects a shard at k-th layer, it deducts the shard IO from AIBs of k-th as well as all subsequent layers. The is because loading such shards only affect *yet-to-be-executed* layers but not the already executed ones. At the end of selection, STI checks all AIBs to see if they are non-negative. If so, STI deems the planning S' valid, otherwise rejects it.

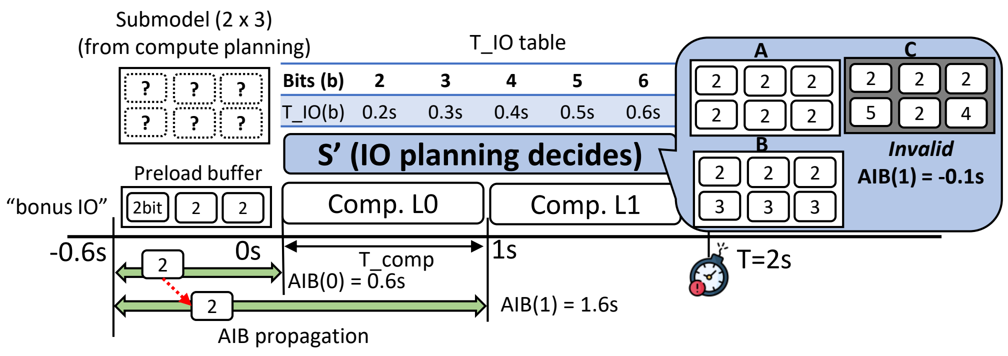


Figure 4.6: A mini example of AIB tracking the layerwise IO budgets.

Example Figure 4.6 shows a mini example of using AIBs to check the validity of S', where it plans for a 2x3 submodel, targeting a 2s deadline with $T_{comp} = 1s$. The engine initializes AIBs recursively from L0, whose $AIB(0) = 0.6s$ due to the three 2-bit shards in S. To plan, the engine first fills S' with S, deducting $0.6s$ from both $AIB(0)$ and $AIB(1)$ because all shards in S are in L0. Since only L1 has spare AIB, the engine can only select shards for it. We show three execution plan candidates A, B, and C. In this case, both candidates A and B are valid because their AIBs are non-negative, meaning loading them does not stall computation L1. Yet, C is invalid, because $AIB(1) = -0.1s$, violating the constraint and stalling the pipeline.

Selecting optimal shard versions

For each T, there exist an enormous number of execution plans. The goal is to select an optimal configuration S', which 1) is valid, and 2) maximizes accuracy. For instance, both A and B in Figure 4.6 are valid, but which has the maximum accuracy?

Key idea To ensure validity, STI respects the key invariant $AIB(k) \geq 0$ for each allocation attempt on layer k. To maximize accuracy, our key idea is to first uniformly increase bitwidths for all shards, then with the rest AIBs it greedily and iteratively allocate highest possible bitwidths to individual shards guided by shard importance. By doing so, we build an information passageway for most important shards, allowing their maximum activations to be preserved in highest fidelity as possible.

The allocation process comprises two passes as follows. In the first pass, STI picks a uniform bitwidth for all unallocated shards in the submodel, i.e. those not in preload buffer. To do so, it enumerates from lowest bitwidth (i.e. 2-bit) and selects the highest bitwidths while AIBs still satisfy the invariant. Notably, it fills a submodel layer with the shards from the same original layer and does not mix up shards across layers, due to quantization preserves intra-layer weight distribution (§4.4.2). If AIBs cannot even support 2-bit shards, e.g. due to T and/or preload buffer S too small, STI still selects them as they are necessary for execution but aborts further allocation. In the second pass, STI iteratively upgrades the bitwidths of individual shards to full 32 bitwidth guided by the shard importance profiled in §4.5.2, until all AIBs are consumed.

The allocation result is an optimal execution plan which instantiates the submodel with individual shard configurations, and is ready to be executed by the IO/compute pipeline.

4.5.5 Submodel execution

Despite STI partitions the model into shards, it still executes the plan (submodel) in a layerwise, pipelined manner from layer 0 to layer n-1. This is because although attention heads can be computed individually, FFN neurons must wait until all shards are loaded to execute.

STI executes both IO and computation as fast as possible; it does not reorder the loading of individual shards in order to meet data dependency between execution, because by design AIBs have already ensured so. To compute, STI decompresses the shards into the working buffer using the dictionaries stored along with them; the working buffer is enough to hold one layer of FP32 weights and shared by all layers during their ongoing execution. After execution, STI evicts loaded shards from top to bottom layers until preload buffer is filled. It does so because shards at bottom layers (i.e. closer to input) are needed early during inference. Preserving as many of them as possible avoids compulsory pipeline stalls in early stages.

4.6 Implementation

We implement STI in 1K SLOC (Python: 800, C: 200) based on PyTorch v1.11 [262] and sklearn v0.23.2 [263], atop two commodity SoCs listed in Table 4.2.

We preprocess the pretrained DynaBERT [229] models. We choose them because they are easily accessible and well documented. We preprocess the model as follows. To quantize a model into k bitwidth, we first partition the model by layers and gathers all weights of the layer into a large flat 1D array. We then fit the 1D array into a Gaussian distribution using `GaussianMixture` with one mixture component from `sklearn.mixture` for detecting outliers. Based on the fitted distribution, we calculate the log likelihood of each sample in the 1D weight array. Following [230] we also use -4 as the threshold – if the weight's log likelihood is below the threshold, we deem it as an outlier and records its array index; in our experiments, a model only has 0.14-0.17% outliers, which are an extremely small portion. For non-outliers which are the vast majority, we sort them based on their values and divided them into 2^k clusters with equal population. We calculate the arithmetic mean of each cluster as one centroid for representing all weights of the cluster. With such, we extract shards from the layer based on their weight composition in Table 4.1 and massively substitutes their weights with k-bit indexes to centroids; for bit alignment, we represent outliers also as k-bit integers but bookkeep their original weights and offsets in the shard. We repeat the process for each layer and for each $k = 2 \dots 6$, which takes a few minutes per bitwidth. We co-locate disk blocks of shards from the same layer for access locality. To measure shard importance, we use dev set from the respective GLUE benchmark on which the model is fine-tuned.

Implementing the layerwise pipeline is straightforward, by intercepting the forwarding function at each BERT layer and using asynchronous IO for loading shards. Yet, we have discovered Python has a poor support for controlling concurrency at fine granularity (e.g. via low-level thread abstraction), which introduces artificial delays to shard decompression. Therefore we implement the decompression in separate 200 SLOC of C code using OpenMP [264], which concurrently substitutes the low-bit integers back to FP32 centroids

Platform	CPU	GPU	Mem.
Odroid-N2+	4x Cortex-A73 + 2x Cortex-A53	Mali-G52	4GB
Jetson Nano	4x Cortex-A57	Nvidia Maxwell w/ 128 CUDA cores	4GB

Table 4.2: Platforms in evaluation. Benchmarks run on Odroid's CPU (its GPU lacks Pytorch support) and Jetson's GPU.

Benchmark	Category	Task	Metrics	Domain
SST-2	Single-sentence	Sentiment	Acc.	Movie rev.
RTE	Inference	NLI	Acc.	News, Wiki.
QNLI	Inference	QA/NLI	Acc.	Wiki.
QQP	Similarity/paraphrase	Paraphrase	Acc./F1	Social QA

Table 4.3: GLUE benchmarks [1] used in evaluation.

using all available cores of our SoCs; we expect the decompression to be further accelerated with GPU, but leave it as future work.

For miscellaneous parameters of a layer which are not part of shards, i.e. layer normalization (layernorm) and biases, we keep them in memory in full fidelity because their sizes are small, e.g. tens of KB per layer.

4.7 Evaluation

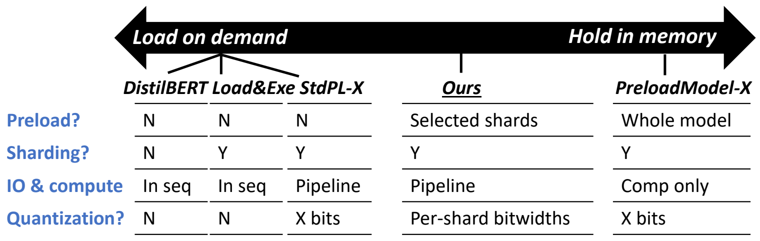
We answer the following questions:

1. Can STI achieve competitive accuracy under time and memory constraints? (§4.7.2)
2. How much do STI's key designs contribute to its performance? (§4.7.3)
3. How do STI's benefits change with available time and memory? (§4.7.4)

4.7.1 Methodology

Setup and metrics Table 4.2 summarizes our test platforms, which are commodity SoCs. We choose them to evaluate STI on both CPU and GPU. Based on user satisfaction of NLP inference delays on mobile devices [233], we set T=150, 200, and 400ms. Prior work reported that beyond 400ms user satisfaction greatly drops [233]. With T under 100ms, all comparisons including STI show low accuracy – there is not enough compute bandwidth. This is a limit in our test hardware, which shall mitigate on faster CPU/GPU.

Table 4.3 summarizes our benchmarks and metrics. We diversify them to include each category of GLUE benchmarks [1], which span a broad range of NLP use cases on mobile devices.


Load on demand
Hold in memory

	DistilBERT	Load&Exe	StdPL-X	Ours	PreloadModel-X
Preload?	N	N	N	Selected shards	Whole model
Sharding?	N	Y	Y	Y	Y
IO & compute	In seq	In seq	Pipeline	Pipeline	Comp only
Quantization?	N	N	X bits	Per-shard bitwidths	X bits

Table 4.4: Baselines for evaluation and their positions in the design space.

Comparisons We consider two NLP models. (1) DistilBERT [215], the outcome of knowledge distillation from BERT. Due to its high popularity on mobile, we use its accuracy as our references and call it *gold accuracy*. Yet, DistilBERT has fixed depths/widths (6 layers x 12 heads) and thus cannot adapt to different target latencies. (2) DynaBERT [229], which is derived from BERT (12 layers x 12 heads), allowing execution of a submodel to meet the target latency.

Based on DynaBERT, we design the following competitive baselines as summarized in Table 4.4.

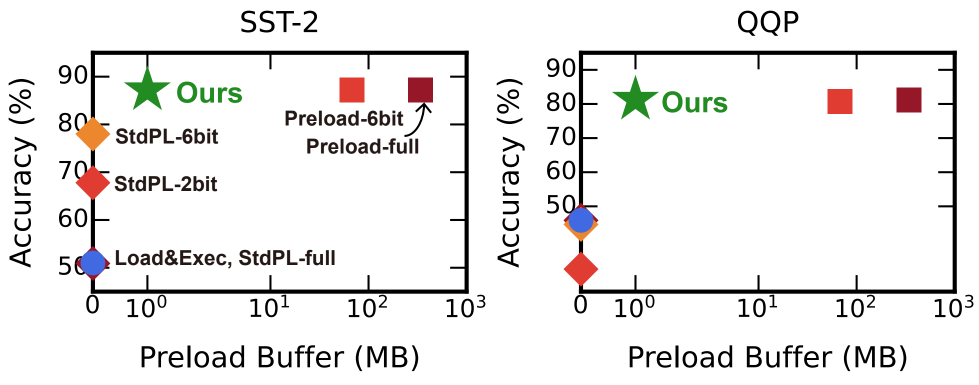
- *Load&Exec*: It loads model as a whole and executes it. It chooses the best submodel so the sum of IO and execution delays is closest to the target latency, using the algorithm described in Section 4.5.3. Model parameters are not quantized (32 bits).
- *Standard pipelining (StdPL-X)*: It executes a layerwise pipeline, overlapping IO and computation. It chooses the best submodel so that the total pipeline delay stays under the target latency. We further augment it with quantization. All parameters in a model have the same bitwidth X.
- *PreloadModel-X*: The whole model is already in memory and no IO is required. It chooses the best submodel so that the total computation delay stays under the target latency. We augment it with quantization; all parameters have the same bitwidth X.

We choose X=6 as the highest quantization bitwidth, as further increasing the bitwidth has little accuracy improvement.

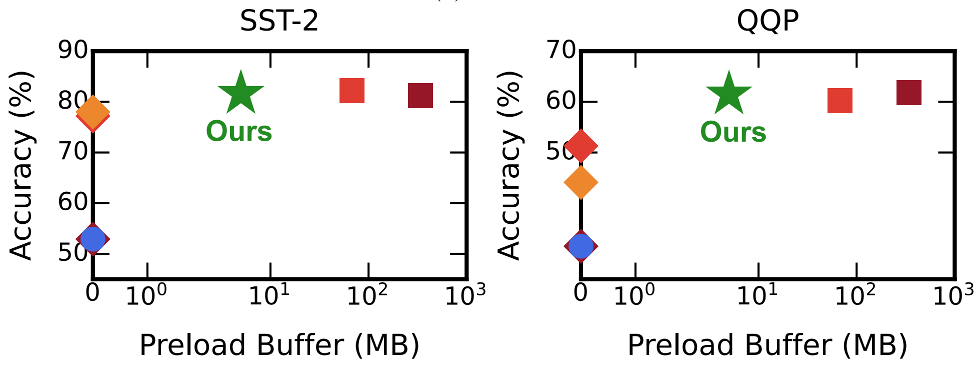
4.7.2 End-to-end results

STI achieves comparable accuracies to *gold* under target latencies (T) of a few hundred ms. Across all benchmarks and latencies, STI accuracy is on average 7.1 percentage point (pp) higher than that of baselines, which is significant.

Compared to preloading the whole model, STI reduces memory consumption by 1-2 orders of magnitude while seeing 0.16 pp higher accuracy averaged across all latencies and benchmarks; compared to loading the model on demand, STI improves the accuracy by 14 pp at the cost of preload memory of no more than 5 MBs.



(a) Odroid



(b) Jetson

Figure 4.7: STI's accuracy is significantly higher than *Load&Exec* and *StdPL*, and is similar/higher compared to *PreloadModel* albeit using 1-2 orders of magnitude smaller memory. T=200ms. Full data and benchmarks in Table 4.5.

Figure 4.7 zooms in accuracy/memory tradeoffs under T=200ms of SST and QQP benchmarks. Note that we use log scale in X-axis (memory consumption) due to its large span. STI uses 204× lower memory than *PreloadModel-full* while having less than 1% average accuracy loss. Even when compared with the quantized version (i.e. *PreloadModel-6bit*), STI uses on average 41× smaller memory to achieve the same accuracy.

Accuracy STI's accuracy matches those of DistilBERT. Given a target latency T, STI achieves consistent and significant accuracy gain over baselines. Table 4.5 shows the full view. On Odroid, STI (Ours) increases average accuracy by 21.05/21.05/17.13/5.83 pp compared with *Load&Exec*/*StdPL-full*/*StdPL-2bit*/*StdPL-6bit*, respectively. On Jetson, STI increases average accuracy by 18.77/18.77/6.53/3.15 pp compared with *Load&Exec*/*StdPL-full*/*StdPL-2bit*/*StdPL-6bit*, respectively. Notably, STI's benefit is game-changing compared with *Load&Exec* and *StdPL-full*. They are barely usable under low latency (T≤200ms).

Memory consumptions Compared with preloading the whole model, STI reduces memory consumption significantly and consistently, by 122× on average. This is because the *PrelodModel* baselines hold the whole 12x12 model in memory. By comparison, STI only needs preload memory of 1MB/5MB on Odroid and Jetson respectively, which is sufficient to hold shards of the first model layer and warms up the pipeline execution.

Benchmark (Gold accu.)	SST-2 (91.3)			RTE (59.9)			QNLI (89.2)			QQP (88.5)		
Target latency (ms)	150	200	400	150	200	400	150	200	400	150	200	400
Load&Exec	50.9	50.9	78.8	47.3	47.3	47.7	44.8	44.8	59.4	45.4	45.9	42.9
StdPL-full	50.9	50.9	78.8	47.3	47.3	47.7	44.8	44.7	59.4	45.4	45.9	42.9
StdPL-2bit	74.7	67.8	89.3	46.9	47.3	51.6	51.2	50.6	53	33.9	31.6	55.2
StdPL-6bit	78.8	78	92	47.3	47.7	67.5	59.3	54.1	88.9	41.6	44.7	88.2
Preload-full |S|:320 MB	78.8	87.2	92	47.7	52.3	68.2	59.4	72.7	88.8	42.9	81.2	88.1
Preload-6bit |S|: 60 MB	78.8	87.2	92	47.3	52.7	67.5	59.3	69.7	88.9	41.6	80.7	88.2
Ours-0MB |S|: 0 MB	78.8	87.2	91.9	47.3	52.7	67.9	56.3	71.0	88.8	39.4	80.7	88.2
Ours |S|:(a)1MB (b)5 MB	78.8	87.2	92	47.7	52.7	68.2	60	71.2	89	42.4	81.3	88.2

(a) Odroid

Benchmark (Gold accu.)	SST-2 (91.3)			RTE (59.9)			QNLI (89.2)			QQP (88.5)		
Target latency (ms)	150	200	400	150	200	400	150	200	400	150	200	400
Load&Exec	51.2	52.9	73.1	47.2	47.2	48.0	51.6	53.1	50.5	36.9	31.5	31.5
StdPL-full	51.2	52.9	73.1	47.2	47.2	48.0	51.6	53.1	50.5	36.9	31.5	31.5
StdPL-2bit	68.1	77.2	85.7	47.6	50.1	48.0	51.9	51.1	62.4	54.2	51.3	74.0
StdPL-6bit	60.2	78.0	90.7	46.5	50.5	58.4	53.1	57.6	81.8	58.3	44.1	82.9
Preload-full |S|:320 MB	65.8	81.5	91.5	46.9	51.6	62.4	53.5	54.8	86.4	58.1	61.8	85.8
Preload-6bit |S|: 60 MB	66.1	82.2	91.5	45.4	49.4	63.8	53.5	54.9	86.1	57.6	60.2	85.4
Ours-0MB |S|: 0 MB	65.9	81.6	91.6	46.9	51.9	63.1	53.6	54.6	86.2	57.3	61.5	85.4
Ours |S|:(a)1MB (b)5 MB	65.9	81.6	91.6	46.9	51.9	62.0	53.6	54.6	86.4	58.3	61.5	85.6

(b) Jetson

Table 4.5: Model execution accuracies; given target latencies, ours are the best or the closest to the best. —S—: preload buffer size. Gold accuracy from DistilBERT [215], which exceed all target latencies. End-to-end DistilBERT execution delays: 3.7s on Odroid, of which IO=3.1s; 3.36s on Jetson, of which IO=3.0s.

	Platform	Odroid (CPU)			Jetson (GPU)		
	Latency (ms)	150	200	400	150	200	400
Compute underutilized	*Load&Exec*	1x4	1x5	3x3	2x1	3x1	5x1
	StdPL-full	1x4	1x5	3x3	2x1	3x1	5x1
IO underutilized	*StdPL-2bit*	3x3	4x3	10x3	2x12	3x12	7x12
	StdPL-6bit	3x3	4x3	10x3	2x8	3x7	7x3
	Preload-full	3x3	5x3	10x3	2x12	3x12	7x12
	Preload-6bit	3x3	5x3	10x3	2x12	3x12	7x12
Compute & IO well utilized	*Ours*	3x3	5x3	10x3	2x12	3x12	7x12

Table 4.6: Sizes (depth×width) of submodels selected under different target latencies. A large submodel means more FLOPs executed, suggesting a higher accuracy. STI is able to run the largest submodel.

Storage & energy overhead For a model, STI only requires 215 MB disk space to store five fidelity versions of {2,3,4,5,6} bits, in addition to the full model (in 32 bits) of 418 MB. This storage overhead is minor given that today's smartphone has tens or hundreds GB of storage.

For a given latency, we expect STI to consume notably more energy than low-accuracy baselines (e.g. *Load&Exec*, *StdPL-full*), as STI has higher resource utilization to achieve higher accuracy. Compared to similar-accuracy, high-memory baselines (i.e. *PreloadModel-full*), we expect STI to consume moderately but not significantly more energy. First, the major energy consumer is active compute (FLOPs); similar accuracies indicate similar FLOPs. Second, although STI adds IO activities, the contribution to the system power is marginal because the whole SoC is already in high power states.

4.7.3 Significance of key designs

Submodel configuration Within a given latency, the result accuracy hinges on total FLOPs executed, which depends on the size of executed submodel. Our results show that STI dynamically adjusts submodel sizes towards the maximum FLOPs. Table 4.6 shows the details. Estimated by comparing submodel sizes: our

FLOPs is as high as that of *PreloadModel*, which however consumes 1-2 orders of magnitude more memory; our FLOPs is 7× higher compared with *Load&Exec* and *StdPL-full*, for which the IO blocks computation most of the time; our FLOPs is 1.3× higher than that of *StdPL-2/6bit*, two strong baselines that increase FLOPs through IO/compute parallelism and quantization as us; at lower T (e.g. $T \leq 200ms$), their IO delays of loading the first layer may block computation, resulting in a smaller model. Figure 4.8 shows such an example. Thanks to a small preload buffer, our executed submodel has 1.25× higher FLOPs (i.e. it has one extra layer), which leads to 9.2 percentage point (pp) higher accuracy.

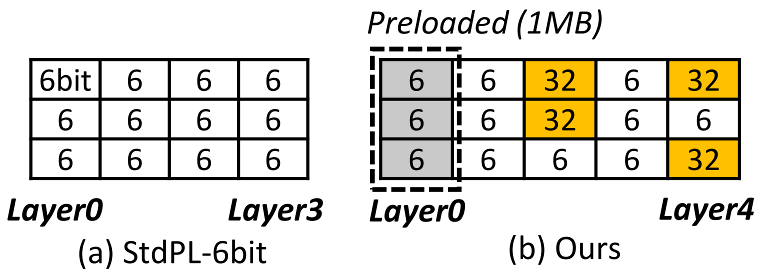


Figure 4.8: A comparison between submodels executed by *Ours* and *StdPL-6bit*. Benchmark: SST-2 on Odroid. T=200ms. *Ours* runs a larger submodel and higher FLOPs, resulting in 9.2 pp higher accuracy.

Table 4.6 also shows that our system adjusts submodels according to platform hardware. Specifically, our system assembles shallow/wide submodels on Jetson (GPU) as opposed to deeper/narrower submodels on Odroid (CPU). The reason is GPU's lack of proportionality on Transformer shards, e.g. executing a layer of 12 shards is only 0.7% longer than a layer of 3 shards. The root cause is that GPU is optimized for batch workload; it pays a fixed, significant cost even in executing a fraction of a transformer layer and for one input example, which is the case of interactive NLP.

Elastic pipelining STI's per-shard bitwidths contribute to its accuracy significantly. By contrast, one fixed bitwidth for all shards in a model is too rigid, resulting in pipeline bubbles. With a full bitwidth of 32 bits (*StdPL-full*), IO takes long and stalls the computation (19.9 pp lower accuracy than STI); with a lower bitwidth (*StdPL-{2,6}bit*), IO bandwidth is left underutilized (8.2 pp lower accuracy than STI). Any fixed bitwidth between 6 and 32 bits does not help either (Section 4.7.1). Unlike them, STI well utilizes both compute and IO through its two-stage planning (§4.5).

Preload buffers show a clear benefit as shown in Table 4.5. By using a small preload buffer of a few MBs, STI achieves a noticeable and consistent accuracy gain compared to not using the preload buffer (*Ours-0MB*). The benefit is most pronounced on QNLI and QQP among the benchmarks, increasing accuracy by up to 3.7 percent point (Odroid). Section 4.7.4 will present a sensitivity analysis regarding its size.

Benchmark	SST-2			RTE			QNLI			QQP		
IO budget (MB)	**0.4**	**2.0**	**4.0**	**0.4**	**2.0**	**4.0**	**0.4**	**2.0**	**4.0**	**0.4**	**2.0**	**4.0**
Random	79.5	79.8	81.8	48.0	48.0	51.3	51.1	51.1	52.8	39.2	40.2	59.8
Ours	81.2	83.8	85.8	50.2	54.5	54.5	53.3	60.3	62.2	56.3	63.3	75.5

Table 4.7: Model accuracies resultant from allocating additional IO budget within a 5x3 submodel of 2-bit shards. Our method shows much higher accuracies than random shard selection.

Shard importance . STI allocates its IO budgets to the most important shards. The accuracy benefit is most pronounced in a small/median submodel where most shards have low to medium bitwidths.

Case study. We demonstrate the efficacy through a differential analysis. Table 4.7 shows an intermediate state of planning: a 5x3 submodel comprising all 2-bit shards. Now the planner is awarded additional IO budgets, e.g. from enlargement of the preload buffer, with which the planner will increase some shards' bitwidths to 6 bits. We compare two strategies: (1) randomly pick shards; (2) pick shards in their importance order (as in STI). Despite the same IO budget is spent, STI shows higher accuracy by up to 23.1 percent point (8.19 percent point on average) across all benchmarks.

4.7.4 Sensitivity analysis

We examine how STI's benefit changes as resource amounts.

Target latencies A more relaxed target latency allows STI to deliver more FLOPs and execute a deeper submodel, suggesting a higher accuracy. Yet, an NLP model's accuracy sees diminishing return as its depth continues to grow, as shown in prior work [265, 229]; as a result, STI's benefit diminishes as the target latency is further relaxed. Specifically, on Odroid (CPU) STI has most significant advantage over baselines (7.7 pp higher accuracy) when target latencies are below 200 ms; in such cases, a feasible submodel has fewer than 10 layers. On Jetson (GPU) STI has most significant advantage when target latencies are below 400 ms and a feasible submodel has fewer than 7 layers. When the target latency grows beyond such ranges, STI's benefits gradually reduce.

Preload buffer size Its significance hinges on the relative speeds of computation (which consumes model parameters) and IO (which loads the parameters), because the buffer bridges the speed gap of the two. When the computation is much faster than IO, an increase in the buffer size will result in large accuracy gain, and vice versa.

On our platforms, STI shows a noticeable and consistent accuracy gain over baselines by using a preload buffer of a few MBs. Since at current preload buffer size STI has already reached best accuracy (i.e. same as *PreloadModel-full*), further increasing the buffer size does not boost the accuracy proportionally. We expect that with faster compute (e.g. neural accelerators), the preload buffer takes in a greater role. The reason is, when execution become faster and can only overlap with loading of low-fidelity shards (e.g. 2 bits), a few high-fidelity shards provided by preload buffer can significantly boost the accuracy. Such a case is shown in Table 4.7, as preload buffer sizes increase from 0.4 to 4.0 MB, the accuracy increase by 19.2 pp.

4.8 Related work

Our system is related to a wide range of ML and systems techniques. We next discuss the similarities and differences.

Model compression is a common technique for reducing model size (IO), facilitating faster loading; it includes model structure [265, 215] and feature pruning [241], and quantization which reduces full-precisions (32bit) parameters into low-bit (e.g. 2bit) representations [243, 266, 258, 252]. We use quantization to compress the model; differently, we scale compression ratios to runtime IO by instantiating multiple compressed versions. Automated quantization searches for optimal bit-widths of a NN in the offline, often on a per layer basis [257, 256, 267]. HAQ [257] adopts the reinforcement learning to find the best mixed precision for each layer, similar with our multiple versions of shards. Compared with them, we do not need any fine-tuning, which is time-consuming and we must make fine-grained decisions (i.e. per-shard) at run time.

Dynamic configuration of DNNs changes model widths and/or depths in order to suit resource constraints [229, 208, 265, 250, 268]. EdgeBERT [208] improves NLP energy efficiency under target latencies via early exit. NestDNN [268] hosts one multi-capacity model on device and switches across submodels depending on available resources. Assuming the whole model always held in memory, these systems miss the opportunities of pipelined IO/compute and therefore incur high memory cost when applied to NLP. Similar to them, we configure the NLP model architecture dynamically. Unlike them, we address the challenge of loading large models through pipelining. Furthermore, our configuration is on the basis of individual shards and adapts to both memory and latency constraints.

Pipeline parallelism for ML Pipelining has been extensively used to accelerate ML. Prior work mainly uses it to scale out ML to multiple machines (overcome limit of single machine resource). Notably for training, PP is used to partition a model or training data over a cluster of machines [237] for maximizing hardware utilization by minimizing pipeline stalls using micro/minibatches [238], exploiting hardware heterogeneity [269], or by adapting pipeline depths on the fly [270]. We share a similar goal of maximizing pipeline utilization and minimizing bubbles. Unlike that they focus on a pipeline of computations (forward/backward passes of different inputs) or network/computation, our pipeline consists of disk IO tasks and computation. Our approach towards high efficiency is through adjusting IO workloads of model shards to the computation.

4.9 Concluding remarks

We present STI, a novel system for speedy transformer inference on mobile devices. STI contributes two novel techniques: model sharding and elastic pipeline planning with a preload buffer. The former allows STI to tune model parameters at fine granularities in a resource-elastic fashion. The latter facilitates STI for maximizing IO/compute utilization on most important parts of the model. With them, STI reduces memory consumption by 1-2 orders of magnitude while delivering high accuracies under a practical range of target latencies.

Part II

Fostering security and privacy

Chapter 5

Minimum Viable Device Drivers for ARM TrustZone

5.1 Introduction

Arm TrustZone is a trusted execution environment (TEE). It hosts small programs called trustlets, which manage sensitive data against an untrusted OS. Designed for IO-rich client devices, TrustZone features *secure IO*: the TEE maps an IO device's physical resources – registers, interrupts, and memory regions – to the TEE's address space, therefore keeping them inaccessible to the OS. Trustlets can use secure IO for: (1) storing credentials, keys, and biometric data [271, 272]; (2) acquiring sensitive audio and video for processing [260, 273, 274]; (3) rendering graphical UI with security-critical contents [275]. In these use cases, IO data moves between trustlets and IO devices, bypassing the OS; the trusted computing base (TCB) consists of only the TEE and the underlying hardware.

Secure IO, however, remains largely untapped today. Popular TrustZone runtimes such as OPTEE and Trusty have been developed for almost one decade [276, 277]; yet they still cannot access important IO – flash storage, cameras, and display controllers [278]. The difficulty lies in device drivers. Implementing drivers – even only supporting a small set of functions needed by trustlets – can be non-trivial. For instance, to read a block from an SD card, a multimedia card (MMC) driver issues more than 1000 register accesses: configuring the MMC controller, exchanging 5–6 commands/responses with the storage hardware, and orchestrating DMA transfers. To do it correctly, developers need to reason about the bus protocols from over 200 pages of MMC standards [279, 280] and the device's register interface. They also need to deal with hardware quirks or bugs [281].

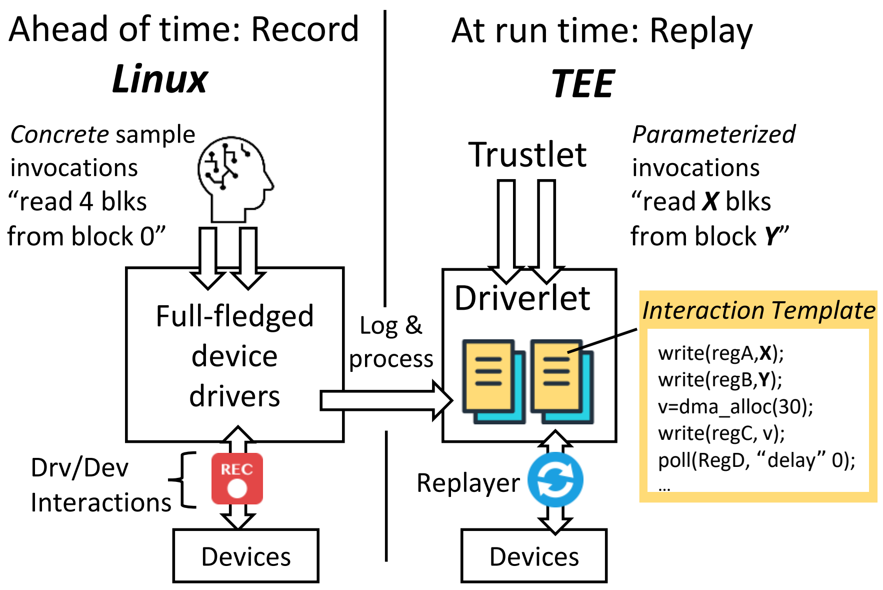


Figure 5.1: Overview of the Driverlet model. Our system records driver/device interactions from full-fledged device drivers and automatically processes them into interaction templates. The results are lightweight driverlets for serving secure IO in TEE.

How about reusing mature drivers from a commodity OS? That route is difficult as well. Mature drivers are designed to be comprehensive. They are often large, structured as multiple abstraction layers. They depend on a variety of kernel services. For instance, the MMC driver in Linux comprises 25K–30K SLoC scattered in over 150 files [16]; it invokes kernel APIs such as the slab allocator, DMA, and CPU scheduling. Developers could port the driver in two ways. (1) They move a driver and all its dependencies to the TEE (i.e. "lift and shift"). This, however, is likely to move excessive, or even most, the Linux kernel code, adding hundreds of K SLoC [282]. It violates TrustZone's principle of a lean TCB. (2) Alternatively, developers may refactor the driver, stripping code unneeded by the trustlets. To do so, they nevertheless have to reason about the driver and device internals and port a variety of kernel APIs. Our experiences in Section 5.7 show high efforts of reasoning, debugging, and trial-and-error.

Approach Since complex device drivers are overkills to simple trustlets, we advocate for a new way for deriving the drivers: instead of reusing a mature driver's *code*, we selectively reuse its *interactions* with the device. The basic idea is shown in Figure 5.1. (1) Developers exercise a mature driver with sample invocations that would be made by the trustlets, e.g. write 10 blocks at block address 42. (2) With symbolic tracing, our recorder logs driver/device interaction events: register accesses, shared memory accesses, and interrupts. (3) At run time, as the trustlets invoke the driver interfaces, the TEE replays the recorded interactions. At a higher level, our approach resembles *duck typing* in the context of a device driver: upon user inputs, a driverlet executes as a driver; upon device inputs, a driverlet reacts as a driver; then it must be as correct as a driver.

Compared to driver porting, our approach requires less developer effort. Developers only reason about the driver/device *interfaces* for recording, while remaining oblivious to their complex *internals*. Our approach respects TEE's security needs. The record runs are done on a developer's machine, an environment considered trustworthy [283]. The resultant recordings comprise only primitive events but no complex code. The replayer is as simple as a few KSLoC and imposes stringent security checks. Section 5.8 will present a detailed security analysis.

Challenges (1) *Correctness.* By following pre-recorded interactions, how does the replayer assure that it faithfully reproduces the recorded IO jobs, e.g. having written given data to block address 42? This is exacerbated by the device's nondeterministic behaviors, e.g. it may return different register values or different interrupts in response to the same driver stimuli. (2) *Expressiveness.* While the recorder exercises the driver with a finite set of concrete inputs, e.g. {*blkid=42, blkcnt=4*}, can the recorded interactions be generalized to cover larger *regions* of the input space, e.g. {*0¡blkid¡0xffff, 0¡blkcnt¡32*}?

Driverlet We present a system for recording and replaying driver/device interactions. For a given IO device, the recorder produces multiple recordings, dubbed a *driverlet*. The driverlet offers satisfactory coverage of IO functions as needed by the trustlets, e.g. to access flash storage at a variety of block granularities. A driverlet thus serves as the device's minimum viable driver.

A driverlet provides the same level of correctness guarantee as the full driver with the following insight: a replay run is *faithful* when the device makes the same *state transitions* as in the record run. Based on this insight, the recorder identifies a series of state-changing events from the recorded interactions. The state-changing events are the "waypoints" that a faithful replay must precisely reproduce. As for other interaction events not affecting the device state, the recorder emits constraints on them to tolerate their deviations from the recording in a principled manner. This approach sets us apart from many record-and-replay systems [284, 260, 285], where the replays must reproduce the recorded executions with high precision including all the non-deterministic events.

From each record run, the recorder distills an *interaction template*, which, at a high level, specifies the behavior envelope for the replay run. The template prescribes *input events* that the replayer should expect, which may come from the trustlet program, the device, or the TEE environment; the template also prescribes *output events* that the replayer should emit to the latter. By reproducing the sequence of input/output events, the replayer is guaranteed to induce the same device state transitions as in the record run, hence reproducing the IO jobs faithfully.

An interaction template is more expressive than a raw log of interactions in three ways. (1) Input events accept dynamic values that are not limited to the recorded concrete values. (2) Output events can be parameterized, encoding the data dependency between the earlier inputs and the later outputs. (3) A driver's polling loops are lifted as meta events, each replayed as a varying number of input/output events until the loop termination condition is met.

Results We implement our system with a suite of known techniques: taint tracking, selective symbolic execution, and static code analysis. We apply our approach to a variety of device drivers previously considered too complex to TEE: MMC, USB mass storage, and CSI cameras. With light developer efforts and no knowledge of device internals (e.g. the device FSM specifications), our recorder generates driverlets, each comprising 3-10 interaction templates and 50-1500 interaction events. The replayer itself has minimum dependencies on the TEE; it is in around 1000 SLoC, which are three orders of magnitude smaller than the full drivers. The driverlets incur modest overheads: on RaspberryPi 3, a low-cost Arm platform, trustlets can execute 100 SQLite queries per second (1.4× slower than a full-fledged native driver) and capture images at 2.1 FPS from a CSI camera (2.7× slower). They provide practical performance to use cases such as secure storage and surveillance.

Contributions This paper contributes:

• A new model called the *driverlet* for reusing driver/device interactions via record and replay. The driverlets generalize the recordings as interaction templates, ensuring sound replay while accepting new inputs beyond those being recorded.

• A toolkit for automatically exercising a full driver and generating driverlets.

• Case studies of applying driverlets to a variety of complex device drivers. The resultant driverlets are immediately deployable to TrustZone. With them, the TrustZone TEE gains access to these devices for the first time to our knowledge.

By fixing a long missed link in TrustZone, driverlets enable holistic, end-to-end protection of the TrustZone's IO.

5.2 Motivations

5.2.1 Example trustlets of secure IO

The following use cases motivate our design. In these cases, TEE protects the IO, which prevents the OS from observing sensitive IO data and tampering with the data.

• Secure storage. Trustlets manage sensitive data, such as credentials, fingerprints, and user emails. The trustlets store and retrieve the data by using an in-TEE flash hardware such as multimedia cards (MMC) [286] and embedded MMC (eMMC) [287].

• Trusted perception. Trustlets ingest sensitive data from the sensor hardware protected within the TEE. A particularly interesting case is video, which often contains privacy-sensitive contents and requires non-trivial camera drivers [288].

• Trusted UI. Trustlets render security-sensitive contents, such as cloud verification codes and bank account information; the UI reads in user inputs such as key presses and touches. Both the display controller and input devices are isolated in TEE [275, 289].

IO needed by today's trustlets The above use cases all rely on trustworthy device drivers, for which we exploit the following opportunities.

• *Trustlets are less sensitive to IO performance.* Today's trustlets are mostly deployed on mobile devices and tolerate overhead [271]. For instance, those managing user credentials or email contents do not need high storage throughput; trustlets for surveillance can be served with median to low frame rates and resolutions (e.g. 720P at 1FPS [290, 291]).

• *Trustlets can be served with simple IO device features.* In the examples above, trustlets only need the write/read of flash blocks, acquisition of video frames, and rendering bitmaps or vector paths to given screen coordinates. They are unlikely to need complex device features, e.g. hotplug of flash cards.

• *Trustlets may share IO devices at coarse-grained time intervals.* Concurrent trustlets are far fewer than the normal-world apps. Even when multiple trustlets request to access the same device, their requests can be serialized without notable user experience degradation. Examples include serializing accesses to a credential storage and serializing drawing requests to a trusted display.

The above observations on existing trustlets motivate our design choice: trading IO performance and fine-grained sharing for driver simplicity and security. As future trustlets may be more performance-demanding, our approach may fall short; we will discuss the limitation in Section 5.3.4.

5.2.2 Prior art

Mature device drivers They are often overkills to trustlets because of their following design choices. (1) *Optimal performance.* For instance, an MMC driver tunes bus parameters periodically (by default every second). It implements a complex state machine for handling corner cases, so that the driver can recover from runtime errors with minimum loss of work. (2) *Device dynamism.* The drivers support full device features as permitted by hardware specifications, e.g. runtime power management [292] and device hotplug [293]. (3) *Fine-grained sharing.* A driver maintains separate contexts for concurrent apps. The driver multiplexes requests at short time intervals. Some drivers, e.g. USB, implement sophisticated channel scheduling for optimizing throughput and meeting request deadlines.

Existing approaches The following approaches may be used to bring drivers into TEE.

1. *Lift and shift.* One may bundle a full driver with all its kernel dependencies and port them to the TEE [282, 205]. While easing porting, this approach often adds substantial code (tens of KSLoc) as well as bringing vulnerabilities to the TEE.

2. *Trim down.* Developers may carve out only needed driver functions from the kernel, which has been done on simpler drivers, e.g. UART [294]. On non-trivial drivers, however, the approach is impractical because the drivers have deep dependencies on complex device configurations and kernel frameworks. Section 5.8 presents our own experiences on trimming three drivers.

3. *Synthesis.* While it is possible to synthesize some drivers from scratch (assuming the device FSM is fully known), this approach requires non-trivial efforts. Developers have to write FSM specifications [295], develop code templates [296], and write glue code. In case of secure IO, we deem such synthesis efforts unwarranted, because trustlets only need simple IO functions and the source code of a mature driver is already available.

4. *Partitioning.* One may partition a full driver, executing only the security-sensitive partition in the secure world and leaving the remainder in the normal world [297, 298]. This approach, however faces two obstacles. First, it is built upon the *Trim down* approach, where the developer must manually carve out code pieces and resolve kernel dependencies. Second, it is difficult for the developer to reason about the security properties at the interface between secure/insecure partitions, where notorious attacks are common [299].

5.3 Approach overview

5.3.1 System model

SoC hardware We assume an IO device *instance* exclusively assigned to the TrustZone TEE. This is feasible as a modern SoC often has multiple independent instances of a device, e.g. 4–6 MMC controllers [150, 151, 300]. Through an address space controller (TZASC), the SoC maps selected physical memory and device registers to the TEE.

Targeted IO devices We focus on IO devices that have strong use cases in TEE. These devices manipulate sensitive IO data while lacking end-to-end data encryption. Examples include USB storage, video/audio devices, and display controllers. We make the following assumptions.

• *Device FSM.* A device operates its internal finite state machine (FSM), which encodes the driver/device protocol and decides the device behaviors [301, 302, 303]. The FSM is reactive to requests submitted by a driver, e.g. read/write flash blocks; it does not initiate requests autonomously. During request execution, the device state transitions are independent of IO data content, e.g. the state of a MMC controller is unaffected by the block contents being read.

We are agnostic to FSM internals: we only assume a device to have an internal FSM, but not the knowledge of the FSM's specifications. This is because devices often have unpublished FSMs with states not exposed to software explicitly.

• *Driver/device interactions.* The driver's interfaces to a device include registers, shared memory, and interrupts. The interactions are a set of input and output events from the driver's perspective. The input events include reading the device registers/shared memory, receiving interrupts from the device, and requesting services from the environment (e.g. DMA memory allocation); the output events include writes to the device registers and the shared memory.

• *State-changing events.* They are input/output events used by a driver to shepherd the device FSM executions. Figure 5.2(a) shows a simple example: the output events prepare a buffer and kick off the FSM execution of command `0x10`; an input event, i.e. read of interrupt (IRQ) status, reflects the FSM execution outcome; seeing the success status (`0x1`), the driver de-asserts the IRQ by writing `0x0` to the status register with an output event. Specifically, we define state-changing events as follows:

1. All output events. They directly change the device state, kick off a hardware job, etc.

2. A subset of input events, including interrupts, service responses from the environment, as well as register/shared-memory reads that have causal dependencies with at least one subsequent output event.

Note that we define state-changing events broadly, so that we err on the side of falsely assessing a successful replay as a failure, rather than a failed replay as a success. The former can be overcome with re-execution while the latter will cause silent errors. Therefore, our assessment of replay success is sound. Our system automatically identifies the state-changing events, as will be described in Section 5.4.

The gold driver We assume that the full driver implements sufficient state-changing events, so that it can assess if the device has finished the state transitions needed by given requests. We do not rule out other bugs in the driver, of which consequences will be discussed in Section 5.8.

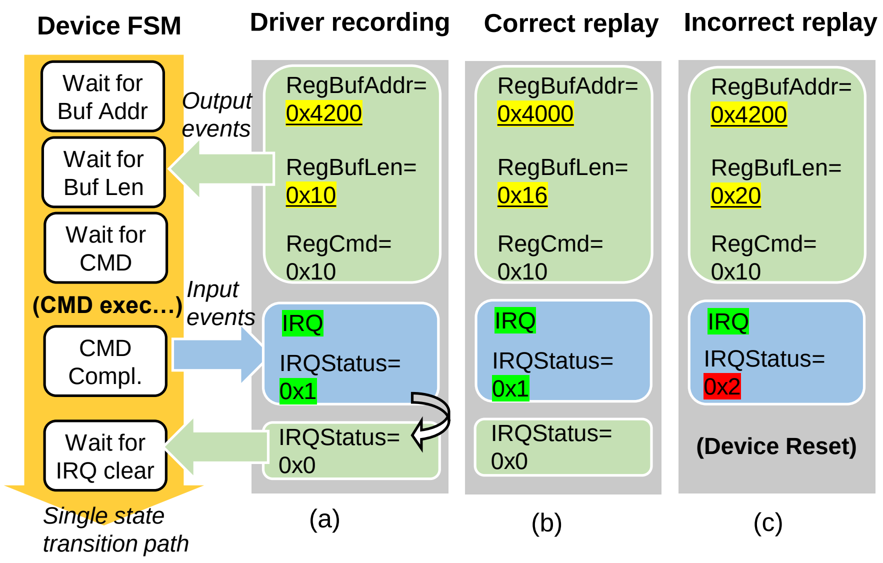


Figure 5.2: A motivating example of the driverlet approach, which captures and reuses a single device state transition path. Device FSM is implicitly assumed. All input/output events listed are state-changing events. Underlined values are generalized in a replay. Highlighted input events in a replay are expected to be matched exactly in a recording.

5.3.2 Our approach

Our idea is to selectively reuse driver/device interactions induced by device state transitions. The core mechanism is as follows.

• *Design prerequisite.* We require that a device always follows the same path of state transitions to finish a given request. As such, we configure the driver so that it constrains the device's state space: disabling irq coalescing, concurrent jobs, and runtime power management.

• *To record.* The driver is invoked with a concrete request, e.g. to read 16 blocks starting from block 42. In this process, the recorder logs the driver/device interactions, including accesses to registers, the DMA memory, and interrupts.

• *To replay.* The recorded log is generalized as an interaction template. The template strikes a balance between simplicity and expressiveness. It dictates a linear sequence of input/output/meta events, which are the replayer's minimum activities to fulfill the recorded IO jobs. The template accepts dynamic inputs (from program/environment/device) which are much broader than just the logged input values.

Section 5.4 and 5.5 will present the mechanism in full.

5.3.3 Why driverlets work

We discuss three key questions tied to the driverlet design: how does a driverlet assure the correctness? how expressive is it? what happens if it fails?

A driverlet is as correct as a gold driver, as long as the replay is faithful. For the replay to be faithful [284], the replayer observes the same sequence of state-changing events as the gold driver observes at record time. This can be proven in two steps. 1) The recorded state-changing events constitute a device state transition path; a faithful replay drives the device through the same path, which is equivalent to the recorded one as far as the device FSM is concerned. 2) If there exists an alternative path undetected by a driverlet, e.g. for a write request the same state-changing events are observed but data is silently lost, the path would also be undetected by the gold driver; this means the recorded gold driver does not implement sufficient state-changing events for inferring device states, contradicting to our assumption (§5.3.1).

Figure 5.2 (b) shows a faithful and correct replay: all the state-changing events match those in the record run (a), except for the parameters in the output events.

A driverlet is as expressive as a subset of full driver functionalities. The subset of functionalities is encoded in device state transition paths, recorded as sequences of state-changing events. Through generalizing the output events, a driverlet can vary the stimuli to the device FSM while still staying on the same state transition paths. It retains the capabilities to access arbitrary IO data but must access the data in ways specified by the recorded paths, e.g. accessing at the granularities of 1, 8, or 32 flash blocks. It leaves out other driver functionalities, such as accessing data at arbitrary granularities, optimizations for large transfers, and dynamic power management.

A driverlet deals with state divergence by device reset. Should the replayer observe any state-changing event mismatching the recording, it deems a divergence in the state transition and a failure in IO jobs, hence an incorrect replay. Figure 5.2 (c) shows the example where the IRQ status (`0x2`) mismatches the recording (`0x1`). Fundamentally, such divergences happen because the device FSM has received unexpected stimuli. We identify three major sources. (1) Residual states left from prior IO jobs. For example, if a device's FIFO is yet to be flushed, the replayer may read different numbers of empty FIFO slots at different times. (2) Fluctuation in chip-level hardware resources such as power, clock, and memory bandwidth. (3) Unexpected hardware failures. For example, a media accelerator loses the connection to the image sensor. To prevent divergences (cause 1 above) and recover from transient failures (cause 2 and 3), the driverlet resets the device before executing each template and upon the occurrence of a divergence. Section 5.8 will discuss the efficacy of recovery.

5.3.4 Limitations

First, driverlets' simplicity hinges on the data-independence of device FSM. As such, driverlets give up on devices where state transitions depend on IO data *content*, a behavior commonly seen on network interface cards (NICs). An example is BCM4329 [304], a WiFi NIC that implements 802.11 MAC in its firmware. Its Linux driver sends different commands to the NIC based on the header content of received WiFi packets. Fortunately, protecting NICs with secure IO is not as crucial as other devices, because security-sensitive network traffic through NIC can be protected with end-to-end encryption.

Second, driverlets' correctness relies on the full driver being gold (§5.3.1). The assumption is based on our empirical observation on the drivers in the mainline Linux. We, however, do not certify the full drivers. Doing so would require involving the device and driver vendors.

5.4 Record

How to use To generate a driverlet, e.g. for MMC, developers launch a *record campaign*, exercising the driver in multiple runs. In each run, the developers supply a different sample input (e.g. $blkcnt = 1, rw = 0$); from the run, a recorder produces an interaction template. Once done, the recorder signs the templates which are thereafter immutable, and reports a cumulative coverage of the input space, e.g. $0 < blkcnt < 32, rw = \{0|1\}$. Note that the sufficiency of coverage is determined by the developers. If developers see a desirable input v uncovered, they do a new record run with input v to extend the input coverage. They conclude the campaign upon satisfaction of the coverage.

5.4.1 Problem formulation

Record entry is the entry point of a recording, which requests the driver to complete an IO job. A record entry may invoke multiple functions in a driver, e.g. a series of `ioctl()` to acquire an image frame.

Recorded interfaces include the following.

• *Program* ↔ *Driver*: the interface seeds the recording. It includes record entries invoked with concrete arguments, e.g. $\{blkcount = 16, blkid = 0\}$.

• *Environment* ↔ *Driver*: it includes a number of kernel APIs invoked by a driver. Examples include DMA memory allocation, random number generation, and timekeeping.

• *Device* ↔ *Driver*: the interface is the frontier of driver/device interaction. It includes access to device reigsters, descriptors in shared memory, and interrupts.

Events		Description
Input	V=read(I, C, A)	Read *A* bytes from register/shm address *I* at *V* with constraint *C*
	V=dma_alloc(C=Null, A)	*I=dma_alloc* from env. Allocate *A* bytes of DMA memory at *V*
	V=get_rand_bytes(C=Null, A)	*I=get_rand_bytes* from env. Get *A* random bytes at *V*
	V=get_ts(C=Null, A)	*I=get_ts* from env. Get timestamp of *A* bytes (usually 4 or 8) at V
	wait_for_irq(C=Null, A)	Wait for an interrupt from IRQ number *A*
Output	write(I, V)	Write *V* to a register/shm address *I*
Meta	delay(A)	Delay for *N* microseconds
	poll(I, E, Cond)	Poll from register/shm address *I*, execute loop body *E* until condition *Cond* is met

Table 5.1: Events in interaction templates for replay. They are generic primitives for all driverlets.

Record outcome: interaction templates A template exports a callable interface to the replayer. The interface has the same signature as a record entry. The template comprises a sequence of events in Table 5.1:

• An input event $V =< I, C, A >$ expects an input V from the interface I; I can be the address of a device register, a shared memory pointer, or an environment API (e.g. `dma_alloc`). When the expected value V is specified, it must satisfy the constraint C; otherwise the replayer will reject the input event as a replay failure. The argument A specifies the input's properties, e.g. expected input length; it can be concrete or a symbolic expression of an earlier input.

• An output event $< I, V >$ writes a value V to an interface I. V can be concrete or a symbolic expression of an earlier input.

• A meta event is *delay* or *poll* $< I, E, Cond >$. The latter polls from an interface I until a termination condition $Cond$ is met. The loop body E is a series of input/output events.

We select these events as they are generic primitives in the kernel driver framework and are applicable to all driverlets. For debugging ease, each event is accompanied by its source location in the full driver.

5.4.2 Key challenges & solutions

We next discuss automatic generation of interaction templates, which must handle input variation while still maintaining correctness. We have addressed three challenges.

Challenge I: How to discover causal dependencies between input/output events? This is to identify state-changing events (§5.3.3) and discover constraints, thus to reject inputs that will compromise replay correctness, i.e. deviation from the correct device state transition path. Naively matching concrete values from record runs does not work because variations in input values may not indicate state changes. For instance, input values from a FIFO statistics register are time-dependent and hence do not always correspond

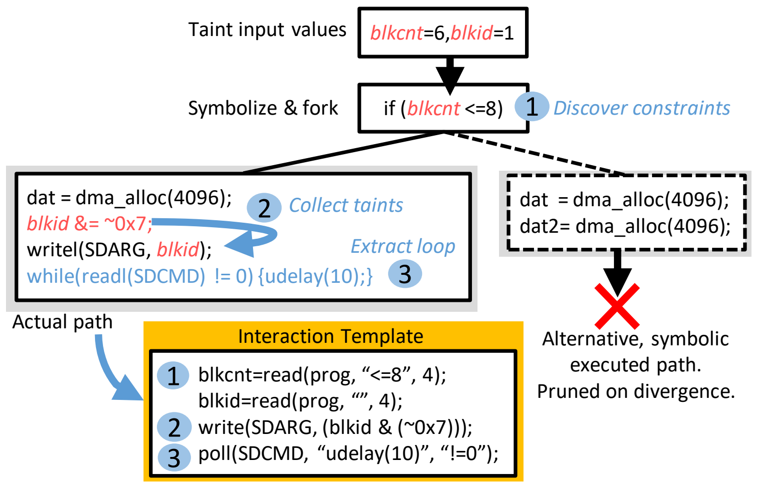


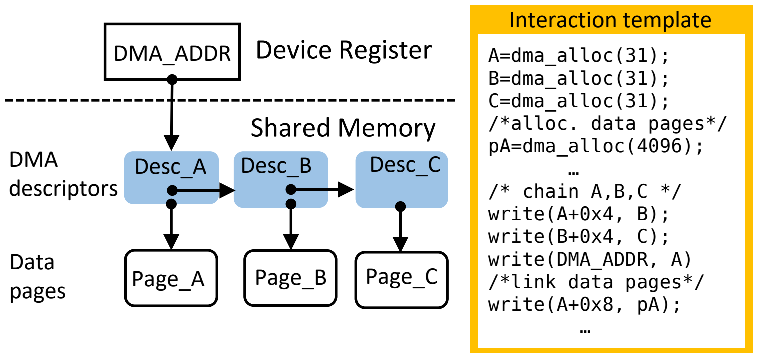
Figure 5.3: An example of our system extracting constraints, data dependencies, and polling loops into a template.

to a device state change. Instead, it's whether subsequent output events causally depend on the FIFO register value that indicates a state change.

Solution: Selective symbolic execution Our idea is to study whether variations in input values impact the driver output events, i.e. a causal dependency. The rationale is the driver, by design, always reacts to state-changing inputs and decides output events correspondingly. An example is the FIFO statistics register mentioned above: finding the FIFO watermark too high, the driver writes to a configuration register to tune the bus bandwidth which changes the device state.

To this end, the recorder uses selective symbolic execution (also called conclic execution [305]) to explore the driver's multiple execution paths and assess if they lead to different device state transitions. To avoid path explosion, the recorder prunes as soon as there is divergence in the output event sequences. This is shown in Figure 5.3. As the driver executes with a concrete input $blkcnt = 6$ and encounters a conditional branch ($blkcnt \leq 8$), the recorder forks the driver execution, explores both (one actual path with concrete $blkcnt = 6$ and an alternative path when $blkcnt > 8$), and compares their subsequent device state transitions. It discovers that the alternative path has an additional DMA memory allocation, which is a divergent input event as defined in §5.3.1; the recorder hence concludes that this path's state transitions are different and prunes it. Due to such causality, it flags the input event of *blkcnt* as a state-changing event. Throughout the execution, the recorder flags state-changing events and collects path conditions, e.g. $blkcnt \leq 8$. These path conditions serve as the constraints that input events must satisfy to stay on the same device state transition path. Section 5.6.1 will present implementation details.

Challenge II: How to discover data dependency between input/output events? Input values may be processed and used as an argument of another input event or as an output value by the driver, resulting



```
A=dma_alloc(31);
B=dma_alloc(31);
C=dma_alloc(31);
/*alloc. data pages*/
pA=dma_alloc(4096);
        …
/* chain A,B,C */
write(A+0x4, B);
write(B+0x4, C);
write(DMA_ADDR, A)
/*link data pages*/
write(A+0x8, pA);
        …
```

(a) DMA descriptor topology of an MMC controller (b) Template reconstructs such pointer topology

Figure 5.4: To reconstruct a complex descriptor topology (a), the driverlet mandates a fixed number of DMA allocations in a template (b).

in data dependencies. For instance, upon the notification of an incoming image frame of size S, the driver requests a DMA memory region of size S, and writes the region's aligned address to a device register.

Solution: Dynamic taint tracking The recorder discovers data dependencies with dynamic taint tracking: it taints all input values at all interfaces, propagating the taints in driver execution and accumulating both arithmetic and bitwise operations on the taint value until the taints reach their sinks. For each taint sink, the recorder replaces the concrete value with a symbol as the taint source with the accumulated operations on the symbol. Figure 5.3 shows an example. The recorder tracks *blkid* and discovers its taint sink *SDARG* and a bitwise operation ($blkid \& = \sim 0x7$) for alignment. It hence emits an output event with a *blkid* symbol plus the operation.

We also face challenges from higher-order dependencies. As shown in Figure 5.3, the driver allocates more descriptors when $blkcnt > 8$. Depending on the number of descriptors, a driver, per the device protocol, often links descriptors as pointer-based structures such as lists or arrays of lists; it may further optimize the structures based on descriptor addresses, e.g. coalescing adjacent ones. Figure 5.4(a) shows the descriptor topology for an MMC controller (details in Table 5.2). For every eight blocks of a request the driver allocates a 4K page and associates a descriptor with the page; it links them via a physical pointer field in the descriptor and writes the address of the head to DMA_ADDR register.

While it may be possible to extract such logic with some device-specific heuristics, such heuristics is likely brittle; both the recorder and the replayer will be more complex in order to encode and interpret the logic. For simplicity, the recorder sets DMA allocation as state-changing, mandating that a template must allocate the same number of descriptors as in the record run. The interaction template in Figure 5.4 (b) shows a faithful reconstruction of the descriptor topology: the template allocates a fixed number of descriptors, and chains them by writing their symbolized addresses to the corresponding descriptor fields.

Challenge III: How to record polling loops? A major source of nondeterminism is polling loops. For example, a driver waits for a command to finish by polling a status register; the number of register reads depends on the timing of command execution. While conceptually simple, a polling loop is known difficult to dynamic code analysis, as it can generate many or even infinite alternative executing paths [306]. To explore all paths is impractical.

Solution: Static Loop analysis The polling loops in the driver/device interfaces are often succinct, local, and have a clean code structure. For example, the RPi3 MMC driver implements polling loops by either using standard register polling functions (e.g. `readl_poll_timeout`) or a short while loop (¡10 SLoC). With static code analysis [307], we find the polling loops and lift each loop as a standalone meta event, which preserves the loop condition and the input/output events inside the loop body. This allows the replayer to execute a varying number of input/output events for a loop.

5.5 Replay

Overview In TEE, a trustlet statically links the replayer and the compressed interaction templates as a library, which constitute "driverlets" for target devices. To use a driverlet, the trustlet invokes the callable interfaces exposed by the interaction templates (§5.4.1). Under the hood, the replayer dynamically selects a template, instantiates it, and executes its input/output/meta events; the replayer resets devices between template executions and upon any device state divergence.

Selecting an interaction template The replayer decompresses the interaction template package within TEE. Upon trustlet invocation, the replayer selects one template that has all constraints satisfied by the trustlet inputs. By design, no two templates can be selected simultaneously; otherwise they should have been merged by the recorder on the same state transition path (§5.4.2). If no template is selected, the replayer reports an error that the given inputs are out of coverage.

Instantiating the template The selected interaction template preserves the symbolized input/output values and refers to them by unique names. Doing so parameterizes the new inputs supplied by the trustlet and allows them to reconstruct the recorded data dependencies. For physical device addresses, the replayer replaces them with newly mapped TEE virtual addresses. It updates the callbacks of all events, pointing them to the respective TEE APIs. Most events do not need special support from the TEE kernel. For instance, `read` and `write`, which constitute over 90% of all events, are directly dispatched to TEE's memory read/write. Meta events are implemented as simple loops and delays. Only input events at the *Env*↔*Driver* interfaces need more environment support, e.g. DMA allocation backed by a CMA pool. Luckily, they are often already implemented by existing TEE kernels, which we will describe in Section 5.6.

Executing events The replayer uses a single-threaded, sequential executor. It maintains two contexts: a normal context corresponds to the kernel context in the original driver; an interrupt context corresponds to the IRQ handler, where only a minimal part is handcrafted to recognize IRQ sources and the rest is for replaying. The scheduling of two contexts is triggered by the `wait_for_irq` input event. The design constrains the device state space in the same way as a record run by limiting hardware concurrency (§5.3.2), hence preventing many potential state divergences.

The executor is transactional. In a successful execution, all input events' constraints must be satisfied, including timely interrupts; it returns to the trustlet the requested data, if any. Otherwise, it soft resets the device and re-executes the template, which we next discuss.

Resetting device states The replayer soft resets the device under two circumstances: 1) between interaction template executions, 2) upon device state divergence. The soft reset brings the device back to a clean-slate state – as if the device just finishes initialization in the boot up process. The soft reset recovers from transient device errors. In case of persistent divergence despite of soft reset, the executor aborts and dumps the call stack; it does so by reporting all previously executed events and their recording sites (source files and line numbers). Section 5.8 will evaluate the reset efficacy and its overhead.

Self security hardening The replayer hardens itself by implementing a list of stringent security measures: it verifies recording integrity by developers' signatures; it only takes inputs from the trustlet; it does pervasive boundary checks (e.g. device physical address) on interaction templates and trustlet inputs to mitigate attacks exploiting memory bugs; it eliminates concurrency to avoid race conditions. Section 5.8.2 will present a security analysis.

5.6 Implementation

5.6.1 Recorder

We implement the recorder in 2K SLoC C code based on S2E [308], a popular symbolic execution engine. We choose it because it provides in-house support for analyzing Linux kernel drivers and is based on QEMU [170], whose dynamic binary translation (DBT) engine enables us to trace driver execution at the instruction granularity. We use existing `LoopDetector` S2E plugin to extract the polling loops. We next focus on the recorder implementation details of selective symbolic execution and dynamic taint analysis, which primarily relies on DBT. We omit the details of DBT lingoes; an interested reader may refer to [170].

Matching state transition paths As described in Section 5.4.2, we first symbolize all input events, including arguments from record entries, and register/shared-memory addresses (e.g. 24 addresses for MMC, which we will describe shortly in Section 5.7). We do so by first annotating their corresponding kernel sources, each with a custom CPU instruction. When the DBT engine translates the custom CPU instruction, it traps the execution and examines any path condition on the symbol; if any, the recorder logs the path condition and forks two new translation blocks, one as the current execution path with the concrete value and the other as the alternative symbolized execution path; Section 5.7 will present the logged path conditions. The recorder maps the state transition paths into the sequence/chain of translation blocks. As long as record runs follow the same sequence of translation blocks, the recorder deems them undergo the same state-changing events and hence are on the same state transition path. To ensure a complete and correct driver execution of the DBT engine, we supply it with concrete input values of device registers, which are collected from the of a RPi3 running the same record campaign side-by-side. A similar practice is done by Charm [309]. To save time and avoid path explosion, the recorder emulates kernel API invocations which are input events at *Env*↔*Driver* interface, e.g. `dma_alloc`. The recorder does so by checking the function addresses being translated; upon meeting them, the recorder logs their arguments (e.g. DMA allocation size) and returns directly with a symbolized result (e.g. DMA address, timestamp), instead of translating the actual kernel functions as-is.

Collecting input event taints To implement dynamic taint analysis, we interpose on the instruction translation, similar to [310]. The recorder inserts Tiny Code Generator (TCG) IR for each translated instruction which checks the taint status of source operand. The recorder applies taint propagation rules similar to [311]; depending on the rule, it updates the taint status for the destination operand. Meanwhile, the recorder logs the taint operations as its corresponding C code for debugging ease. In practice, we have found an input event is usually tainted for only a few times before reaching its sink as an output event, which is often a device register; Section 5.7 will present them.

5.6.2 Replayer

We implement the replayer in 1K SLoC within OPTEE-OS [14], whose key responsibility is to execute the replay events in an interaction template with new, dynamic values at runtime. For simplicity, we emit the recorded interaction templates as standalone header files, consisting of human-readable input/output/meta events for debugging ease; each event is encoded as a function, whose signature is listed in Table 5.1. For instance, a read event from SDCMD register expecting a 0x0 value manifests as `read(SDCMD, "=0x0", 4)`. We hence statically compile the templates and links them with the replayer. The replayer implements the events as follows. It implements read/write as uncached memory access to ensure device memory coherence; it implements poll/delay events as while loops, which continuously check against the termination conditions or timeout. For the rest events that are more complex, the replayer leverages the existing OPTEE-OS facilities: for DMA allocation, it uses the default OPTEE-OS memory allocator (i.e. `malloc`), which already allocates contiguous pages; OPTEE-OS also implements hardware RNG to for random bytes and RPC to normal world for getting timestamps.

5.7 Experiences

We put ourselves in the shoes of developers and apply our system to a variety of devices: MMC, USB, and the CSI camera. We select them because they have important use cases of secure IO and their drivers are complex and known difficult for TEE. For each driverlet, we as developers write only 20 SLoC in C as a record campaign to exercise the driver execution; we also record the device initialization process in the driver loading phase.

SoC	Raspberry Pi Model 3B+, 1GB	**Normal OS**	Linux 4.14
CPU	4x Cortex-A53@1.4 GHz	**Secure OS**	OP-TEE 3.9

MMC	Transcend 16GB microSDHC Class10 UHS-1 Memory Card
USB	Intenso GmbG Micro Line (8GB) VID:0x8644 PID:0x8003
CSI Cam	Arducam 5MP Camera with OV5647 sensor

Table 5.2: The test platform and peripherals used.

Our test platform is RPi3; Table 5.2 shows the details of the board and peripherals. We choose RPi3 because of its popularity, good support by open-source TEE and the QEMU emulation. As we will show, despite RPi3's high popularity and an active developer community, building TEE drivers with existing approaches is nevertheless challenging.

For each driver, we report our findings and answer the following questions:

- How complex is the driver and why is it complex?
- With hindsight, what are discovered by our toolkit and what interactions are recorded?

5.7.1 MMC

Driver overview

MMC is a common interface to off-chip flash, e.g. SD cards and eMMC. The Linux MMC framework supports more than 20 MMC controller models with diverse interfaces and poor documents. The framework abstracts the common driver logic as a "core" with 15K SLoC in 40 files; the driver for a concrete MMC controller plugs in the MMC core via a wide interface, including 44 callbacks implemented in four structs. The driver itself often has a few K SLoC. All combined, the MMC framework implements an FSM with thirteen states and hundreds of transitions among them. The FSM supports rich features, including streaming access of blocks and medium hotplug.

Recording outcome

replay_mmc(*rw*, *blkcnt*, *blkid*, *flag*, *buf*)					
Events	**RW_1**	**RW_8**	**RW_32**	**RW_128**	**RW_256**
Input	24/27	24/27	27/30	39/42	55/58
Output	17/14	17/14	32/29	76/73	150/147
Meta	3/3	3/3	3/3	3/3	3/3

Table 5.3: Breakdown of 10 interaction templates of MMC given the replay entry *replay_mmc*. RD/WR templates of same *blkcnt* merged in one column (e.g. RW_1), separated by "/" (e.g. 24, 27 input events for RD_1, WR_1 respectively).

We choose to implement a record campaign of 10 requests: read/write of 1, 8, 32, 128, 256 blocks. We reserve the 15-th DMA channel for recording. The replay entry, 10 automatically generated templates and their events breakdown are listed in Table 5.3. Each template covers the full range of 31M blocks (512 bytes each) available. We have found templates are similar with each other, e.g. RW_8 and RW_32 only differ by 2 DMA allocations, due to additional descriptors.

Post analysis

Our system has observed two distinct state transition paths w.r.t different flags. (1) if *O_DIRECT* is specified, the full driver shifts individual words of data blocks from/to the SDDATA register. (2) otherwise the driver uses DMA transactions to move the data. Notably, even when using the DMA transaction, the driver moves the last 3 words via SDDATA on read path. This seems to work around an undocumented bug in the SoC's DMA engine, which cannot move the last few words of a transfer.

Each interaction template involves 15 different registers out of 24 total registers of MMC controller and a system-wide DMA engine, in three groups: 8 for configuring the controllers, 5 for sending MMC commands, and 2 for controlling the DMA engine. Key symbolized input values are encoded into 32-bit words written to 4 different registers, as shown in Table 5.4. Notably, including read/write, five different SD commands (CMD17, 18, 23, 24, 25) are sent to the SDCMD register; CMD23 (set block count) is used on the read path but not write path. Our system has also gathered taints on *blkid* by Linux block layer for 8-block alignment. We tried manually feeding misaligned *blkid*, which has caused state transition divergence.

Input	Constraints	Taint sink & operations
rw	=0x1(RD)||0x10(WR)	SDCMD = ((0x8000) | ((*rw*) << 6))
blkcnt	≥0 && ≤0x8 && ≤0x400	SDHBLC = *blkcnt*
blkid	≥0 && ≤0x1df77f8	SDARG = *blkid* & (~0x7)

Table 5.4: Key constraints and taint operations of inputs on the RW_1 template of the MMC driverlet.

5.7.2 USB

Device overview

USB serves as a common transport between CPU and diverse peripherals, e.g. keyboards, and flash drives. A typical USB controller exposes more than 100 registers. A device driver programs them to initiate *transactions* and dynamically schedule them on multiple transmission channels; the controller translates each transaction into up to 12 types of packets on the bus. Through 21K SLoC in 82 files, Linux kernel fully implements the driver FSMs, which are big in order to accommodate rich features (e.g. dynamic discovery of bus topology) and various runtime conditions (e.g. device speed mismatch, bus checksum errors).

We focus on USB mass storage for its significance to TEE's secure storage. The driver accepts block requests, translates the requests to various SCSI commands, and further maps the commands to USB bulk transactions.

Recording outcome

We apply the same record campaign as MMC for 10 read/write requests. We reserve the 1st transmission channel. We disable Start-of-Frame (SoF), which is used to schedule USB transactions proactively for isochronous USB devices and is unfit for our model (§5.3.1).

For the record campaign, our system emits 200-1500 events for 10 interaction templates, each covering the whole 15M blocks of the USB storage. Interestingly, the number of events are identical in a read template and the corresponding write template; it appears only some output values to certain descriptors differ.

Post analysis

Our system has captured non-trivial interactions. To write blocks smaller than one LBA (4KB), the driver reads back the entire LBA, updates in memory, and writes back. The driver also selects the best SCSI commands: there exist five variants of a SCSI read/write command, which have different lengths and can encode different ranges of LBA; the driver picks the 2nd shortest ones (i.e. read 10, write 10) just long enough to encode the requested LBA addresses.

Our system identifies 14 USB controller registers out of the 64 KB register range in three categories: 5 manage USB peripheral states (e.g. device power state); 3 manage the controller itself (e.g. interrupts); 6 manage transmission channels (e.g. DMA address & size). Unlike MMC, the USB driver communicates with the device primarily via two descriptors: one command block wrapper (CBW) for SCSI commands and the other for querying command status (CSW). The data dependencies are similar to MMC, except aligned *blkid* and *blkcnt* are written to CBW instead of registers. Our system identifies two statistic inputs which are unseen in MMC: a monotonic command serial number and an HFNUM register read. As they are not state-changing, our system does not impose any constraints.

5.7.3 Camera

Device overview

On a modern SoC, CPU typically offloads video/audio processing to accelerators, which communicate with CPU primarily via messages backed by shared memory. We studied VC4, the multimedia accelerator of RPi3. According to limited information, VC4 implements key multimedia services, including camera input, display, audio output. It communicates with CPU via a complex, proprietary message queue called VCHIQ. Using VCHIQ as a transport, each media service further defines its message format and protocol, e.g. MMAL (MultiMedia Abstraction Layer) for cameras. The details of VCHIQ, as well as internals of VC4, remain largely undocumented.

We focus on one media service essential to secure IO: image capture from a CSI camera (a pervasive image sensor interface used in modern mobile devices).

replay_camera(*frame*, *resolution, buf, buf_size, img_size*)			
Events	**OneShot**	**ShortBurst**	**LongBurst**
Input	34	61	331
Output	36	54	234
Meta	5	22	115

Table 5.5: Events breakdown of 3 interaction templates under a given resolution for the CSI camera with the replay entry *replay_camera*.

Recording outcome

We choose the record campaign: capture 1, 10, 100 image(s) at 720p, 1080p, 1440p.

For the record campaign, as listed in Table 5.5, our system emits 3 interaction templates (OneShot, ShortBurst, LongBurst for capturing 1, 10, 100 images(s) respectively) of 75-680 events. Templates cover all resolutions supported by the camera, and a practical range of frames. Unlike both MMC and USB, the template's input/output events are mostly accessing shared memories.

Input	Constraints	Taint sink & operations
resolution	= 720p || 1080p || 1440p	(*queue*+0x239c0) = *resolution*
buf_size	>= *img_size*	(*queue*+0x24000) = *buf_size*
img_size	>= 0 && =(*queue*+0x5630)	(*queue*+0x5e86) = *img_size*, (*pg_list*+0x0) = *img_size*
pg_list	!= NULL	(*queue+0x24198*) = *pg_list*
queue	!= NULL	MBOX_WRITE = *queue & ~(0x3fff)*

Table 5.6: Key constraints and operations of input values for the camera driverlet. *queue* and *pg_lsit* are DMA addresses allocated from `dma_alloc`.

Post analysis

Of the templates, our system identifies only three registers in use. Two of them are a pair of "doorbells" for inter-processor interrupts between CPU and VC4; only one MBOX_WRITE register acts as a sink for *queue*, which points to base address of message queue. We summarize the discovered dependencies in Table 5.6. A notable constraint is for *img_size* input value. It is assigned by VC4 and is sent back to VC4 in a message initiating bulk receive procedure; later when the procedure finishes, VC4 passes another input value indicating successful transmission size at *queue+0x5630*, which *img_size* must exactly match.

Catering to concurrent media services, the message queue has a sophisticated structure. It is divided into many 4KB slots, each assigned either to CPU or VC4 for enqueueing messages independently. Each slot holds multiple messages, ranging from 28 bytes to 306 bytes. These messages belong to tens of types, either for configuring the device, e.g. opening a service "port" of VC4, setting frame resolution; or for moving data, e.g. bulk receive. Slot 0 is special, as it contains metadata that describes the whole message queue and will be updated by both CPU and VC4, e.g. the number of slots, slot allocations, read/write locations in the message queue. The doorbell registers – BELL0 and BELL2 signal CPU and VC4 to parse new message, respectively. Upon a new doorbell, a slot handler thread actively polls and parses the message; a sync thread synchronizes CPU-VC4 shared states in slot 0; a recycle thread actively frees and recycles used slots.

5.8 Evaluation

In this section we answer the following questions:

1. How does our system reduce developer efforts? (§5.8.1)
2. Why are driverlets correct and secure? (§5.8.2)

3. What is the overhead of driverlets? (§5.8.3)

4. How to use driverlets to build trustlets? (§5.8.4)

5.8.1 Analysis of developer efforts

We compare three approaches to implementing the *same IO functionalities* as described in Section 5.7.

	CMDs	Proto. Spec.	Dev. Spec.	Trans. Paths	Reg./Fields	Desc./Fields
MMC	5	231	30	10	17/63	1/8
USB	4	650	Unavailable	10	14/100	4/32
VCHIQ	8	Unavailable	Unavailable	9	3/3	10/104

Table 5.7: Efforts for building drivers from scratch, showing the needed device knowledge. Proto. Spec. is Protocol Specifications and Dev. Spec. is Device Specifications; both are counted in *number of pages*.

Build From Scratch Developers handpick a set of device commands to implement, for which they consult device specifications, implement the state transitions, and work around hardware quirks (§5.7.1). Table 5.7 gives a summary of the needed knowledge. We estimate that each driver takes a few months to build.

	Functions	Dev. Conf.	Macros	Callbacks	SLoC
MMC	22	11	90	79	1K
USB	58	14	427	142	3K
VCHIQ	137	9	405	159	11K

Table 5.8: Efforts for porting Linux drivers, showing the code the developers need to reason about and potentially modify. Dev. Conf. is Device Configurations.

Port Developers familiarize themselves with device specifications, and decide what driver/kernel functions to port (and what not to). They must spin off the code paths, which span 22–137 driver functions as we measure. To resolve the kernel dependencies of the select code paths, the developers have to port at least 1K, 3K, 11K SLoC for each of the MMC, USB, and VCHIQ drivers; they need to port at least 5K SLoC for emulated kernel frameworks such as block, power and memory management. We estimate that it takes a few months to understand each driver and port its dependencies, plus several months to build the emulated kernel frameworks.

Our approach By comparison, we require much lower developer efforts. We build the toolkit in several weeks, which is a one-time effort. To derive each driverlet, we familiarize ourselves with the full driver's register definitions and instrument the input interfaces for code analysis. Each driverlet takes 1–3 days.

5.8.2 Correctness & Security analysis

We experimentally validate the correctness and security concerns. We discuss them separately: correctness violation is caused by software semantics bugs; security breaches are caused by active attackers who compromise the software.

Correctness

Driverlets' correctness can be affected by semantics bugs in the OS and the driver for recording, e.g. the driver writing to a wrong device register. Such bugs result in malformed recordings and incorrect replay outcomes. Driverlets neither mitigate nor exacerbate such semantic bugs. Our recorder and replayer may introduce semantic bugs, e.g. due to implementation glitches. Yet, we expect such bugs to be rare because of software simplicity: the recorder and the replayer are only 3K SLoC.

Experimental validation We further validate driverlets' correctness experimentally by following the practice in prior work [312, 313]. We develop test scripts to do the following:

• *Statically vetting of templates.* Our scripts scan templates as a sanity check for the integrity of state-changing events, e.g. which SCSI command is written to what register, what MMAL message is sent. The scripts verify that the templates conform to the record campaign and developer requests.

• *Validation of IO data integrity.* For MMC & USB, our scripts verify that values read by driverlets match those by native drivers and that writes reach the storage; for VCHIQ, the scripts analyze the captured images and verify that they are in the valid JPEG format.

• *Stress testing templates.* Our scripts enumerate templates to stress test and validate the coverage of input space. The scripts verify a 100% coverage for MMC/USB blocks (MMC: ¿31M, USB: ¿15M of blocks); for VCHIQ, the scripts repetitively invokes templates for 10K times and verify runs produce integral frames.

• *Fault injection.* We validate that driverlets handle state divergences properly. To do so, we unplug the MMC/USB storage medium amid a replay run for a large data transfer (2K blocks). The driverlet correctly detects divergence and attempts re-execution with reset. Because the injected failure is non-recoverable through soft reset, the driverlet eventually gives up. It reports unexpected values from two status registers (SDEDM for MMC and GINSTS for USB) as well as the source lines of the register reads in the original drivers, allowing quick pinpointing of the failure causes.

Security

Threat model We follow the common threat model of TrustZone [314, 297]. On the target machine: we trust the SoC hardware including any firmware; we trust the TEE software; we do not trust the OS.

We assume that the OS and driver on the developer's machine for recording are uncompromised. The rationale is that the developer machines are often part of a software supplychain with strong security measures. Compromising them requires high capability and long infiltration campaigns [283].

Security benefits By leaving drivers out of TEE, driverlets therefore keep the TEE immune to extensive vulnerabilities in the driver code. Examples include dirtyCoW [315] caused by race conditions in the page allocator, BadUSB [316] caused by unrestricted privileges in the USB stack, and memory bugs in drivers [317, 318, 319]. They could have been exploited by adversarial peripherals (a malicious USB dongle [316]) or malformed requests sent from the OS to the TEE.

Attacks against driverlets (1) Fabricating interaction templates is unlikely. This is because they are signed by developers, whose recording environment is trusted. (2) Attacks against the replayer. Vulnerabilities in the replayer may be exploited by entities external to the TEE, e.g. an adversarial OS or peripherals. Successful attacks may compromise the replayer or even the whole TEE. However, such vulnerabilities are unlikely due to replayer's low codebase (only 1K SLoC), simple logic (such as minimalist memory management and well-defined event semantics), and stringent security measures (§5.5).

5.8.3 Overhead

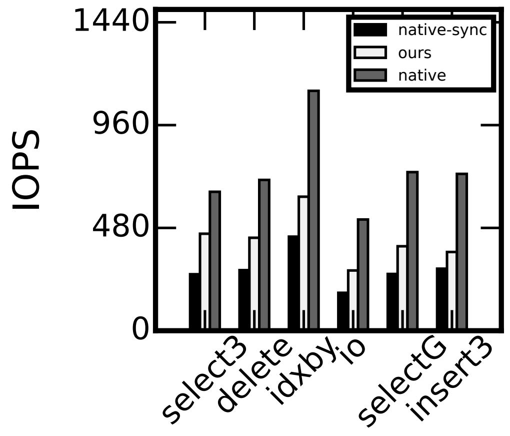
Methodology

The details of our test hardware are listed in Table 5.2. Because the RPi3 board does not implement TZASC, we modified Arm trusted firmware to assign devices instances to TEE. We isolate the whole MMC and VC4 instance. We reserve 3 MB of TEE RAM and use the stock OPTEE allocator for DMA memory.

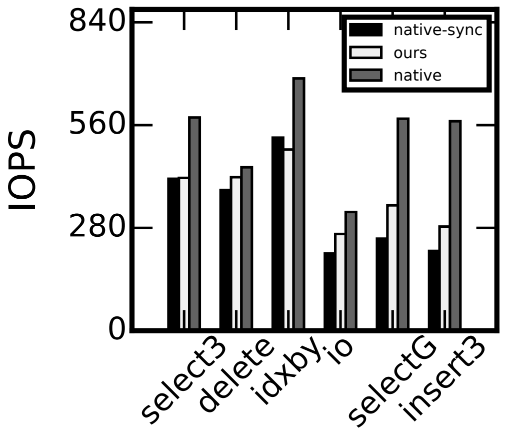
	RW_1	RW_8	RW_32	RW_128	RW_256	R:W
select3	36	12	8	4	12	10:0
delete	28	20	5	4	12	9:1
idxby	56	35	5	4	12	9:1
io	15	16	5	4	14	8:2
selectG	42	18	18	5	14	6:4
insert3	39	15	19	4	16	5:5

Table 5.9: Benchmarks used from SQLite test suites and a breakdown of interaction template invocations. Template details are shown in Table 5.3 and in Section 5.7.

Benchmarks 1) **SQLite-MMC**: we choose SQLite, a popular lightweight database to test MMC. We pick 6 tests from SQLite test suites to diversify read/write ratios; breakdown of their template invocations and read/write ratios are shown in Table 5.9. The tests issue their disk accesses in TEE and we report IOPS. 2) **SQLite-USB**: we test USB mass storage with the same SQLite test scripts. 3) **Camera** (OneShot/Short-Burst/LongBurst): we request VCHIQ to capture 1, 10, 100 still images frames at 720P, 1080P, and 1440P. We report the latency of each request.



(a) SQLite-MMC



(b) SQLite-USB

Figure 5.5: SQLite benchmarks for MMC and USB driverlets. Driverlets' overhead increases with write ratios due to they mandate synchronous IO jobs while native drivers do not.

Note that unlike many TrustZone systems [286, 297], driverlets do not incur world-switch overheads because they fully run inside the TrustZone.

Comparisons We compare driverlets with drivers on Linux 4.14 which finishes the same benchmarks as follows: for **SQLite**, we run a test harness invoking the full drivers with the same block accesses with default flags (**native**) and with an additional O_SYNC flag (**native-sync**); for **Camera**, we run `v4l2-ctl` to request the same number of frames at corresponding resolutions (**native**).

Macrobenchmarks

SQLite-MMC Figure 5.5a shows the results. MMC driverlet achieves a decent performance: on average, it achieves 434 IOPS, executing over 100 queries per second. As a reference, the throughput is a few orders of magnitude higher than secure storage hardware, e.g. RPMB [320].

Compared with the native driver, MMC driverlet's throughput is 1.8× lower on average. The overhead grows with the write ratio, e.g. select3 (read-most) incurs 1.4× overhead while insert3 (write-most) incurs 2×. This is because the driverlet mandates synchronous IO jobs: while the native driver does not wait for writes to complete, the replayer must wait to match state changing events. To validate, we mandate O_SYNC flag in the native driver execution (native-sync) and measure throughput 1.5× lower than driverlets. This is because driverlets forgo complex kernel layers such as filesystems and driver frameworks.

SQLite-USB Figure 5.5b shows the results. The driverlet achieves 369 IOPS which is over 90 queries per second. The overhead compared with the native driver is 1.5×. Such overhead is also caused by synchronous

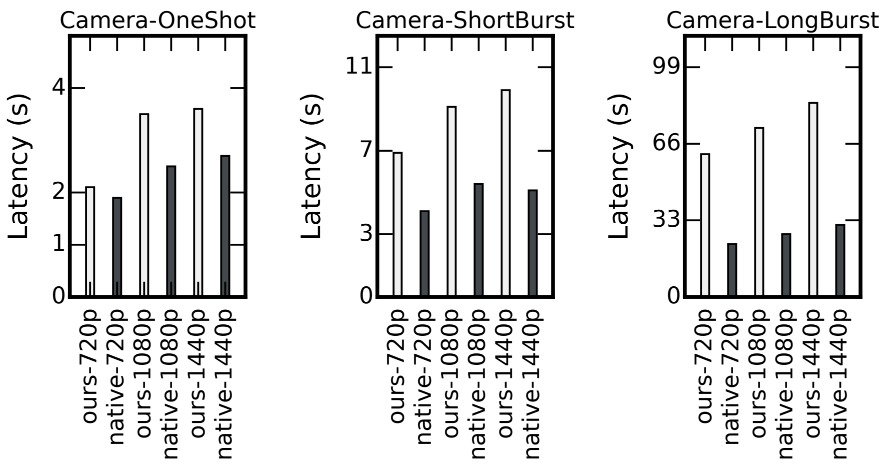


Figure 5.6: Image capturing latency for Camera benchmarks. OneShot, ShortBurst, and LongBurst are for capturing 1, 10, and 100 images(s) respectively.

writes, where the write-most workload (insert3) incurs the highest overhead of 2×. Native-sync is 1.7× lower than the native USB driver and 1.2× lower than USB driverlet.

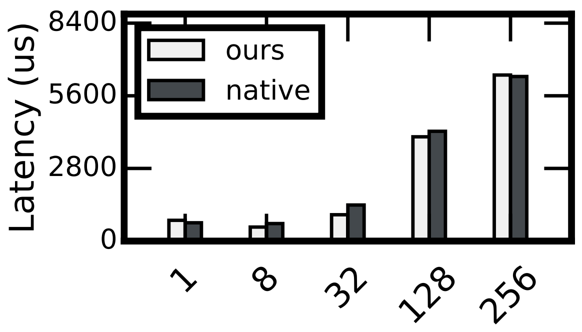
Camera Figure 5.6 shows the results. The per-frame latencies of driverlet range from 2.1s (720p) to 3.6s (1440p) which are usable to many surveillance applications that periodically sample images [290]. The per-frame latencies decrease with the number of frames per burst, because for each burst the driverlet pays a fixed cost to initialize the camera and the media accelerator. It replays 41 events for the initialization and 5 events to capture each subsequent frame.

Compared with the native driver, our latency is only 11% higher for a one-frame burst and is 2.7× higher for a 100-frame burst. This is again because the driverlet must wait for individual IRQs as mandated by the templates (§5.3.3), while the native driver processes coalesced IRQs. As requests contain more events (e.g. 75 vs. 680 to capture 1 and 100 frames, Table 5.5), the delays of waiting for IRQs are more pronounced.

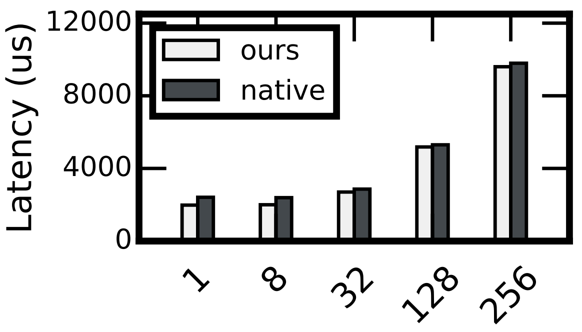
Microbenchmarks

We show latencies to execute individual templates for MMC and USB in Figure 5.7.

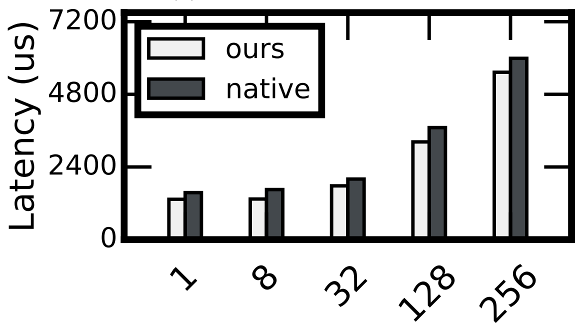
In both reads/writes, the driverlet's latencies are near-native or even slightly lower than the native ones (12% and 13% lower for MMC and USB respectively). On larger block writes (e.g. 256), the latency of USB driverlet is even 40% lower; this is due to the driverlet, unlike a native driver, does not run transfer scheduling logic for individual 4KB data pages. This confirms observations in the macrobenchmarks, where the driverlet outperforms native-sync.



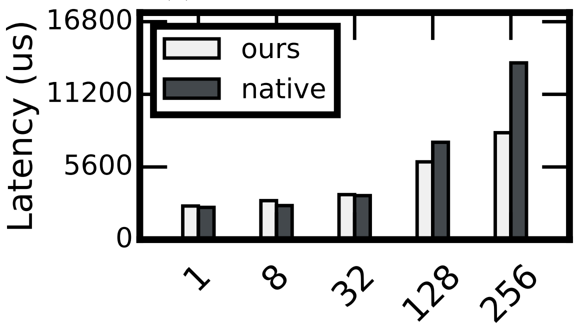
(a) MMC read latency



(b) MMC write latency



(c) USB read latency



(d) USB write latency

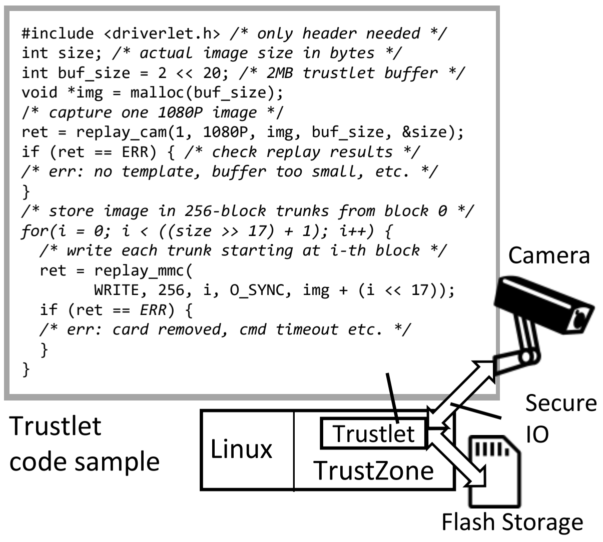
Figure 5.7: Microbenchmarks of read/write on the MMC and USB driverlets. X-axis: number of blocks, Y-axis: Latency in milliseconds. Driverlets achieve near-native performance or even outperform the native 256-block writes due to its simplicity.

Memory overhead.

The driverlet executables for MMC, USB, and VCHIQ are of 6 KB, 26 KB, and 19 KB, respectively. For implementation ease, our current recorder emits templates as human-readable documents. Conversion to binary forms is likely to further reduce their sizes.

5.8.4 End-to-end use case

To showcase the use of driverlets, we build an end-to-end trustlet for secure surveillance in only 50 SLoC and less than an hour. As shown in Figure 5.8, the trustlet periodically samples image frames and stores the frames on an SD card. Without driverlets, such a trustlet cannot enjoy secure IO path: it has to invoke the OS for the needed drivers. With driverlets, the trustlet code simply includes one header file and invokes the two interfaces for camera and MMC respectively. Corresponding to the invocations, the replayer selects two interaction templates: one for image capturing and the other for writing 256 blocks. To store each frame which is 1–2 MB, the replayer invokes the latter multiple times. We have measured that capturing each frame takes 3.7s, in which most is spent on initializing the camera and storing the image only takes 154 ms.



```
#include <driverlet.h> /* only header needed */
int size; /* actual image size in bytes */
int buf_size = 2 << 20; /* 2MB trustlet buffer */
void *img = malloc(buf_size);
/* capture one 1080P image */
ret = replay_cam(1, 1080P, img, buf_size, &size);
if (ret == ERR) { /* check replay results */
/* err: no template, buffer too small, etc. */
}
/* store image in 256-block trunks from block 0 */
for(i = 0; i < ((size >> 17) + 1); i++) {
  /* write each trunk starting at i-th block */
  ret = replay_mmc(
        WRITE, 256, i, O_SYNC, img + (i << 17));
  if (ret == ERR) {
  /* err: card removed, cmd timeout etc. */
  }
}
```

Figure 5.8: A trusted perception trustlet built atop driverlets, which expose two simple interfaces (*replay_cam* and *replay_mmc*).

5.9 Discussions

A simpler recorder without symbolic tracing To generate driverlets, the developers rely on symbolic tracing. While the tool (i.e. DBT) exists for popular hardware such as RPi3, it may not be always available or easily accessible. Under such circumstances, developers may resort to tracing the *concrete* interactions at the three interfaces (§5.3.1), exercised by desired record campaigns, as has been demonstrated in prior work [260]. Additionally, our experiences in Section 5.6 and 5.7 show that the generated constraints and taints are likely simple, which entail manageable efforts to verify the state transition paths.

Applicability to TEEs other than TrustZone Despite we choose Arm TrustZone as a key use case for driverlets, driverlets themselves rely only on secure IO and are not tied to Arm TrustZone nor a specific TEE. For instance, driverlets can be used in Keystone of RISC-V [321], which achieves secure IO via PMP (for memory/registers isolation) and processor M-mode (for routing IRQs); for SGX which lacks native secure IO support, driverlets require additional techniques [322]. Driverlets are not tied to a specific TEE kernel or host environment, neither. As long as the host environment implements replayer and replay events correctly, driverlets should work out-of-box, e.g. in TEE kernels such as Trusty [323] and QTEE [324] or unikernels [173].

5.10 Related Work

Driver reuse To reuse drivers, some "lift and shift" [205, 160, 282, 161]; some trim down simple drivers [14]; Our system shares same goal. Different from them, our system derives drivers by a novel use of record and replay.

Record and replay is well-known and primarily applied to bug finding [325, 326, 327, 284] and security analysis [328, 329, 330]. It also enables offloading [331, 332], emulation [333], and cheap versioning [285]. We are inspired by them to reproduce a subset of program behaviors, e.g. device/driver interactions. The key difference is we generalize replay inputs such that replay completes requests beyond those recorded. Quite related to driverlets, GPUReplay [260] also records and replays the device interactions at hardware/software boundary. Compared to it, driverlets face a different challenge: how to generalize and parameterize recordings; this challenges necessitates a new construct – interaction template; by contrast, GPUReplay does not parameterize input events.

Program analysis techniques have been widely applied by extensive works for testing [312] and finding vulnerabilities [307] in kernel drivers and excavating data structures (e.g. in binaries [334, 335] and network protocols [336]). Inspired by them, our system is built atop well-known program analysis techniques. Differently, we use those techniques for a distinct goal: reuse device state transition paths.

Trusted execution environment is commonly used to shield trustlets from untrusted host OS [297, 337, 338, 274]. Lacking storage drivers, the trustlets delegate IO to OS [286, 339, 340] and mediate their accesses [341]. Similar with them, we leverage TEE's strong security guarantees; differently, we provide key missing device drivers for them. Some works bring drivers to TEE, e.g. IPU [275], GPU [260]. Compared to them, which are point solutions to individual devices, we present a holistic approach to systematically derive a set of drivers.

5.11 Conclusions

We present a novel approach to deriving device drivers for TrustZone. Our toolkit records driver/device interactions from a gold driver and accordingly distills interaction templates; by replaying a template with new dynamic inputs, the driverlet completes requests beyond the one being recorded while assuring correctness. We build the recorder/replayer and show that driverlets have practical performance on MMC, USB, and VCHIQ. Driverlets fix the key missing link for secure IO, and for the first time open a door for trustlets to access complex yet essential devices.

Chapter 6

Protecting File Activities via Deception for ARM TrustZone

6.1 Introduction

TrustZone is the trusted execution environment (TEE) on Arm CPUs. To use TrustZone, developers encapsulate security-sensitive code as trustlets, which are isolated in the secure world and shielded from an untrusted OS [297, 342].

File services for TEE Many trustlets store security-sensitive data as files, such as sensor readings and login credentials. As shown in Figure 6.1(a), trustlets often export file calls (e.g. open/read/write/close) to the OS which hosts a modern filesystem. Doing so gives trustlets access to modern file features such as crash consistency and flash optimizations from various mature filesystems; meanwhile the filesystem code does not have to be pulled into the TEE, keeping the TEE lean.

Question & challenges Reliance on an external filesystem suffers from a key drawback: leak of file activities. Although a trustlet can encrypt file contents for confidentiality and integrity, it has to send file *activities* in the clear. The activities include file operation types (e.g. read, write, seek, and create), sizes/offsets, and access occurrence (e.g. "the trustlet just created a file"). From the received file activities, the OS can infer a truslet's secrets such as input data [15, 282]. Section 6.2.3 will show evidence.

In general, access activities can be obfuscated by injecting sybil activities [343, 344, 345]. When it comes to file activities, existing solutions are inadequate. (1) Popular obfuscation techniques, e.g. ORAM [346], focus on hiding data access *addresses* in a flat, memory-like space. While they can generate random offsets within a file [15], they cannot generate file operations with rich semantics, e.g. "read /a/index at offset 42;

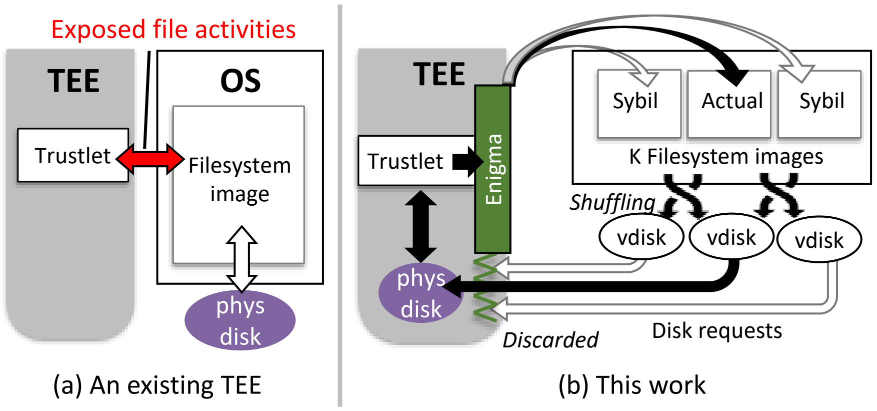


Figure 6.1: An overview of Enigma.

then open /b/data and write at offset 1024". (2) How to make sybil file activities *credible*? Merely making them *legal*, e.g. no out-of-bound reads, is not enough to deceive an OS that has prior knowledge of the true file activities. (3) How to minimize the cost of sybil activities, which often amplifies the actual activities significantly? For instance, a file backed by ORAM-like disk blocks consumes up to 10× more space and slows down access by at least one order of magnitude [15].

Enigma is a deception mechanism that hides file activities for a TrustZone TEE. It centers on two insights. First, while invoking an external filesystem, the TEE conceals the underlying physical disk[1]. This allows the TEE to inject numerous sybil file calls but discard their disk activities covertly with little cost. Second, to make sybil activities credible, the TEE should borrow knowledge from the trustlet under protection.

1. Sybil filesystems with covert emulation Sybil activities are expensive as they pollute the actual data on disk. We therefore instantiate multiple filesystem images: one *actual* image, to which the TEE sends actual file calls; and many *sybil* images, to which the TEE sends sybil calls. This is shown in Figure 6.1(b). The separation of filesystem images allows the TEE to fulfill their disk requests differently: performing all the disk requests from the actual image; silently dropping *filedata* accesses from the sybil images. Essentially, the TEE emulates storage for sybil images with only their *metadata*, reducing their overheads to just enough for deceiving the OS. The TEE further implements measures against OS probing the internals of such covert emulation.

2. Protecting filesystem identity via shuffling As the OS observes longer history of file activities, it poses an increasing threat. For instance, the OS can determine which image may be actual by comparing the current and the past file activities on an image. If the OS succeeds, it uncovers all the actual file activities in retrospect and in the future.

[1]This paper uses disks to refer to storage hardware including flash.

Our defense is to prevent the OS from observing file activities on any filesystem image for long. In the spirit of moving target defense (MTD [347]), TEE periodically shuffles filesystem images that have identical OS-visible states. Not knowing the shuffling scheme, the OS can only track an image's activities for a short period of time, less than several seconds in our implementation. TEE does shuffling efficiently by only updating metadata references, not the metadata itself or filedata.

3. Generating credible activities calls via replay The TEE should issue sybil file calls close to what the trustlet would issue; it cannot draw sybil calls, for example, from generic file traces. The TEE can only deceive the OS when it knows the trustlet better than the OS. The challenge is that the TEE can hardly model a trustlet's behaviors or assess how much the OS already knows about the trustlet.

Our idea is for the TEE to replay file traces pre-recorded from the very trustlet to be protected. The file traces hence form a tight envelope of the trustlet's actual file activities. *Enigma* provides support for developers to collect file call segments and for the TEE to produce an unbounded stream of sybil calls at run time.

Results By constraining lightweight modifications to generic subsystems and interfaces, Enigma eschews heavy internal changes to individual filesystems and works with unmodified EXT4 and F2FS, reusing over 60K SLOC filesystem-specific implementations. Through a study of six diverse trustlets from which we collect over 200K file calls through testing, we show *Enigma* is practical to deploy and effectively hides the file activities that leak trustlet secrets.

Enigma provides the following guarantees: (1) Against random guess attacks: the probability of a successful guess is 1/K; the successes of individual attacks are independent. (2) Against an external, persistent observer: the maximum period of continuously observing any filesystem image is T. Both K and T are user-configurable.

On a low-cost ARM board (RaspberryPi 3) running 20 concurrent filesystem images, *Enigma* incurs 2.2× access slowdown and consumes 25% additional disk space (with 1.5 MB per sybil image on average); with as many as 50 concurrent filesystem images, *Enigma* incurs 3.9× access slowdown and 37% additional disk space.

Contributions This paper presents *Enigma*, a novel mechanism that generates credible, rich sybil file activities at low cost. *Enigma* contributes the following new designs:

• Sybil filesystem images emulated with only their metadata, which makes strong deception with numerous sybil file activities affordable.

• Continuous shuffling of filesystem identities, which prevents an external observer from collecting long histories of file activities, sybil or actual.

• Replaying file call segments recorded from the trustlet under protection, which effectively deceives a knowledgeable observer.

For a TrustZone TEE, *Enigma*'s deception approach opens the door to using more untrusted external OS services.

6.2 Motivations

6.2.1 TrustZone and its file services

TEE secure storage Arm TrustZone statically partitions an SoC's physical memory and IO devices between normal world and a secure world (i.e. TEE) [348]. The TEE can isolate storage medium, e.g. a SD card or a flash partition, from the normal world OS. The resource partitioning differs from Intel SGX where memory is mapped to TEE dynamically and the OS controls IO hardware.

TEE needs file services Mobile/embedded devices produce and store security-sensitive data such as user health logs and audios samples. TEE can isolate sensitive data from high-risk software such as the OS, for which TEE needs a modern filesystem to keep the data persistent. For instance, journaling [349] prevents data corruption, which is not uncommon on battery-powered smart devices; wear-leveling extends flash lifespan [350], which is key to flash longevity as IoT devices or their flash can be difficult to replace.

Unfortunately, modern filesystems are complex, making them unsuitable to run within the TEE. First, a modern filesystem has substantial code. For instance, EXT3/4 and F2FS have 35K and 17K SLOC respectively. They would significantly bloat TEE's trusted computing base (TCB), e.g. the popular OPTEE which only has around 25K SLOC. Filesystem vulnerabilities [351, 352] then become attack vectors against the TEE. Second, lifting-and-shifting a modern filesystem to TEE requires tedious effort. Not only the filesystem but also extensive kernel APIs must be ported, e.g. VFS, page allocation, and workqueues. Trimming the filesystem code for TEE is error-prone, for which developers need to thoroughly understand filesystem logic and test rigorously. Maintaining a filesystem's separate versions for OS and TEE complicates distributing security updates and patches, which may give rise to a fragmented ecosystem.

For these reasons, forwarding file calls to the OS is a common practice of trustlets.

6.2.2 The Linux storage stack

Our design exploits the following OS storage features.

The stack layers At the top of the stack, the virtual filesystem (VFS) is a filesystem-agnostic frontend receiving filesystem calls, such as read or write, from filesystem clients. VFS dispatches filesystem calls to concrete filesystem implementations; VFS also caches recent file access. A filesystem translates the file calls to disk requests, e.g. block read/write, and submits the requests to an underlying block layer.

Filedata vs. Metadata A filesystem's all on-disk state constitutes its *image*. The image consists of filedata as user contents and metadata which describes the file and the filesystem. Common metadata examples are inodes, directory structures, and block maps. Metadata often constitutes a small fraction of filesystem image, a premise to be tested in Section 6.8. To execute a file call, a filesystem often examines the metadata, e.g. reading inodes of a file in order to locate the disk blocks.

6.2.3 The attacks

Threat model We trust the software in TEE. Both the TEE's file contents and file activities may expose its secret. The file activities are driven by TEE software only and *not* by untrusted entities, e.g. a normal-world app communicating with the TEE. The filedata and file/directory names can be encrypted by the TEE.

The OS hosts a filesystem for the TEE. The OS is:

• *Curious.* The OS probes the TEE's secrets passively and actively. (1) It monitors the TEE's file activities, including file calls, disk requests, data move, and timing of these activities; (2) it inspects a filesystem's in-kernel state; (3) it may deviate the filesystem logic to request disk reads or writes.

• *Knowledgeable.* The OS knows the sequences of file calls that the trustlet may issue.

• *With unbounded memory.* The OS can memoize the full histories of file calls and disk requests it has ever observed. Following the TrustZone convention [353], we deem hardware attacks (e.g. bus snooping) and their side channels out of scope.

Side channels through file activities A file call exposes the following information that cannot be easily obfuscated: file call types [354]; accessed file paths, in particular the relative location in a directory tree; arguments, e.g. sizes, offsets, and flags. A trace of file calls is known to leak the caller's secret [15, 355, 356].

We identify three common side channels from file traces: (1) *Occurrence*: the events that a trustlet access files; (2) *File paths*; (3) *Access patterns*: the combination of access offsets, sizes, and flags in a sequence of file calls.

Trustlets & attacks We motivate our designs with the following trustlets, including their side channels and secrets. Table 6.2 shows a detailed summary.

• *Databases* for embedded environments such as SQLite manage on-device user data [357]. Prior work shows a database's file access patterns depend on queries [15, 356, 358]. For instance, given a database's schema and a sequence of file offsets, the OS can learn a query's secret: read 64 bytes at offset 0 and read 128 bytes at offset 4096 gives away page-align predicate columns (e.g. "user physical activities") and the rows selected (e.g. "hours when the user is sleeping").

• *Fulltext search* is for on-device QA over emails or messages [359]. Given a keyword, an engine reads a binary index file, locates file offsets where the keyword appears, and reads in contextual lines. From the access offsets, the OS can infer the secret: the keyword and the hit locations [356].

• *Model loading.* A trustlet loads a neural network model from a file, for which it may issue thousands of file calls to read and parse model layers. As reported by prior work [360] and verified by us on TensorflowLite [235], from the file path and offsets the OS can learn the secret: the loaded model.

• *Video surveillance.* A trustlet on a camera detects video events of interest, e.g. motion or vehicles, and saves video frames of interest. The OS observes file writes and learns the secret: occurrences of events being detected.

• *Data historian.* A trustlet on a robot collects sensor messages to a ROSBag file [361]. The messages are of variable lengths depending on sensor types (e.g. point clouds and sound samples) as well as data contents (e.g. a point cloud's density). From the write sizes, the OS can learn the secret: sensor types, data contents, and hence the robot's activities.

• *Credential manager.* A trustlet loads one of multiple key files for authentication with a remote server. From the file path, the OS can infer the secret: the loaded key, which corresponds to user identity (e.g. a specific private key) or server identity (e.g. a specific CA certificate).

6.2.4 System Overview

Figure 6.2 shows the system architecture.

Initialization With secure IO, the TEE isolates the physical disk and exposes K virtual disks to the normal world. From the OS's perspective, a virtual disk is no different than a physical block device, except that the OS submits block requests to the TEE. The TEE requests the OS to initialize K images with the same filesystem implementation on the K virtual disks.

Operation As a trustlet emits a stream of actual file calls, the TEE generates additional K-1 streams of sybil calls ① by replaying pre-recorded file call segments (§6.5). The TEE sends the actual file calls to the actual filesystem image and the sybil calls to their respective sybil images.

The OS runs K filesystem images with an unmodified filesystem implementation (although the kernel's generic storage subsystem is lightly modified, see Section 6.6). At the end of each file call execution, the OS submits disk read/write requests to their corresponding virtual disks in the TEE ②. The TEE omits all *filedata* accesses from the sybil filesystem images and only executes their *metadata* accesses ③, which is

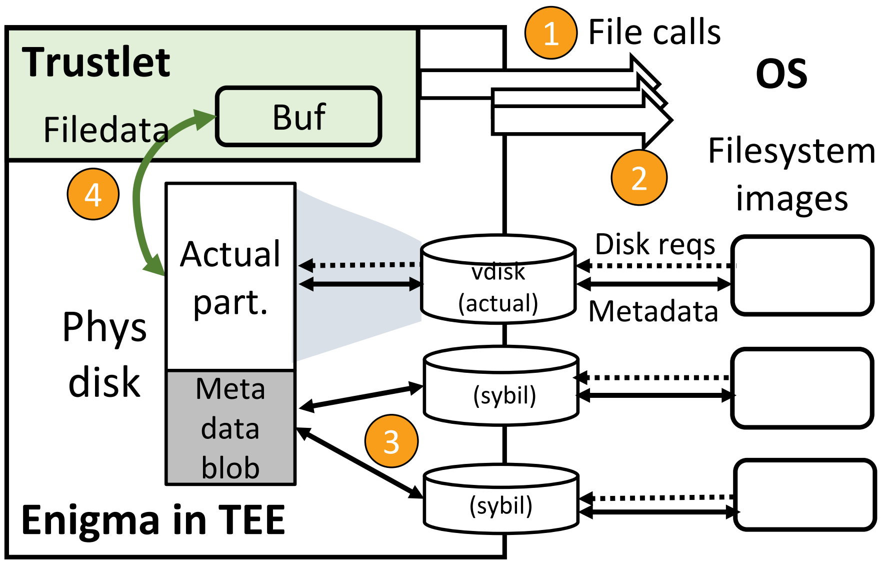


Figure 6.2: The Enigma architecture, showing the filedata/metadata separation and the physical disk layout

needed by the filesystem logic. The TEE periodically unmounts filesystem images, shuffles the virtual disks backing them, and remounts the images on the shuffled virtual disks.

To probe the virtual disk internals, the OS will attempt to: (1) read filedata. (2) tamper with filedata. (3) measure delays of disk access. The TEE implements mechanisms to block these attempts.

What the OS can and cannot see? The OS sees K virtual disks exported by the TEE, on which K filesystem images are mounted. It does not know which image is actual.

The OS sees K streams of file calls sent to the K images. In each stream, the OS can see all file calls in the clear, including their types and arguments. The OS cannot access filedata referenced in file calls. The patterns of file calls fit in the OS's prior knowledge about the trustlet's file activities. All streams show similar statistics, including throughputs of file calls and bytes read and written.

To execute file calls, the OS can freely access metadata (e.g. inode) on virtual disks but not filedata, as such attempts are blocked by TEE; the delays in accessing metadata are the same across all virtual disks.

From time to time, the OS sees: the TEE unmounts some images and takes some virtual disks offline; the TEE puts online new virtual disks with random names. The OS cannot associate any new virtual disks to those disappeared.

Limitations To simplify security reasoning, we consider that one actual filesystem image is exclusively used by one trustlet; concurrent trustlets use separate filesystem images.

6.3 Sybil images with covert emulation

We seek to minimize the cost of sybil images so that *Enigma* can afford a lot of them for strong protection.

6.3.1 Metadata-only sybil images

The first question is what are the minimum disk requests for maintaining a sybil image? We exploit a filesystem invariant: a filesystem only relies on its *metadata* to function properly (e.g. to read or update inodes) [362]. Note that the filedata/metadata division in a modern filesystem can sometimes be ambiguous, for which we will describe treatment (§6.6). *Enigma* therefore enforces that the code of filesystem and OS only access metadata content and its access status (e.g. write completion), but not those of the filedata.

The second question is how to hide the fact that TEE only stores metadata for sybil images? The TEE conceals the physical disk from the OS using the TrustZone's secure IO (Section 6.2). Note that the OS still manages insecure storage devices out of TEE. As shown in Figure 6.2, the TEE directly backs the actual image with a contiguous physical disk region. This preserves the locality of actual disk accesses. The TEE stores the metadata for all sybil images in a single binary blob. To save disk space, it compresses the metadata with copy-on-write (CoW). CoW is effective because the sybil images are mutated by file calls from the same trustlet and are likely to share similar metadata.

We next describe the OS's probing attacks against the covert emulation, as well as our defense.

6.3.2 Isolating filedata paths

Threat: Knowing sybil filesystems contain only metadata, the OS submits disk read requests for filedata on virtual disks; it knows the disk as sybil if no data comes out.

Defense: TEE mitigates the threat by isolating the filedata path, which keeps the OS oblivious to *if* filedata disk requests are actually executed on disk, or *when* such requests are completed. The isolation is as follows.

1. The filedata path: trustlet ↔ disk. Filedata flows between a trustlet and the physical disk without leaving the TEE. As shown in Figure 6.2, a trustlet makes file calls with opaque references of filedata buffers (e.g. "read from offset 10 to buffer <a 64-bit int>") (①); the OS executes the file calls and issues disk requests containing these opaque references (②); the TEE receives the disk ops, maps the opaque references back to the filedata buffers, and moves the filedata between the trustlet and the disk (④) without going through the OS. To prevent fabrication, each opaque reference is a 64-bit integer and used one time only [363, 314].

2. The metadata path: OS ↔ disk. Metadata flows between the OS and the TEE's disk. As shown in Figure 6.2 (③): the filesystem code generates disk requests for metadata (e.g. "write to an inode at block 42"), which contains cleartext references to OS buffers. The TEE executes the requests and copies metadata

between the TEE and the supplied OS buffers. The TEE notifies the OS of metadata access completion. The TEE never examines metadata or takes any action based on the metadata content.

6.3.3 Rejecting OS access to filedata

Threat: The filesystem code, by design, may rightfully repurpose disk blocks without notifying the underlying disk. For instance, after flushing the redo log (metadata), EXT4 may store filedata to the underlying blocks [364]. A malicious OS may tamper with the repurposed filedata blocks, breaking filedata integrity and/or revealing the identity of sybil images (i.e. those containing no proper filedata after filedata writes).
Defense: The TEE rejects OS from accessing filedata. To do so, it tracks filedata blocks on the physical disk and keeps the filedata/metadata dichotomy up to date.

Without requiring intimate filesystem knowledge, TEE enforces a simple policy: *allow the OS to only read back disk blocks it previously wrote to.* When a block is written initially, the TEE tags the block as "metadata" or "filedata" depending on whether the written data comes from an OS buffer or a trustlet buffer. The TEE accommodates block repurposing: in case data is written from an OS buffer to a "filedata" block, the TEE erases the block so no existing content is leaked, changes its tag as "metadata", and grants the access.

Some filesystems may inline filedata in metadata blocks for efficiency, e.g. EXT4 may inline files smaller than 160B. The TEE tags these blocks as "mixed" and tracks inlined filedata ranges. As a result, the TEE allocates disk space for inlined filedata on sybil images, which incur less than 1% of the space overhead in our measurement.

6.3.4 Defense against timing attacks

Threat: The OS measures disk delays of metadata access. Since a sybil image's metadata is stored more compacted than the actual image, the OS may see lower access delays.
Defense: The TEE pads delays of metadata access so that the OS sees uniform delays. Note that it does not have to pad delays for filedata access, the completion notifications for which bypass the OS as shown in Figure 6.2.

The TEE delays each metadata access to all filesystems to be longer than most (e.g. 99%) of the actual delays, a value determined by profiling when filesystem is mounted. Delaying is practical for two reasons. i) Metadata accesses only constitute a small fraction of all disk access. ii) The actual access delays on embedded flash show low variation because the storage has limited internal buffering. We will show timing side channel reduction and overhead in evaluation.

6.4 Filesystem identity shuffling (FIDS)

TEE obfuscates file call histories in the spirit of moving target defense [347]. This is because an OS collecting long histories of file calls poses two threats. (1) The OS is more capable of attacks. It can uncover filesystem identities by reasoning about the histories. (2) The OS, if accidentally discerns, the actual filesystem, creates higher damage. It learns all the actual file calls in the past and in the future.

6.4.1 The mechanism

Critically, FIDS follows an *egalitarian* principle. The TEE frequently shuffles identities of all filesystems, sybil and actual. A filesystem image's OS-visible state is $\langle v, M\rangle$: v is the name of virtual disk that the image is mounted on; M is the metadata, which is exposed to the OS by design.

Shuffling. Shuffling prevents the OS from connecting segments of file call histories. It is performed on a set filesystem images $\{\langle v_1, M\rangle, ..., \langle v_n, M\rangle\}$ that currently have identical metadata M, where n is the number of filesystems. TEE assigns each backing virtual disk a new, random name: $\{\langle v'_1, M\rangle, ..., \langle v'_n, M\rangle\}$. Since the OS cannot connect new disk names v'_i to the old names v_i, it cannot connect new filesystem identities to the old ones.

TEE triggers shuffling by time (e.g. an image has not participated in shuffling for a period of T) or by activities (e.g. the image has served N file calls since its last shuffling). Section 6.8 will evaluate the impact of T, and show that even in the worst case (i.e. accidental filesystem identity exposure), a practical T (e.g. a few seconds) leaks no significant secrets of the trustlet.

Forking. Forking keeps shuffling going when all images have distinct metadata. In case TEE attempts to shuffle an image $\langle v, M\rangle$ but no other images have metadata M, TEE creates an image with metadata M and shuffles the two images, resulting in $\langle v', M\rangle$ and $\langle v'', M\rangle$. The OS cannot connect either of the two new identities to the old image $\langle v, M\rangle$.

Retiring. TEE deletes an image to reclaim its blocks.

The mechanism is inexpensive. First, shuffling and forking only manipulate *references* to metadata, not metadata itself nor filedata. Section 6.6 presents more details. Second, TEE does not need to execute forking often, as there often exist abundant images with the same metadata. For instance, a read-most workload only mutates metadata occasionally.

FIDS operations are visible to the OS, e.g. after mounting, the OS knows images with identical metadata may have been forked. FIDS, however, does not leak filesystem identities, because both actual and sybil images can be forked and shuffled; although the actual image cannot be retired, *Enigma* ensures that such a behavior does not leak identity, as will be discussed in §6.4.3.

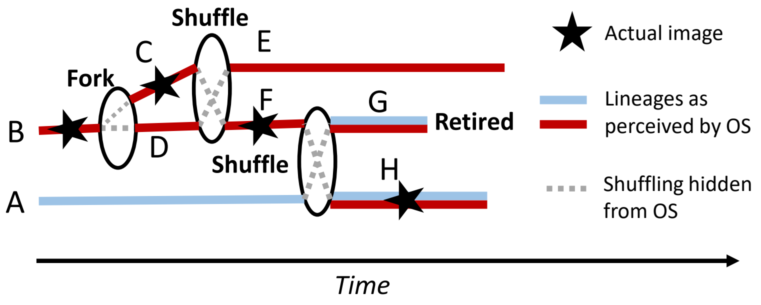


Figure 6.3: A minimal example of filesystem identity shuffling. As time goes by, the OS perceives multiple lineages as actual but cannot distinguish them

6.4.2 Why FIDS works

We use *lineage* to describe an image's OS-visible history. A lineage starts with one of the initial K images. It includes descendant images created by forking and *OS-perceived* descendants created by shuffling. Figure 6.3 shows an example: the first forking on image B creates two descendant C and D; the subsequent shuffling of C and D result in images E and F. C–F all belong to the lineage of B. After shuffling A and F, the resultant G and H belong to both the lineages of A and B.

FIDS limits continuous observation The OS can track the history of any image (e.g. by tagging them with unique metadata), but only for a continuous period no longer than T. This prevents the OS from collecting large number of samples (e.g. a few thousand [365]) and building statistical models for filesystem images, thwarting template attacks. Consider the actual image B in Figure 6.3. Without FIDS, B has a straight-line lineage and its full history of file activities are exposed to the OS. While FIDS does not change the true history of B (annotated with ★) which is only known to TEE, it makes the history appear uncertain to the OS. Forking adds branches to B's lineage; shuffling merges and then splits lineages. OS can only see a lineage tree that clouds over the actual history.

FIDS confines damage of identity exposure Assume the OS, via an unexpected channel, discovers image B as actual. Without FIDS, the OS can track backward in time and reveal all the past file calls on B; it can track forward to learn all future calls on B. All the actual file calls are hence leaked. FIDS prevents the OS from backtracking no further than the most recent shuffling event; earlier than that event, all the images participating in shuffling become the probable ancestors of B. Similarly, the OS can forward track no further than the next shuffling/forking event on B. Section 6.8 will quantify the resultant uncertainty to the OS.

6.4.3 Defense against extinct lineage attacks

Threat: While forking and shuffling are egalitarian, retiring cannot be: TEE never retires the actual image, giving the OS a chance to weaken or break the actual image's anonymity. If the descendants of an earlier image X have all retired (i.e. an *extinct* lineage), the OS can rule out X from being actual.

Figure 6.3 shows an example: if the TEE retires image A prior to its renaming with F, the only remaining lineage is the one from B, which the OS can deem as the actual. A successful OS pinpointing the actual image thus exposes the filesystem identity as described above.

Defense: TEE picks images to retire by respecting two invariants: (1) at any moment in history, there are always K alive *lineages* stemmed from the initial K images; (2) the retiring event leaves at least K images alive. The first invariant ensures that the OS's backtracking cannot rule out any of the initial image from being actual. The second invariant ensures the strength of anonymity at any time in history. Both invariants combined, the OS cannot find a time in the past when there were fewer than K lineages.

To enforce the two invariants, a challenge is to avoid memoizing all FIDS events which grow unbounded. The TEE implements a simple rule: avoid retiring an image if it is the last surviving image on a lineage. To do so, the TEE only keeps K tags: tagging each image with its ancestor as one of the K initial images and propagating the tag to descendants.

The above design will not result in too many images of which none can be retired. As long as there are more than K images, there are multiple images belonging to the same lineage; retiring any will satisfy both invariants above.

6.5 Generating sybil file calls

We ensure that TEE generates sybil file calls close to what the trustlet would actually issue, from which the OS is unable to discern the actual file calls. The objective is *not* to match the actual file traces in deployment, but to generate diverse trace segments that provide strong cover.

6.5.1 Design

First, how to fit sybil file calls in a trustlet's envelope of file activities? For instance, a database may show a variety of file access patterns depending on queries; one pattern can be "read 8 bytes, skip 16 bytes, and read 42 more bytes". If a stream of file calls do not show any such pattern that the OS knows must exist in actual file calls, the OS can determine the stream of file calls as sybil. However, it is difficult for *Enigma* to model a trustlet's file activities or assess how much the OS knows about the activities.

Our solution is to exploit the knowledge already encoded in a trustlet: the TEE replays historic file traces from the trustlet to be protected. To this end, developers exercise the trustlet with a set of inputs and record file traces during the execution, which we will show in Section 6.5.2.

Second, how to generate sybil calls that provide strong protection? Our insight is that the efficacy of sybil calls hinges on *the set of plausible secrets* they represent as cover traffic for the *actual* secret; we measure such efficacy as the set's cardinality and entropy estimation [366]. Intuitively, a library of sybil calls would offer stronger anonymity if the library represents more plausible secrets and these secrets are uniformly distributed in the space of secrets.

To quantify the set of plausible secrets, we exploit an observation: a trustlet's file calls are driven by input events [357, 367], which are associated with the trustlet's secret. Therefore, we retrofit the idea of viewing a file trace as independent segments, where each trace segment encodes a secret value. For instance, the file trace of a database can be segmented per query and each segment encodes a secret $\langle C, R \rangle$: a query's predicate columns (C) and its selected rows (R).

With the above rationale, *Enigma* assists developers to collect sybil trace segments. It requires the developers to (1) provide annotations for segmenting file traces, e.g. by input events; (2) provide test inputs, such as concrete database queries; (3) annotate the inputs with plausible secrets they represent. A test harness exercises the trustlet and records the resultant trace segments. It reports cardinality and entropy of the current secret set [366] and makes suggestion towards improving them. The developers finish collection when they are satisfied with the metrics.

For example, to collect trace from a database trustlet, the developers provide as input a set of queries, each annotated with a secret $\langle C, R \rangle$ for the query. The set of plausible secrets is therefore $\{\langle C_1, R_1 \rangle, \langle C_2, R_2 \rangle, ...\}$. After running a batch of queries and collecting the trace segments, the test harness suggests to increase the secret set entropy by running more queries that select more diverse columns.

Third, how to replay the segments? TEE replays by sampling from a library of trace segments; it preserves both the pre-recorded order and arguments of file calls within a segment. As it runs, it gradually renews the pre-recorded segments with segments (both sybil and actual) collected in deployment. As a result, the sybil traces evolve to be even closer to what the trustlet is issuing in deployment. We further address the following issues. *(1) Time the emission of segments.* TEE emits at random intervals so that sybil file call throughput and read/write throughputs approximate that of the actual trace. *(2) Delay between file calls.* To prevent timing side channel, the TEE uniformly pads all the intervals between file calls to the maximum interval it has observed. *(3) Make sybil calls consistent with filesystem images.* The TEE adjusts sybil calls before replay, for instance, to create files, to truncate the offsets of out-of-bound access, to redirect access from non-existing files.

6.5.2 Case study

We study the trustlets in Table 6.2. Our input for recording should be seen as examples; developers are likely to have inputs better matching their deployment, e.g. queries from their deployed databases or logs from their robots.

Database. We run SQLite on a database of user health activities with 3 columns in numeric types. We run a suite of queries [368] and segment file traces by query. We collected 500 trace segments constituted by 15K file calls.

FullText. We run Lucy, an embedded search engine [369] over 2GB of emails [370]. The inputs are 100 searches for top keywords. We segment file calls by search. The collected 100 segments consist of 100K file calls.

ModelLoad. We run TensorflowLite. Our inputs are 10 sample NN models loaded for inference. We segment file traces by each model load. We collected 100K file calls in 10 segments.

VideoEvent. We run an OpenCV motion detector. Our inputs are 100 hours of street camera videos in Bangor [371]. We segment file calls by per video hour and have collected 9K file calls in 100 segments.

Historian. We run the ROSBag drive data historian with 10 different drives from the autonomous driving dataset [361]. We instrument the run script to segment file calls by per test drive. We have collected 9K file calls in 9K segments.

CredLoader Our test script generates 50 key files and invokes the Openssh client on these key files for login. We instrument the test script to segment file calls by each login attempt. We have collected 300 file calls in 100 segments.

6.6 Implementation

We implement *Enigma* in 2K SLOC, atop OPTEE and Linux as summarized in Table 6.3. Of the code, 1K SLOC is for modifying the generic kernel page cache and block subsystem; filesystem-specific code incurs only less than 50 SLOC of changes. In another 1K SLOC, we use the MMC driverlet inside TEE [19], which provides read/write functions sufficient to our needs. We use a 32-GB microSD for storage and partition it into two: one is 4GB and used by the untrusted OS as its rootfs; the other is managed by *Enigma* as the isolated physical disk. We next describe implementation details of Enigma – how we apply lightweight instrumentations to generic kernel subsystems and avoid heavy modifications to individual filesystems.

TEE $\leftrightarrow$ OS interfaces We instrument two interfaces for communicating between TEE and OS. At them, we inject SMC instructions and handle world switches.

1. TEE → OS. Via the interface, *Enigma* issues file calls (actual and sybil) to OS. To this end, we instrument filesystem syscalls (e.g. generic_perform_read) at VFS layer. We modify the data buffer pointer of the interface (i.e. iov_iter) to pass opaque references pointing to in-TEE buffer addresses (§6.3.2) instead of userspace addresses; we further preserve them in the kernel page struct, which we will describe shortly.

2. OS → TEE. At the filesystem bottom, we instrumented the block IO (bio) interface (e.g. submit_bio). It dequeues bios to TEE; when it does so, we retrieve the opaque references from the page struct pointed by the bio and pass them back to TEE. We also modify bio callbacks to let the filesystem execute asynchronously w.r.t. TEE invocations and disable bio merging.

Isolating filedata path While being conceptually independent of each other, filesystems work closely with kernel memory management (MM) layer. For instance, a filesystem is also responsible for reading filedata into the page cache, and in coordination with the MM layer, writes dirty pages (filedata) back to the disk.

To isolate the filedata path described in Section 6.3.2, we disengage the page cache layer as follows. We first modify the page cache allocation methods (e.g. pagecache_get_page) to preserve the opaque references in page cache. By design, OS must allocate page cache before manipulating data pointed by foreign addresses (e.g. by userspace or opaque references). At TEE → OS interface when OS allocates pages, we tag all newly allocated pages and inherit opaque references in their page struct. We then block kernel attempts to copy from/to opaque references. To do so, we associate the tagged kernel pages (i.e. contain opaque references) with pre-allocated user pages which only have dummy filedata, and direct all accesses to tagged pages to them. With all changes reflected on tagged kernel pages and dummy user pages, OS is oblivious to opaque references and makes decision based on its intact logic (e.g. whether to flush dirty pages). This transparently bypasses the page cache layer without disruptive changes.

As a result of the above modifications: 1) on the filedata write path, OS allocates kernel pages which preserve opaque references, and copies dummy user pages (i.e. filedata to write) to them. After filesystem execution, OS generates bios whose filedata points to these kernel pages. It then submits the bios to TEE, returning opaque references. 2) on the filedata read path, it is a mirror process.

Block translation tables A bio request received by the TEE carries a buffer address and a virtual block number. The latter is translated to a block number for the isolated physical disk. The TEE does the translation by consulting with its per-image block translation tables (BTT). A BTT maps an OS-visible virtual block number to a TEE-visible physical block number. BTT entries are only for metadata blocks. Filedata block numbers do not need translation – they are either directly mapped to the physical disk or discarded.

BTTs reduce the cost of manipulating sybil filesystem images. (1) Much of FIDS becomes BTT operations.

To fork an image, the TEE duplicates its BTT without duplicating the disk blocks. To shuffle two images, the TEE unmounts the images, shuffle their BTTs, and re-mounts them. To retire an image, the TEE frees its BTT. (2) The TEE implements CoW by setting BTT entries of identical metadata blocks pointing to the same physical copy. When any shared disk block is written to, the TEE allocates a new disk block and updates BTT entries for all filesystems.

Store BTTs securely The TEE stores encrypted BTTs in the normal world. There are two reasons: 1) BTTs should enjoy equal confidentiality and reliability as user files; 2) storing BTTs on an in-TEE filesystem (with crash consistency, etc.) would defeat our goal of leaving filesystem out of TEE.

Outsourcing BTT storage leaks no secrets: BTT lookups are driven by disk requests submitted from the OS; the input block numbers are from the OS; the output block numbers will not be decrypted until they are in the TEE. The TEE updates BTTs only in an egalitarian fashion. In shuffling filesystem images, the TEE re-encrypts all their BTT entries. To allocate a new disk block for a virtual block, the TEE re-encrypts BTT entries corresponding to the virtual block of *all* filesystems.

Metadata vs. filedata The following details are from real-world filesystems. (1) Because filesystem logic needs to access directory contents, *Enigma* treats directories as metadata, although they may be implemented as special files by some filesystems. (2) *Enigma* treats a journaling filesystem's journal as metadata. By default, common journaling filesystems write dirty metadata (e.g. inodes) to their journals. As an expensive option, they can write filedata to journals for stronger consistency. In our current implementation we turn the option off. (3) Some OS functions may read filedata, e.g. exec() will parse the header of an executable file. These functions, however, are not supposed to be invoked on TEE-owned files (e.g. OS should not exec() a TEE program file). TEE can safely reject the read attempts.

6.7 Security analysis

	Attacks	Defense/mitigation	C	I	A
Passive	Observing file activities	Sybil filesystems with credible activities §6.3 & §6.5	✓	-	-
	Observing filedata move	Isolating filedata path §6.3.2	✓	-	-
	Measure disk timing	Delay padding §6.3.4	✓	-	-
	Learning history of file activities	Filesystem identity shuffling §6.4	✓	-	-
Active	Dropping file calls	TEE checks file integrity [297]	-	○	○
	Dropping (un)mounting requests	TEE checks filesystem integrity [372]	-	○	○
	Unsolicited reads from filedata	Reject OS access §6.3.3	✓	-	-
	Unsolicited writes to filedata	TEE erases existing filedata §6.3.3; detect by file check [297]	✓	○	-
	Unsolicited writes to metadata	TEE not touching metedata §6.3.2; detect by filesystem check [372]	✓	○	-
	Supplying wrong block numbers of filedata	TEE checks file integrity [297]	-	○	-
	Fabricated references of filedata	Strong opaque references §6.3.1	✓	✓	-

Table 6.1: *Enigma* thwarts attacks against confidentiality. **C:** Confidentiality, **I:** Integrity, **A:** Availability. ✓**:** Attack thwarted, ○**:** Attack detected **-:** Not targeted by the attack

6.7.1 TCB

Enigma keeps substantial OS code out of the TEE: 37K for EXT4, 22K for F2FS, and 37K for a block layer, as reported by SLOCCount [373] in kernel v4.19. The *Enigma* runtime only adds 1K SLOC to the TEE and its replay-based MMC driver adds another 1K SLOC. The TEE exports two interfaces to the OS, for issuing file calls and for receiving disk requests. Through the two interfaces, the normal/secure workloads share no state. Following a common practice [374], the TEE passes messages with arguments packed into CPU registers during world switches, minimizing the risks of data leak. The only input data to the TEE is metadata. The TEE is secure against invalid or malformed metadata: 1) the TEE simply moves the metadata between OS buffers and the physical disk; it never touches the metadata; 2) the TEE sets the backing memory to be non-executable.

6.7.2 Security guarantees

Against random guess attacks In each attempt, the attackers randomly pick one of K filesystem images and infer secret based on the file calls on the image. They break the obfuscation if they either hit the actual image, or hit a sybil image that coincides with the actual secret. Hence the probability to break *Enigma* is $\frac{1}{K} + \frac{K-1}{K}(1/N)$, where N is the number of plausible secrets represented by the trace library. As developers generate traces towards an N much higher than K (§6.5.2), e.g. N=500 secrets for *Database*, the above probability is close to $1/K$.

Section 6.8 will compare *Enigma* to ORAM when they provide same probabilistic guarantees.

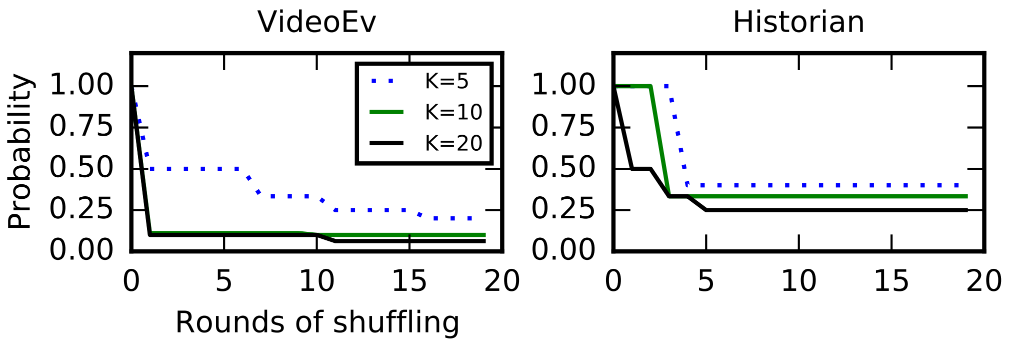


Figure 6.4: FIDS diminishes probability of OS distinguishing an image's history from histories of other images. Write-most benchmarks are shown. On read-most benchmarks the probability converges even faster

Against observation of file call histories We measure P: the probability that the OS pinpoints the history of a given filesystem image F. P is reciprocal to the number of images (M) that have histories indistinguishable from F to the view of the OS, due to shuffling (§ 6.4). We model M as a set cardinality:

$M = |\bigcup_{i=1}^{r} S_i|$, where r is rounds of shuffling, S_i is the set of filesystems participating in i-th shuffling round; $M \in [1, K]$.

We study how P changes as a function of shuffling round r. In the above form of P, S_i depends on the set of filesystem images having identical metadata. Therefore, we run all benchmarks with varying K, collect their traces, and report the average P as a function of r. The results are shown in Figure 6.4. In general, as an image participates more rounds of shuffling, P drops from 1 (OS is certain about its history) to lower values (less certain). P drops faster with a larger number of images (higher K), because of richer shuffling opportunities and thus faster increase in uncertainty.

Against disk timing side channels We quantify the mitigation (§6.3.4). We measure the delays during 100 runs of filesystem image initialization, consisting of over 400K physical disk accesses as described. We set the padding threshold to be 1.6 us, 99-percentile of the measured delays.

We then run a filesystem with two versions: (1) *Insecure*; (2) *Enigma* with various paddings. We use *mutual information* (MI), a common notion of information theory used in privacy preservation [375, 376], to characterize the correlation between two random variables: (a) the access timing distribution of 1000 randomly sampled blocks on version 1 (i.e. *Insecure*) and (b) the access timing distribution of the same set of blocks on version 2 (i.e. ours with various paddings). Higher MI implies the higher certainty that the second timings are sampled from an emulated disk.

Our measurement shows a significant MI reduction, from 0.342 bits (i.e. *Enigma* w/o padding) to 0.007 bits. Such a level of residue MI is considered negligible in prior work [375], suggesting it is difficult for the OS to correlate whether a disk is emulated to the observed access delays.

Against other attacks Table 6.1 shows that *Enigma* leaks no filedata or activities to an adversarial OS even when the OS deviates from the filesystem logic or injects malformed metadata. *Enigma*'s confidentiality is susceptible to hardware side channels, e.g. through CPU cache. We rely on existing mitigations [375]. *Enigma* defeats filedata integrity attacks by rejecting OS accesses (§ 6.3.3). *Enigma* can detect availability attacks but cannot prevent them, e.g. the OS powering down the whole device.

6.8 Evaluation

We seek to answer the following questions on *Enigma*:

- How much is the space overhead? (§6.8.2)
- How much is the slowdown in file accesses? (§6.8.3)
- What is the performance impact of FIDS? (§6.8.4)

6.8.1 Methodology

Trustlets & description	IO Char.	Dataset	Access delay	Side channels	Comparisons (Baselines)
Database. Query on-device database of user health data.	Single file Rand RW	select(1-8) benchmarks from SQLite [368]. Total: 500 queries, 14K file calls. File size: 800KB	Per query	Sizes & offsets	ORAM
Fulltext. Search text files for on-device QA.	Multi files Rand RD	Lucy [369] on pre-indexed Enron emails [370]. Total: 100 queries; 100K file calls. File size: 2 GB	Per search	Sizes & offsets	
ModelLoad. Load ML models from files.	Multi files Rand RD	TensorflowLite [235] loading 10 neural nets, Total: 80K file calls. File size: 41 MB.	Load per NN	Sizes & offsets File paths	
Historian. Log data bags from multi. sensors on a robot.	Single file Append	ROSBag on EU Long-term dataset [361]. Total: 36K file calls. File size: 659 MB.	Log per data bag	Sizes & offsets	PadWrite
VideoEv. Log images of motion events detected.	Multi files Seq WR	50 1080P images in Bangor video [371]. Total: 9K file calls. File size: 100 KB	Log per image	Access occurrences	InjectCreate
CredLoader. Load credentials for authentication with servers.	Multi file Seq RD	Load 50 key files generated with ssh-keygen. Total: 150 file calls. File size: 0.9 KB.	Load per key	File paths	InjectFiles

Table 6.2: A summary of benchmarks in *Enigma*.

Setup and metrics Table 6.3 summarizes our test platform. We choose Rpi3 [377] for its good support for TrustZone. Table 6.2 summarizes the trustlets as benchmarks and their traces. In TrustZone, we do not run them but extract their file traces for replay, making benchmarks simple and reproducible. Note that our benchmark programs are for reproducing trustlets' file activities; production trustlets likely have different, more compact implementations, e.g. by linking to embedded libraries. We deliberately diversify file behaviors and file sizes. For each benchmark, we create the smallest disk partition that can accommodate the benchmark files. Note that the smallest disk sizes supported by F2FS and EXT4 are 39 MB and 2 MB.

Filesystem choices We pick two mainstream filesystems that exercise *Enigma* in different ways.

• F2FS is *Enigma*'s reference filesystem. A log-structured filesystem popular on mobile devices, F2FS extensively optimizes for NAND flash [378]. For flash longevity, F2FS allocates blocks on demand and generates compact metadata. We create the test image with mkfs.f2fs v1.11.0.

• EXT4 is our stress test for *Enigma*. A journaling filesystem, EXT4 issues dense metadata writes [379]. Since *Enigma* stores metadata for sybil images, it incurs higher overhead with EXT4. We create the test image with mkfs.ext4 v1.44.5.

Baselines First, we consider *Insecure* which incurs no overhead: the TEE invokes a filesystem image without any protection. Furthermore, we consider protection baselines that specifically hide the side channel of each trustlet. As such, these protections pay no cost for unneeded protection and are therefore competitive against *Enigma*. They are summarized in Table 6.2 with details as follows.

• *ORAM* [15] is the baseline protection for Database, FullText, and ModelLoad. It mitigates side channels due to access sizes and offsets within a single file through obfuscation. By design, ORAM guarantees that the probability of random guesses recovering the file trace is $P = 1/2^{LM}$, where L is the height of ORAM tree

SoC	Raspberry Pi Model 3B+, 1GB	**Normal OS**	Linux 4.19
CPU	4x Cortex-A53@1.4 GHz	**Secure OS**	OP-TEE 3.9

Table 6.3: The test platform used in *Enigma* evaluation.

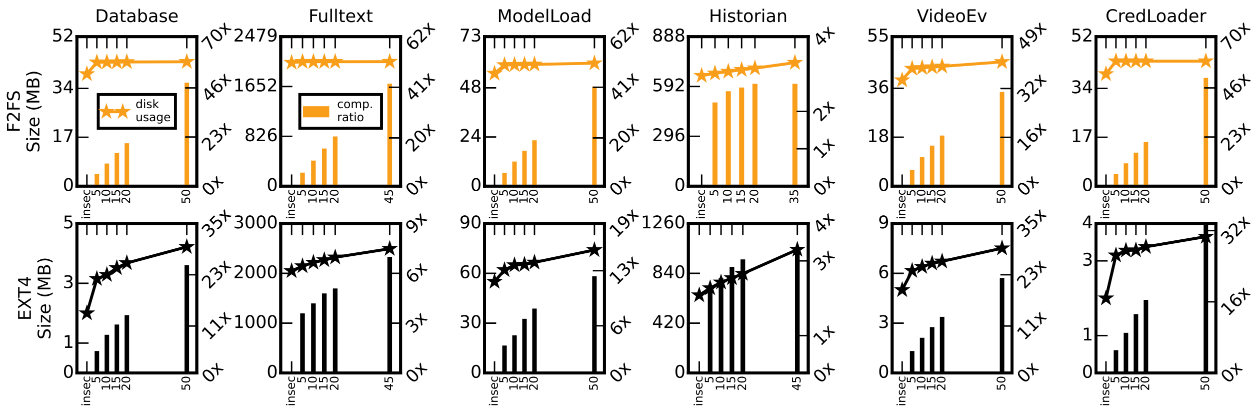


Figure 6.5: Disk usage (lines) and metadata compression ratio (columns). *Enigma*'s usage is modestly higher than *Insecure* and grows gracefully with K on most benchmarks. X axis: number of filesystem images (K). Note that an image of F2FS/Ext4 has a minimum size of 2 MB/39MB by design.

and M is number of accesses in the trace [380]. Since *Enigma* provides $P = 1/K$, we compare the overheads of ORAM and *Enigma* when their guarantees match, i.e. $P = 1/2^{LM} = 1/K$. We set $K = 50$, ($P = 0.02$), the largest number of filesystem images *Enigma* can run on our test platform with limited TEE memory. Since ORAM's L and M must be integers, we choose the closest value $LM = 6$, where M depends on a benchmark's trace segments, e.g. $M = 3$ for *Database*.

• *PadWrite* hides the append sizes for *Historian*. The TEE pads the size of each append to be the largest append the benchmark may issue. No sybil files or calls are injected.

• *InjectCreate* hides file creation occurrences for *VideoEv*. The TEE creates the same number of sybil files as the actual files. The creation times are independent of the actual creation. Since *VideoEv*'s file sizes are not secret, the TEE creates the sybil files with same sizes as the actual.

• *InjectFiles* hides file paths for *CredLoader*. The TEE injects the same number of sybil files as the actual files in the actual image and emits sybil reads to them. Since *CredLoader*'s access offsets within files are no secret, the sybil reads use the actual offsets.

6.8.2 Space overhead

Disk overhead comes from (1) the metadata size per sybil image amplified by (2) the number of sybil images ($K - 1$).

As shown in Figure 6.5, the disk overhead is modest in most benchmarks. Compared to *Insecure*, *Enigma* increases the disk usage by 1%-58% (18% on average) when $K = 5$. When K reaches as high as 50, the disk space of *Enigma* as compared to *Insecure* is 38% on average, which roughly translates to 1% per additional sybil image.

Our experiments show the efficacy of the metadata CoW compression (§6.3). For example, with K=20, turning off CoW increases the disk overhead by 2× on F2FS and 4× on EXT4. When we further turn off discard of filedata, the disk overhead is almost linear to K.

Baseline	ORAM			PadWrite	InjectCreate	InjectFiles
Trustlet	Database	Fulltext	ModelLoad	Historian	VideoEv	CredLoader
w/ F2FS	39	16,800	373	759	39	39
w/ Ext4	11.2	16,800	373	759	5	2

Table 6.4: Disk space (MB) needed by baselines. Note that EXT4/F2FS have the least allowable disk sizes of 2MB/39MB respectively. Figure 6.5 shows the disk usage of *Enigma*.

Comparison with baselines Table 6.4 shows their minimal disk usage. *ORAM* incurs 9× disk overhead, 3× to 9× higher than *Enigma* with $K = 50$, which is consistent with prior ORAM-based file protection [15]. This is because ORAM-based protection must store the whole ORAM tree, several times larger than the address space (file size) to be protected.

For trustlets that can be protected with simple obfuscation, the baselines may use less disk space than *Enigma*. For instance, on *Historian* and *VideoEv* which append to a single file, their disk usage is 6% and 18% lower than *Enigma* with $K = 20$. This is because 1) intensive filedata writes update metadata frequently, which makes *Enigma*'s compression less effective; 2) the metadata on many sybil images exceeds the total size of small files (e.g. 5MB).

Memory overhead of *Enigma* is from storing BTTs and the metadata of sybil images. Such memory consumption grows with K. It is allocated in the normal world only and the stable consumption is modest, e.g. 26 MB and 18 MB for running *ModelLoad* on EXT4 and F2FS when $K = 20$, a small fraction of the 1GB DRAM on our board.

6.8.3 File access delays

We measure delays of file access sequences as defined in Table 6.2. The rationale is the mobile/embedded trustlets are often event-driven and latency-sensitive.

Figure 6.6 shows the results. On most benchmarks, the delays grow gracefully with K. Compared to *Insecure* with only the actual image: *Enigma* with $K = 20$ increases the delays by 1.3×-4.5× (2× on average), showing a sublinear growth. *Enigma* benefits from its elimination of filedata access for sybil images, which discards 95% of disk requests on average. The delay of *VideoEv* does not grow because the TEE issues file calls to images at different times in order to hide access occurrence; these file calls do not contend. When $K > 20$, *Enigma*'s concurrent execution of filesystemes is bound by four ARM cores on our board. For

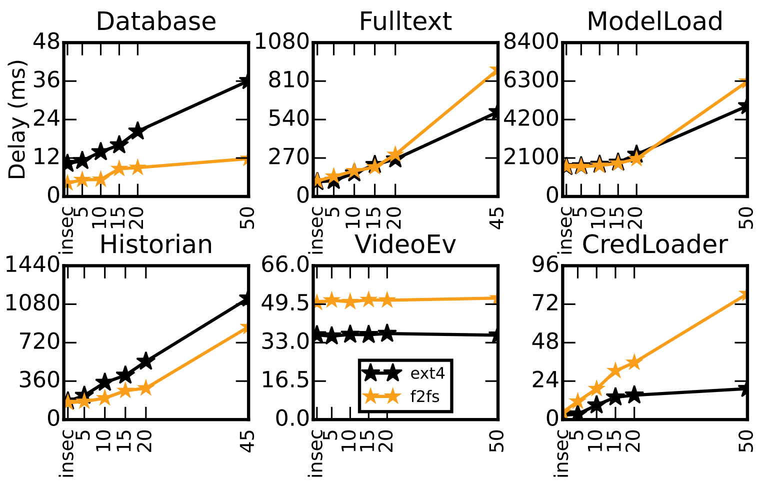


Figure 6.6: File access delays. *Enigma*'s delays are modestly higher than *Insecure* and grow gracefully with K on most benchmarks. X axis: number of filesystem images (K). Delay metrics defined in Table 6.2.

Baseline	ORAM			PadWrite	InjectCreate	InjectFiles
Trustlet	Database	Fulltext	ModelLoad	Historian	VideoEv	CredLoader
w/ F2FS	271	Timeout	283,539	204	50	54
w/ Ext4	289	Timeout	295,281	222	48	31

Table 6.5: File access delays (in ms) by baselines for comparison. *Enigma*'s results are in Figure 6.6

Historian and *Fulltext* on F2FS with $K > 45$, our test board runs out of TEE memory. Note that the delays of normal/secure world switches (i.e. ns) are negligible compared to the disk IO delays (i.e. us).

Comparisons Compared to ORAM, *Enigma*'s delays are lower by $8\times - 70\times$ (on average $37\times$). In contrast to *Enigma* which *discards* sybil filedata, *ORAM amplifies* filedata by tens of times. While a benchmark reads tens of MBs of data, *ORAM* amplifies reads by $18\times$ and adds $20\times$ extra writes. This results in $40\times$ disk IO and saturates our disk bandwidth.

On *Historian*, *Enigma*'s delay up to $4\times$ higher than *PadWrite*. This is expected, as *PadWrite* precisely caters to *Historian*'s append-only pattern and its side channel. Its only overhead is write of additional filedata. On *CredLoader* with F2FS, the delay with $K = 50$ is 63% higher than the baseline *InjectFiles*. The reason is in F2FS's low performance in accessing many small files of this benchmark.

6.8.4 FIDS overhead

Costs of FIDS operations Shuffling and forking do not require data copy or move. Of their delays, 10%–30% comes from BTT manipulation while the remaining comes from stopping and restarting filesystem images as done by the untrusted Linux kernel. With EXT4: 1) forking takes 90ms/180ms on a 64MB/2GB disk, respectively. 2) shuffling two disks of 64MB/2GB each takes 80ms and 130ms, respectively. 3) retiring an

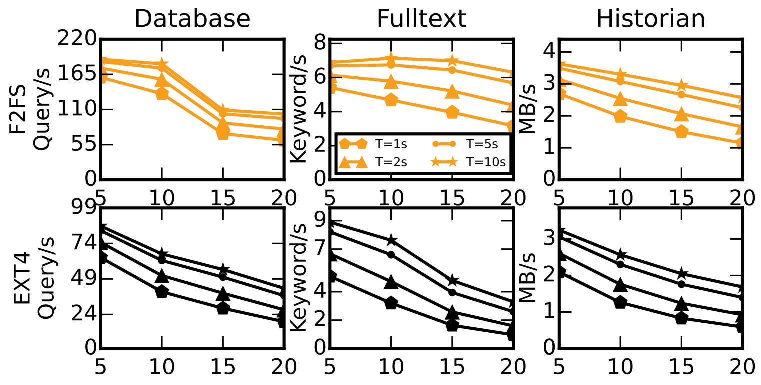


Figure 6.7: Trustlet throughputs under different FIDS intervals (T). X axis: number of images (K).

image takes less than 1 ms. Compared to EXT4, F2FS shows 39% – 66% shorter delays due to its "fastboot" option. Our measurements suggest FIDS efficiency can benefit from further optimization of the Linux kernel, e.g. by parallelizing mounting/unmounting of many filesystem images.

Impact on trustlet throughputs Because by design FIDS is executed in the background off the file access path, we focus on its impacts on a trustlet's throughputs.

Figure 6.7 shows three benchmarks where throughput matters. We chose the intervals based on [381], which reports low-frequency and bursty file activities for mobile/edge device; under such intervals, at most 1-2 secrets may be exposed even in case of accidental filesystem identity leak. We validate their throughputs are bound by disk IO because the throughputs are higher when running them on *Insecure* filesystem. Hence, our test trustlets do not execute app logic, e.g. database code; they execute file accesses as quickly as *Enigma* allows.

Even under strong protection (e.g. $K = 20$; FIDS every second), the benchmarks deliver throughputs appropriate to the IoT/embedded scenarios. *Database* and *Fulltext* can process tens of queries per second and several queries per second, respectively, sufficient to queries driven by a single user. *Historian* can log a few MBs of data per second, which can support a robot's 1–2 HD video streams or point clouds at 3–5 FPS [382]. As the developers relax the protection, the throughputs improve by 1.5× to 2×. Using *Insecure* increases these throughputs by 2×-10×.

6.9 Related Work

Side-channels & mitigations Timing side channels are often mitigated by deploying low-res timer [383], padding delays [375]. We do not focus on them but apply these techniques to mitigate the side channel of our emulated disk. Access pattern side channels (e.g. memory [355, 356], file [15]) exploit data-dependent execution to infer user input (e.g. queries). They are often mitigated by distorting the access pattern (e.g. via ORAM [346, 363, 358]). Motivated by them, we also protect access pattern; unlike them, we preserve the patterns yet hide them under credible sybil ones.

Hide data in plain sight To hide data that must eventually be released, an old wisdom is to add noise [384] as deception. Due to its practicality, recent systems start to adopt it for anonymous location sharing [344], query processing [345]. Compared to them, we are the first to apply it to file services. Another approach is to continuously reset attacker's observations on the data (e.g. ASLR [385] limits attacker's observation on address spaces). We echo its motivations; we deal with filesystem identities (actual vs. sybil), a different domain.

TEE and file services To enable files services for these apps, some include filesystem code inside TEE (e.g. through porting [386], libraryOS [387], build anew [388]). Compared to them, we do not include nor invent filesystem code, instead we take a forwarding approach which reuses unmodified filesystem code. Some forward file calls as we do [297, 389]. In comparison, we focus on the ignored side channel caused by such forwarding. Some exposes to apps a raw block device interface [286], which is backed by a file in normal OS for crash consistency. Similar to it, we also store a backing file in normal OS (i.e. BTTs). Different from it, we store inside TEE a compact representation of filesystems, similar to David [362].

6.10 Concluding remarks

Applicability to SGX *Enigma* may hide the file activities of an SGX TEE (enclave) albeit with different implementation requirements. Unlike a TrustZone TEE, an SGX enclave lacks the capability of direct disk access. Thus, *Enigma* may use a hypervisor to manage disk hardware for the enclave and isolate the disk from the untrusted OS, similar to [390, 391].

Conclusions *Enigma* hides file activities of a TrustZone TEE. With *Enigma*, the TEE generates sybil calls by replaying; the TEE backs only one image with the actual disk while other images with emulated storage; the TEE prevents the OS from learning long history of any image. We build *Enigma* and show that *Enigma* works with unmodified file systems, incurs affordable overhead, and represents a new design point in guarding IoT storage stack. *Enigma* opens the door for a TEE to external untrusted OS services.

Chapter 7

Concluding Remarks

Advances in technology node endow tiny devices *at the edge* (e.g. smart speakers, cameras, phones) with more computing power. Sitting close to user data, they are a preferred place for executing latency- and security-sensitive tasks, without the cloud overhead. Yet, the current edge suffers from poor efficiency and the lack of systems support for hardware security features, hindering the wider adoption of the edge computing paradigm.

This dissertation addresses the above problems and shows a practical path towards designing systems for making an edge device more efficient and secure.

Towards better efficiency The first and foremost question is how to find the ground truth power consumption of individual apps, so they may adapt accordingly? For instance, a 3D game may lower its rendering quality to save power, based on its current power consumption. To answer the question, this dissertation introduces *Power sandbox (Psbox)*, an energy accounting facility implemented as an OS principal. Psbox is the first to pinpoint the power entanglement problem, where the energy cannot be cleanly separated and attributed to co-running apps once it is aggregated. To address the problem, Psbox isolates the vertical environment of the running app at the OS level, insulating its energy impacts from other co-running apps. Provided as a pay-as-you-go OS service, Psbox enables the app running at the edge to know its ground truth power consumption, which it can adapt accordingly towards better efficiency.

This dissertation next introduces Transkernel and STI, two systems which complement each other and jointly improve the runtime efficiency of the edge.

Transkernel achieves so at the kernel level by taming the energy-hungry kernel suspend/resume workloads, a major energy bottleneck in today's smart device due to many small but inevitable idle epochs. Its key insight is to exploit the peripheral cores on the commodity edge device. Compared with the CPU (e.g.

Cortex-A series), a peripheral core is wimpy (e.g. Cortex-M series) but has 1-2 order magnitude smaller idle power, making it an ideal place for executing the idle epochs in suspend/resume process. To this end, Transkernel features a specialized dynamic binary translator (DBT) on the peripheral core, which executes unmodified kernel suspend/resume code. By co-designing the DBT software with the ISA similarity of the heterogeneous cores, Transkernel has low overhead and saves 34% energy. More importantly, Transkernel shows that while cross-ISA DBT is typically used under the assumption of efficiency loss, it can enable efficiency gain, even on off-the-shelf hardware. STI, on the hand, explores an alternative path of improving efficiency at userspace level. Targeting at the NLP inference where large models (e.g. a few hundred MBs) are common, STI pinpoints the design inadequacies of existing runtime in utilizing the trinity of resources – computation, IO, and memory; STI finds that the key resources – memory for preload and IO/compute for model execution – are managed in isolation and lack coordination. To address it, STI proposes to manage the NLP model parameters in finer grained *shards*, which have different importance; to execute inference at runtime, STI instantiates an IO/compute pipeline and judiciously selects the most important shards to maximize the pipeline utilization towards a higher accuracy. STI facilitates efficient NLP inference at the edge: on a comprehensive suite of benchmarks, STI shows that it is possible to achieve a competitive accuracy under a range of practical latencies while consuming 1-2 MB of memory.

Fostering security and privacy This dissertation then switches gear towards the other key inadequacy at the edge – the unusable TEE features. Due to lack of device drivers and filesystems, the trustlet inside TEE must rely on the OS, often through forwarding requests for execution. Doing such critically hinders its intended security, which leaves both data and control path to the OS'es discretion; the latter, if malicious, can peek into and even tamper with the data or control path for sabotaging TEE security, thus making it unusable.

To enable the practical use of TEE, the key challenge is how to enable the use of drivers and filesystems in TEE, which has a large codebase. As an example, an MMC driver framework contains 15K SLOC entangled with over 20 different kernel subsystems, and the EXT4 filesystem consists of over 37K SLOC, which is even larger than the core of OPTEE-OS used in TEE. It is therefore impractical to port them as a whole nor building them from scratch – the former bloats the TCB while the latter throws away years of engineering efforts and likely contain many bugs.

To address the challenge, this dissertation shows how to reuse mature device drivers and filesystem from Linux inside TEE in a practical way, hence facilitating secure and private use of TEE.

Driverlet reuses mature Linux device drivers through specializing the drivers (hence driver"let") with respect to intended workloads, e.g. simple IO functions. By recording the driver/device interactions from the mature Linux driver, a driverlet compactly packs all functions sufficient for driving the state machines on the device, achieving the same IO function as desired by trustlet, e.g. disk read/write, frame capture. Enigma, on the other hand, reuses mature Linux filesystems. It still forwards the file requests as-is but obfuscate them with curated noise which constitutes sybil traces recorded with the trustlet knowledge and waits for the resultant block requests from Linux filesystems; with an in-TEE MMC driverlet, it filters out block requests from noise while only executing those from the trustlet, without being noticed by the OS. Evaluated on a range of comprehensive and representative workloads, both driverlets and Enigma show practical overheads, usable to today's trustlets.

A holistic tapestry Together with the five systems, this dissertation composes a holistic tapestry towards a more efficient and secure edge. It offers solutions and insights into developing kernel, userspace, as well as TEE systems, covering the full software stack of an edge device. This dissertation next briefly discusses its learned design hints in future edge systems.

Design hints to future edge systems

This final section summarizes the general lessons and hints we learned when designing and implementing the five systems in this dissertation. These lessons and hints are behind-the-scene design choices; they are more or less subjective, which make them less appropriate in submissions. We hence lay our two cents (lessons) down as the final concluding remarks, in the fond hope that they may help inspire future edge system designs.

First, coping with an ossified Linux. Besides the research contributions themselves, we are urged to constantly reflect on the *practicality* of our systems when designing them: how well can they integrate with Linux? How much facility can they reuse from Linux?

We did so out of one key vision: *existing Linux kernel is ossified*. Yet, "kernel ossification" does not mean the OS research is dead or that an OS cannot be improved. Instead, it stresses the status quo of the prevalence of Linux kernel – a giant elephant in the room, which is unlikely to see a full-sweep replacement of it, because there is too much legacy. Embracing the prevalence and coping with the legacy fosters wider adoption of new systems.

Therefore, many of systems in this dissertation bear the idea in mind. For instance, we intend Transkernel to execute *unmodified* kernel code and provides firmware-level compatibility across different kernel versions, because doing so enables faster deployment of the OS model and a wider adoption. More so, when we design driverlets and Enigma, building drivers/filesystems from ground up is out of the question in the first place. Why not reuse the mature code which stands the test of time?

We summarize our learned lessons on how to cope with an ossified Linux as follows.

1. **Dancing on the right interface.** Part of the beauty of Linux code lies in the well designed interfaces which cleanly glue different modules. By carefully navigating through the interfaces, new systems shall reduce the reasoning of a large chunk of code into just a few function calls. Transkernel does this: in deciding which kernel services to emulate, it judiciously chooses those with clean and stable ABIs which happen to be stateless, e.g. `schedule()`. Doing so relieves the DBT from the burden of translating the complicated subsystems, which do not contribute to the kernel suspend/resume process. Driverlets also is an example. It is a pioneering work which shows mature Linux device drivers can be reused via reasoning about the interactions at driver/hardware boundary (e.g. register access, DMA allocations). By comparison, prior works choose to pack almost all kernel dependencies, e.g. like a library OS [205].

2. **Code reuse through record and replay.** The technique is well-known and primarily applied to bug finding [325, 326, 327, 284] and security analysis [328, 329, 330]. But it is never considered for code reuse, which is a unique opportunity enabled by edge systems. This is because one key characteristic of an edge system is how "stable" its workloads are and how "simple" the IO devices are. For instance, a smart camera periodically captures frames through a similar kernel stack with a fixed set of VC4 commands. The "stability" forms many *beaten paths* [147], where variables are often only dependent on runtime inputs, e.g. the number of frames requested and corresponding resolutions. This provides future edge systems a unique possibility for reusing the existing code through record-and-replay, which captures the beaten paths and drive the same state transitions on the device, as long as the variables are captured as well. Similarly, the app behavior (i.e. execution logic) can also be imitated through record and replay exemplified by Enigma: it records sybil traces resultant from actual workloads of the app and replay them at runtime as a strong cover for the actual traces.

Second, incorporating app knowledge. An edge system is tightly coupled with its workloads and the apps running on it. As we have shown earlier in this dissertation, optimizing for their workloads is rewarding in multiple aspects: faster deployment, comparable if not better performance, and new doors to many more tradeoffs. To do so, our hint is to channel between apps and the edge system, which requires to set up the two-way communication.

1. **System → App.** On one hand, we encourage the edge system to provide system-level support that exposes low-level information to apps and enables adaptive execution. This shares a similar vision as Exokernel [171] and library OS [172], where apps are deeply coupled with OS resource management. More importantly, the system shall provide such information accurately. The example is exposing power information to apps, as demonstrated by psbox: first, it questions the prior narratives, which assume the aggregated power can be cleanly separated; it then shares the accurate information via enforcing temporal and spatial balloons at OS level. We expect more such information to appear in future edge device as modern hardware becomes more

aggressive in resource multiplexing. Therefore, before exposing the information to apps, the edge system shall rethink the nature of the information – is the information shared by multiple running apps? Can it really be attributed to different apps and what measures shall be taken to facilitate accurate attribution?

2. **App → System.** On the other hand, the edge system incorporates the apps knowledge and optimizes for it. To this end, our first hint is to understand the app workloads and have the edge system supply only *sufficient* functionalities for fast deployment. The idea has first been demonstrated by Transkernel where the DBT only translates kernel beaten path while falling back to native execution when execution spins off the beaten path. It is then highlighted by driverlets, which shows how to pinpoint a set of basic IO functions needed by the app and designs drivers specially for them. Our second hint is to take one step further and peek into what is actually being executed. It may be the execution logic of the app, which Enigma captures by recording sybil file traces. It may as well be large files, e.g. pretrained ML models, which STI shows how to profile and schedule according to model parameter importance. Doing so opens door to new tradeoffs, which help the edge system reconcile tensions between multiple resources.

Bibliography

[1] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding.

[2] Alexa. The top 500 sites on the web. http://www.alexa.com/topsites.

[3] Android IP Webcam Application. https://play.google.com/store/apps/details?id=com.pas.webcam&hl=en.

[4] Apple. Apple watch human interface guidelines. https://developer.apple.com/library/prerelease/ios/documentation/UserExperience/Conceptual/WatchHumanInterfaceGuidelines/, 2015.

[5] iphone 13 and iphone 13 mini - apple. https://www.apple.com/iphone-13/. (Accessed on 04/19/2022).

[6] Texas Instruments. Cortex-M3: Processor technical reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337h/index.html. (Accessed on 05/07/2019).

[7] Arm trustzone. http://www.arm.com/products/processors/technologies/trustzone/index.php.

[8] Clayton Shepard, Ahmad Rahmati, Chad Tossell, Lin Zhong, and Phillip Kortum. Livelab: Measuring wireless networks and smartphone users in the field. *SIGMETRICS Performance Evaluation Review*, 38(3):15–20, January 2011.

[9] Renju Liu and Felix Xiaozhu Lin. Understanding the characteristics of android wear os. In *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, 2016.

[10] Chao Xu, Felix Xiaozhu Lin, Yuyang Wang, and Lin Zhong. Automated os-level device runtime power management. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 239–252, New York, NY, USA, 2015. ACM.

[11] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 48–63, New York, NY, USA, 1999. ACM.

[12] Shuang Zhai, Liwei Guo, Xiangyu Li, and Felix Xiaozhu Lin. Decelerating Suspend and Resume in Operating Systems. In *Proc. ACM Workshp. Mobile Computing Systems & Applications (HotMobile)*, HotMobile '17, pages 31–36, New York, NY, USA, 2017. ACM.

[13] Mario Almeida, Stefanos Laskaridis, Abhinav Mehrotra, Lukasz Dudziak, Ilias Leontiadis, and Nicholas D. Lane. Smart at what cost?: Characterising mobile deep neural networks in the wild. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 658–672. ACM.

[14] Linaro. Op-tee: Open portable trusted execution environment. https://www.op-tee.org/, 2017.

[15] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A data oblivious filesystem for intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[16] Liwei Guo, Shuang Zhai, Yi Qiao, and Felix Xiaozhu Lin. Transkernel: Bridging monolithic kernels to peripheral cores. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 675–692. USENIX Association, 2019.

[17] Liwei Guo, Tiantu Xu, Mengwei Xu, Xuanzhe Liu, and Felix Xiaozhu Lin. Power sandbox: power awareness redefined. In *Proceedings of the Thirteenth EuroSys Conference*, page 37. ACM, 2018.

[18] Liwei Guo, Wonkyo Choe, and Felix Xiaozhu Lin. Efficient NLP inference at the edge via elastic pipelining. *CoRR*, abs/2207.05022, 2022.

[19] Liwei Guo and Felix Xiaozhu Lin. Minimum viable device drivers for ARM trustzone. In Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 300–316. ACM, 2022.

[20] Liwei Guo, Kaiyang Zhao, Yiying Zhang, and Felix Xiaozhu Lin. Enigma: Privacy-preserving file service for arm trustzone. *Under Construction*, 2020.

[21] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 123–132, New York, NY, USA, 2002. ACM.

[22] Mian Dong and Lin Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 335–348, New York, NY, USA, 2011. ACM.

[23] Mian Dong, Tian Lan, and Lin Zhong. Rethink energy accounting with cooperative game theory. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, MobiCom '14, pages 531–542, New York, NY, USA, 2014. ACM.

[24] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 153–168, New York, NY, USA, 2011. ACM.

[25] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[26] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 387–400, Boston, MA, 2012. USENIX.

[27] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power containers: An os facility for fine-grained power and energy management on multicore servers. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 65–76, New York, NY, USA, 2013. ACM.

[28] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 317–328, New York, NY, USA, 2012. ACM.

[29] Jason Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, WMCSA '99, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society.

[30] T. Stathopoulos, D. McIntire, and W. J. Kaiser. The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes. In *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, pages 383–394, April 2008.

[31] Niels Brouwers, Marco Zuniga, and Koen Langendoen. Neat: A novel energy analysis toolkit for free-roaming smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, pages 16–30, New York, NY, USA, 2014. ACM.

[32] Aaron Schulman, Tanuj Thapliyal, Sachin Katti, Neil Spring, Dave Levin, and Prabal Dutta. Stanford CS battor: Plug-and-debug energy debugging for applications on smartphones and laptops. Technical report, 2016.

[33] Eric Brier, Christophe Clavier, and Francis Olivier. *Correlation Power Analysis with a Leakage Model*, pages 16–29. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[34] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. Powerspy: Location tracking using mobile device power analysis. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 785–800, Washington, D.C., 2015. USENIX Association.

[35] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. A study on power side channels on mobile devices. In *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, Internetware '15, pages 30–38, New York, NY, USA, 2015. ACM.

[36] H. Hlavacs, T. Treutner, J. P. Gelas, L. Lefevre, and A. C. Orgerie. Energy consumption side-channel attack at virtual machines in a cloud. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pages 605–612, Dec 2011.

[37] Perf. https://perf.wiki.kernel.org/index.php/Tutorial.

[38] John Levon. OProfile - A System Profiler for Linux. http://oprofile.sourceforge.net/about/.

[39] L. Mukhanov, D. S. Nikolopoulos, and B. R. d. Supinski. Alea: Fine-grain energy profiling with basic block sampling. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 87–98, Oct 2015.

[40] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: making smartphones last longer with code offload. In *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[41] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[42] B. Zhao, W. Hu, Q. Zheng, and G. Cao. Energy-aware web browsing on smartphones. *IEEE Transactions on Parallel and Distributed Systems*, 26(3):761–774, March 2015.

[43] Sergiu Nedevschi, Lucian Popa, Gianluca Iannaccone, Sylvia Ratnasamy, and David Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 323–336, Berkeley, CA, USA, 2008. USENIX Association.

[44] Ning Ding, Daniel Wagner, Xiaomeng Chen, Abhinav Pathak, Y. Charlie Hu, and Andrew Rice. Characterizing and modeling the impact of wireless signal strength on smartphone battery drain. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 29–40, New York, NY, USA, 2013. ACM.

[45] Aaron Schulman, Vishnu Navda, Ramachandran Ramjee, Neil Spring, Pralhad Deshpande, Calvin Grunewald, Kamal Jain, and Venkata N. Padmanabhan. Bartendr: A practical approach to energy-aware cellular data scheduling. In *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*, MobiCom '10, pages 85–96, New York, NY, USA, 2010. ACM.

[46] Kenneth C. Barr and Krste Asanović. Energy-aware lossless data compression. *ACM Trans. Comput. Syst.*, 24(3):250–291, August 2006.

[47] Geoffrey Challen and Mark Hempstead. The case for power-agile computing. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 15–15, Berkeley, CA, USA, 2011. USENIX Association.

[48] Hui Chen, Bing Luo, and Weisong Shi. Anole: A case for energy-aware mobile application design. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ICPPW '12, pages 232–238, Washington, DC, USA, 2012. IEEE Computer Society.

[49] Shivajit Mohapatra, Nalini Venkatasubramanian, Nikil Dutt, Cristiano Pereira, and Rajesh Gupta. Energy-aware adaptations for end-to- end video streaming to mobile handheld devices. In E. Macii, editor, *Ultra Low-Power Electronics and Design*, chapter 10, pages 266–290. Springer Science & Business Media, 2007.

[50] Henry Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 198–214, New York, NY, USA, 2015. ACM.

[51] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 280–293, New York, NY, USA, 2009. ACM.

[52] Mohammad Hosseini, Alexandra Fedorova, Joseph Peters, and Shervin Shirmohammadi. Energy-aware adaptations in mobile 3d graphics. In *Proceedings of the 20th ACM International Conference on Multimedia*, MM '12, pages 1017–1020, New York, NY, USA, 2012. ACM.

[53] Meng Zhu and Kai Shen. Energy discounted computing on multicore smartphones. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 129–141, Denver, CO, 2016. USENIX Association.

[54] Andreas Weissel, Björn Beutel, and Frank Bellosa. Cooperative i/o: A novel i/o semantics for energy-aware applications. *SIGOPS Oper. Syst. Rev.*, 36(SI):117–129, December 2002.

[55] Nicholas D. Lane, Yohan Chon, Lin Zhou, Yongzhe Zhang, Fan Li, Dongwon Kim, Guanzhong Ding, Feng Zhao, and Hojung Cha. Piggyback crowdsensing (pcs): Energy efficient crowdsourcing of mobile sensor data by exploiting smartphone app opportunities. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 7:1–7:14, New York, NY, USA, 2013. ACM.

[56] Guru Prasad Srinivasa, Rizwana Begum, Scott Haseley, Mark Hempstead, and Geoffrey Challen. Separated by birth: Hidden differences between seemingly-identical smartphone cpus. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*, HotMobile '17, pages 103–108, New York, NY, USA, 2017. ACM.

[57] John C. McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta. Evaluating the effectiveness of model-based power characterization. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[58] Fengyuan Xu, Yunxin Liu, Qun Li, and Yongguang Zhang. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 43–55, Lombard, IL, 2013. USENIX.

[59] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM.

[60] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[61] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397, London, UK, UK, 1999. Springer-Verlag.

[62] *Dynamic Time Warping*, pages 69–84. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[63] G. Huang, M. Xu, F. X. Lin, Y. Liu, Y. Ma, S. Pushp, and X. Liu. Shuffledog: Characterizing and adapting user-perceived latency of android apps. *IEEE Transactions on Mobile Computing*, PP(99):1–1, 2017.

[64] David T. Nguyen, Gang Zhou, Guoliang Xing, Xin Qi, Zijiang Hao, Ge Peng, and Qing Yang. Reducing smartphone application delay through read/write isolation. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 287–300, New York, NY, USA, 2015. ACM.

[65] ARM. 64 bit juno arm development platform. http://www.arm.com/files/pdf/Juno_ARM_Development_Platform_datasheet.pdf, 2014.

[66] Philip J. Mucci. PapiEx - execute arbitrary application and measure hardware performance counters with PAPI. http://icl.cs.utk.edu/~mucci/papiex.

[67] Vince Weaver. The unofficial Linux Perf Events web-page. http://web.eece.maine.edu/~vweaver/projects/perf_events. Last accessed: Dec. 12, 2013.

[68] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 189–194, New York, NY, USA, 2010. ACM.

[69] Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002.

[70] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. Acpi - advanced configuration and power interface. http://www.acpi.info/.

[71] John K. Ousterhout. Scheduling techniques for concurrebt systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems, Miami/Ft. Lauderdale, Florida, USA, October 18-22, 1982*, pages 22–30, 1982.

[72] Nikunj A. Dadhania. Gang scheduling in cfs. https://lwn.net/Articles/472797/, 2011.

[73] R. Chandra, P. Bahl, and P. Bahl. Multinet: connecting to multiple ieee 802.11 networks using a single wireless card. In *IEEE INFOCOM 2004*, volume 2, pages 882–893 vol.2, March 2004.

[74] Lei Xia, Sanjay Kumar, Xue Yang, Praveen Gopalakrishnan, York Liu, Sebastian Schoenberg, and Xingang Guo. Virtual wifi: Bring virtualization from wired to wireless. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 181–192, New York, NY, USA, 2011. ACM.

[75] Microsoft Hardware Dev Center. Virtual wifi in kernel mode. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/virtual-wifi-in-kernel-mode/, 2017.

[76] Texas Instruments. WL18x7MOD WiLink 8 Dual-Band Industrial Module – Wi-Fi, Bluetooth, and Bluetooth Low Energy, 2015.

[77] Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 225–238, New York, NY, USA, 2012. ACM.

[78] Measurement Computing. USB-1608G Series User's Guide, 2012.

[79] Matt Blaze. A cryptographic file system for unix. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 9–16, New York, NY, USA, 1993. ACM.

[80] Texas Instruments. Processor SDK Demos Video Analytics. http://processors.wiki.ti.com/index.php/Processor_SDK_Demos_Video_Analytics.

[81] Aaron Carroll and Gernot Heiser. The systems hacker's guide to the galaxy: energy usage in a modern smartphone. In *Proc. of the 4th Asia-Pacific Workshop on Systems (APSYS)*, page 5. ACM, 2013.

[82] M. Dong and L. Zhong. Chameleon: A color-adaptive web browser for mobile oled displays. *IEEE Transactions on Mobile Computing*, 11(5):724–738, May 2012.

[83] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan and Claypool Publishers, 2013.

[84] Zehan Cui, Yan Zhu, Y. Bao, and M. Chen. A fine-grained component-level power measurement method. In *2011 International Green Computing Conference and Workshops*, pages 1–6, July 2011.

[85] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 323–338, Berkeley, CA, USA, 2008. USENIX Association.

[86] Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10(1):53–79, 1992.

[87] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.

[88] IHS Inc. Led by iphone 6s, sensor hubs market is growing fast, ihs says, ihs markit press release, 2017.

[89] Kionix. Kx23h-1035: Arm-based sensor hub with accelerometer. http://www.kionix.com/product/KX23H-1035, 2014.

[90] Nandan Nayampally. ARM DynamIQ: Expanding the possibilities for artificial intelligence. 2017.

[91] Apple Inc. iPhone X, Tech Specs. https://www.apple.com/iphone-x/specs/, 2017.

[92] Texus Instruments. Ads7040: Ultra-low-power ultra-small-size sar adc. http://www.ti.com/product/ADS7040, 2017.

[93] Texus Instruments. Ina231, ina3221 triple-channel, high-side measurement, shunt and bus voltage monitor with i2c and smbus-compatible interface. http://www.ti.com/lit/ds/symlink/ina3221.pdf, 2016.

[94] Texus Instruments. Ina3221, 28-v, bi-directional, zero-drift, low-/high-side, i2c out current/power monitor w/ alert in wcsp. http://www.ti.com/product/INA231, 2018.

[95] Nvidia. Jetson tx1 voltage and current monitor configuration application note. https://developer.nvidia.com/embedded/tegra-2-reference, 2017.

[96] Nvidia. Tegra x2: Technical reference manual. https://developer.nvidia.com/embedded/tegra-2-reference, 2017.

[97] Hardkernel. Odroid xu3: Board detail. http://www.hardkernel.com/main/products/prdt_info.php?g_code=g140448267127&tab_idx=2, 2014.

[98] Google. Sensors Overview. https://developer.android.com/guide/topics/sensors/sensors_overview.html.

[99] Apple. Core Motion. https://developer.apple.com/documentation/coremotion.

[100] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[101] QuickLogic. SenseMe™ - Sensor Algorithm Library for Mobile Devices. https://www.quicklogic.com/technologies/sensor-hub/senseme/.

[102] Felix Xiaozhu Lin, Zhen Wang, Robert LiKamWa, and Lin Zhong. Reflex: using low-power processors in smartphones without knowing them. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 13–24, New York, NY, USA, 2012. ACM.

[103] Haichen Shen, Aruna Balasubramanian, Anthony LaMarca, and David Wetherall. Enhancing Mobile Apps to Use Sensor Hubs Without Programmer Effort. In *Proc. Int. Conf. Ubiquitous Computing (UbiComp)*, UbiComp '15, pages 227–238, New York, NY, USA, 2015. ACM.

[104] Daniyal Liaqat, Silviu Jingoi, Eyal de Lara, Ashvin Goel, Wilson To, Kevin Lee, Italo De Moraes Garcia, and Manuel Saldana. Sidewinder: An energy efficient and developer friendly heterogeneous architecture for continuous mobile sensing. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 205–215, New York, NY, USA, 2016. ACM.

[105] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: A platform for os-level power management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 289–302, New York, NY, USA, 2009. ACM.

[106] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. A validation of dram rapl power measurements. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, pages 455–470, New York, NY, USA, 2016. ACM.

[107] D. Hackenberg, T. Ilsche, R. Schöne, D. Molka, M. Schmidt, and W. E. Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–204, April 2013.

[108] P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, pages 283–294, April 2008.

[109] Patrick Titiano. Leveraging open-source power measurement standard solution. http://events.linuxfoundation.org/sites/events/files/slides/Leveraging_Open-Source_Power_Measurement_Standard_Solution_0.pdf.

[110] Bartosz Golaszewski. sigrok: Adventures in integrating a power-measurement device. http://events.linuxfoundation.org/sites/events/files/slides/ELC_pres_bgolaszewski.pdf.

[111] Yan Zhai, Xiao Zhang, Stephane Eranian, Lingjia Tang, and Jason Mars. Happy: Hyperthread-aware power profiling dynamically. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 211–217, Philadelphia, PA, 2014. USENIX Association.

[112] Farshad Ghanei, Pranav Tipnis, Kyle Marcus, Karthik Dantu, Steve Ko, and Lukasz Ziarek. Os-based resource accounting for asynchronous resource use in mobile systems. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16, pages 296–301, New York, NY, USA, 2016. ACM.

[113] Aman Kansal, Feng Zhao, Jie Liu, Nupur Kothari, and Arka A. Bhattacharya. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 39–50, New York, NY, USA, 2010. ACM.

[114] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Energy management in mobile devices with the cinder operating system. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 139–152, New York, NY, USA, 2011. ACM.

[115] K. Rao, J. Wang, S. Yalamanchili, Y. Wardi, and Y. Handong. Application-specific performance-aware energy optimization on android mobile devices. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 169–180, Feb 2017.

[116] Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. Integrated CPU-GPU power management for 3D mobile games. In *Proc. of the 51st Annual Design Automation Conference (DAC)*, pages 40:1–40:6, 2014.

[117] Chao Xu, Felix Xiaozhu Lin, Yuyang Wang, and Lin Zhong. Automated OS-level Device Power Management for SoCs. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, New York, NY, USA, 2015. ACM.

[118] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[119] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[120] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class gpu resource management in the operating system. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 37–37, Berkeley, CA, USA, 2012. USENIX Association.

[121] Renju Liu and Felix Xiaozhu Lin. Understanding the Characteristics of Android Wear OS. In *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, MobiSys '16, pages 151–164, New York, NY, USA, 2016. ACM.

[122] Matthew Lentz, James Litton, and Bobby Bhattacharjee. Drowsy Power Management. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, SOSP '15, pages 230–244, New York, NY, USA, 2015. ACM.

[123] Fengyuan Xu, Yunxin Liu, Thomas Moscibroda, Ranveer Chandra, Long Jin, Yongguang Zhang, and Qun Li. Optimizing Background Email Sync on Smartphones. In *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, pages 55–68, 2013.

[124] Xiaomeng Chen, Abhilash Jindal, Ning Ding, Yu Charlie Hu, Maruti Gupta, and Rath Vannithamby. Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization. In *Proc. Ann. Int. Conf. Mobile Computing & Networking (MobiCom)*, MobiCom '15, pages 40–52, New York, NY, USA, 2015. ACM.

[125] Xiaomeng Chen, Ning Ding, Abhilash Jindal, Y. Charlie Hu, Maruti Gupta, and Rath Vannithamby. Smartphone Energy Drain in the Wild: Analysis and Implications. In *Proc. ACM SIGMETRICS (SIGMETRICS)*, pages 151–164. ACM, 2015.

[126] Xing Liu, Tianyu Chen, Feng Qian, Zhixiu Guo, Felix Xiaozhu Lin, Xiaofeng Wang, and Kai Chen. Characterizing Smartwatch Usage in the Wild. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 385–398, New York, NY, USA, 2017. ACM.

[127] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. Farmbeats: An iot platform for data-driven agriculture. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 515–529, Boston, MA, 2017. USENIX Association.

[128] Ulf Hansson. SDIO power on/off time impacts system suspend/resume time! http://connect.linaro.org/resource/sfo17/sfo17-402/, 2017.

[129] LWN. Redesigning asynchronous suspend/resume. https://lwn.net/Articles/366915/, 2009.

[130] LKML. [git pull] pm updates for 2.6.33, 2009.

[131] Jim Morrison, Daniel Yang, and Chad Davis. Apple watch: teardown. https://www.techinsights.com/about-techinsights/overview/blog/apple-watch-teardown/. (Accessed on 01/10/2019).

[132] MediaTek. Microsoft Azure Sphere MCU with extensive I/O peripheral subsystem for diverse IoT applications. https://www.mediatek.com/products/azureSphere/mt3620, 2018.

[133] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A mobile operating system for heterogeneous coherence domains. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 285–300. ACM, 2014.

[134] David Meisner and Thomas F. Wenisch. DreamWeaver: architectural support for deep sleep. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, ASPLOS '12, pages 313–324, New York, NY, USA, 2012. ACM.

[135] Yuvraj Agarwal, Steve Hodges, Ranveer Chandra, James Scott, Paramvir Bahl, and Rajesh Gupta. Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage. In *Proc. USENIX Symp. Networked Systems Design and Implementation (NSDI)*, pages 365–380. USENIX Association, 2009.

[136] Yoann Padioleau, Julia L Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 59–71. ACM, 2006.

[137] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms. In *Proc. The European Conf. Computer Systems (EuroSys)*, EuroSys '15, pages 29:1–29:16, New York, NY, USA, 2015. ACM.

[138] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, SOSP '09, pages 221–234, New York, NY, USA, 2009. ACM.

[139] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: a new OS architecture for scalable multicore systems. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, pages 29–44. ACM, 2009.

[140] Gang Lu, Jianfeng Zhan, Xinlong Lin, Chongkang Tan, and Lei Wang. On Horizontal Decomposition of the Operating System. *CoRR*, abs/1604.01378, 2016.

[141] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, pages 69–87, 2018.

[142] A. Leonard Brown and Rafael J. Wysocki. Suspend-to-RAM in Linux. In *Ottawa Linux Symposium*, pages 39–52, 2008.

[143] Intel. Intel suspendresume project. https://01.org/suspendresume, 2015.

[144] Asim Kadav and Michael M. Swift. Understanding Modern Device Drivers. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, ASPLOS XVII, pages 87–98, New York, NY, USA, 2012. ACM.

[145] David Meisner, Brian T. Gold, and Thomas F. Wenisch. PowerNap: Eliminating Server Idle Power. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, ASPLOS XIV, pages 205–216, New York, NY, USA, 2009. ACM.

[146] Qi Zhu, Meng Zhu, Bo Wu, Xipeng Shen, Kai Shen, and Zhiying Wang. Software Engagement with Sleeping CPUs. In *Proc. Workshp. Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[147] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-Pop: securing privileged operating system kernels by keeping on the beaten path. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 1–13. USENIX Association, 2017.

[148] Texas Instruments. OMAP4 applications processor: Technical reference manual. http://www.ti.com/product/OMAP4470, 2010.

[149] Texas Instruments. AM5728 Sitara Processor: Dual Arm Cortex-A15 & Dual DSP, Multimedia — TI.com. http://www.ti.com/product/AM5728. (Accessed on 05/14/2019).

[150] NXP Semiconductors. i.MX 6SoloX - fact sheet. https://www.nxp.com/docs/en/fact-sheet/IMX6SOLOXFS.pdf. (Accessed on 05/14/2019).

[151] NXP Semiconductors. i.MX 7 Series Applications Processors — Arm® Cortex®-A7, Cortex-M4 — NXP. https://www.nxp.com/products/processors-and-microcontrollers/arm-based-processors-and-mcus/i.mx-applications-processors/i.mx-7-processors:IMX7-SERIES, 2017. (Accessed on 05/14/2019).

[152] NXP Semiconductors. i.MX 8M Family of Applications Processors Fact Sheet. https://www.nxp.com/docs/en/fact-sheet/i.MX8M-FS.pdf. (Accessed on 05/14/2019).

[153] Jacob Sorber, Nilanjan Banerjee, Mark D Corner, and Sami Rollins. Turducken: hierarchical power management for mobile devices. In *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, pages 261–274. ACM, 2005.

[154] NXP Semiconductors. i.MX 7DS power consumption measurement. https://www.nxp.com/docs/en/application-note/AN5383.pdf, 2016.

[155] J.C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *IEEE Micro*, 28(3):26–41, 2008.

[156] Peter Greenhalgh. Big.LITTLE processing with ARM Cortex-A15 and Cortex-A7. Technical report, 2011.

[157] Emily Blem, Jaikrishnan Menon, Thiruvengadam Vijayaraghavan, and Karthikeyan Sankaralingam. ISA wars: Understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures. *ACM Transactions on Computer Systems (TOCS)*, 33(1):3, 2015.

[158] Tong Li, Paul Brett, Rob Knauerhase, David Koufaty, Dheeraj Reddy, and Scott Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *Proc. IEEE Int. Symp. on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2010.

[159] Andrey Ponomarenko. ABI Compliance Checker. https://lvc.github.io/abi-compliance-checker/, 2018.

[160] Balazs Gerofi, Aram Santogidis, Dominique Martinet, and Yutaka Ishikawa. PicoDriver: Fast-path Device Drivers for Multi-kernel Operating Systems. In *Proc. Int. Symp. on High-Performance Parallel and Distributed Computing (HPDC)*, HPDC '18, pages 2–13, New York, NY, USA, 2018. ACM.

[161] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The Design and Implementation of Microdrivers. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, ASPLOS XIII, pages 168–178, New York, NY, USA, 2008. ACM.

[162] Greg Kroah-Hartman. The Linux Kernel Driver Interface – Stable API Nonsense. https://www.kernel.org/doc/Documentation/process/stable-api-nonsense.rst. (Accessed on 05/04/2019).

[163] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, 2003.

[164] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proc. USENIX Annual Technical Conference (ATC)*. Boston, 2010.

[165] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2004.

[166] Michael Larabel. A Stable Linux Kernel API/ABI? "The Most Insane Proposal" For Linux Development. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Kernel-Stable-API-ABI, 2016.

[167] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-ISA datacenters. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 645–659. ACM, 2017.

[168] Amanieu d'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. Low Overhead Dynamic Binary Translation on ARM. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, PLDI 2017, pages 333–346, New York, NY, USA, 2017. ACM.

[169] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, Stephen McCamant, Youfeng Wu, and Jayaram Bobba. Enabling Cross-ISA Offloading for COTS Binaries. In *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, pages 319–331. ACM, 2017.

[170] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 41–46, 2005.

[171] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[172] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, ASPLOS XVI, pages 291–304, New York, NY, USA, 2011. ACM.

[173] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 461–472. ACM, 2013.

[174] Mike Turquette. The Common Clk Framework. https://www.kernel.org/doc/Documentation/clk.txt.

[175] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[176] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[177] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. Optimizing Binary Translation of Dynamically Generated Code. In *Proc. Int. Symp. on Code Generation and Optimization (CGO)*, CGO '15, pages 68–78, Washington, DC, USA, 2015. IEEE Computer Society.

[178] Alastair Reid. Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, pages 161–168, 2016.

[179] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. Enhancing Cross-ISA DBT Through Automatically Learned Translation Rules. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, ASPLOS '18, pages 84–97, New York, NY, USA, 2018. ACM.

[180] Piyus Kedia and Sorav Bansal. Fast Dynamic Binary Translation for the Kernel. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, SOSP '13, pages 101–115, New York, NY, USA, 2013. ACM.

[181] VMware. Technical note: Virtual machine to physical machine migration, 2004.

[182] Peter Feiner, Angela Demke Brown, and Ashvin Goel. Comprehensive kernel instrumentation via dynamic binary translation. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 135–146. ACM, 2012.

[183] eLinux.org. PandaBoard Power Measurements. http://elinux.org/PandaBoard_Power_Measurements.

[184] Micron Technology, Inc. TN4201 LPDDR2 System Power Calculator. https://www.micron.com/support/tools-and-utilities/power-calc, 2013.

[185] Hitoshi Oi. A Case Study of Energy Efficiency on a Heterogeneous Multi-Processor. *SIGMETRICS Perform. Eval. Rev.*, 45(2):70–72, 2017.

[186] Marcus Hähnel and Hermann Härtig. Heterogeneity by the numbers: A study of the ODROID XU+E big.little platform. In Yuvraj Agarwal and Karthick Rajamani, editors, *Proc. Workshp. Power-Aware Computing and Systems (HotPower)*. USENIX Association, 2014.

[187] Nadja Peters, Sangyoung Park, Samarjit Chakraborty, Benedikt Meurer, Hannes Payer, and Daniel Clifford. Web browser workload characterization for power management on HMP platforms. In *Proc. IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES)*, pages 26:1–26:10. ACM, 2016.

[188] Dumitrel Loghin, Bogdan Marius Tudor, Hao Zhang, Beng Chin Ooi, and Yong Meng Teo. A Performance Study of Big Data on Small Nodes. *Proc. VLDB Endow.*, 8(7):762–773, 2015.

[189] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, 2009. 1531805.

[190] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard P. Fettweis. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 189–203, 2016.

[191] Changwoo Min, Woonhak Kang, Mohan Kumar, Sanidhya Kashyap, Steffen Maass, Heeseung Jo, and Taesoo Kim. Solros: a data-centric operating system architecture for heterogeneous computing. In *Proc. The European Conf. Computer Systems (EuroSys)*, page 36. ACM, 2018.

[192] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a File System with GPUs. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, ASPLOS '13, pages 485–498, New York, NY, USA, 2013. ACM.

[193] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. Execution migration in a heterogeneous-ISA chip multiprocessor. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 261–272, New York, NY, USA, 2012. ACM.

[194] Raymond J Hookway and Mark A Herdeg. Digital FX! 32: Combining emulation and binary translation. *Digital Technical Journal*, 9:3–12, 1997.

[195] Darrell Boggs, Gary Brown, Nathan Tuck, and K. S. Venkatraman. Denver: Nvidia's First 64-bit ARM Processor. *IEEE Micro*, 35(2):46–55, 2015.

[196] Alexander Klaiber. The technology behind Crusoe processors. *Transmeta Technical Brief*, 2000.

[197] Simon Rokicki, Erven Rohou, and Steven Derrien. Hardware-accelerated dynamic binary translation. In *Proc. ACM/IEEE Design Automation & Test in Europe Conf. (DATE)*, pages 1062–1067, 2017.

[198] Simon Rokicki, Erven Rohou, and Steven Derrien. Supporting runtime reconfigurable VLIWs cores through dynamic binary translation. In *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, pages 1009–1014, 2018.

[199] Amanieu d'Antras, Cosmin Gorgovan, Jim Garside, John Goodacre, and Mikel Luján. HyperMAMBO-X64: Using Virtualization to Support High-Performance Transparent Binary Translation. In *Proc. Int. Conf. Virtual Execution Environments (VEE)*, VEE '17, pages 228–241, New York, NY, USA, 2017. ACM.

[200] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proc. Int. Symp. on Code Generation and Optimization (CGO)*, pages 104–113, 2012.

[201] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, pages 177–192. USENIX Association, 2008.

[202] Jon Howell, Bryan Parno, and John R. Douceur. How to Run POSIX Apps in a Minimal Picoprocess. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 321–332, 2013.

[203] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In Joseph S. Sventek and Steven Hand, editors, *Proc. The European Conf. Computer Systems (EuroSys)*, pages 247–260. ACM, 2008.

[204] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proc. USENIX OSDI*, 2004.

[205] Antti Kantee and Justin Cormack. Rump Kernels No OS? No Problem! *Login: USENIX Magazine*, 39(5), 2014.

[206] Sam (Likun) Xi, Marisabel Guevara, Jared Nelson, Patrick Pensabene, and Benjamin C. Lee. Understanding the Critical Path in Power State Transition Latencies. In *Proc. ACM/IEEE Int. Symp. Low Power Electronics & Design (ISLPED)*, ISLPED '13, pages 317–322, Piscataway, NJ, USA, 2013. IEEE Press.

[207] Allan de Barcelos Silva, Marcio Miguel Gomes, Cristiano André da Costa, Rodrigo da Rosa Righi, Jorge Luis Victoria Barbosa, Gustavo Pessin, Geert De Doncker, and Gustavo Federizzi. Intelligent personal assistants: A systematic literature review. 147:113193.

[208] Thierry Tambe, Coleman Hooper, Lillian Pentecost, Tianyu Jia, En-Yu Yang, Marco Donato, Victor Sanh, Paul Whatmough, Alexander M. Rush, David Brooks, and Gu-Yeon Wei. EdgeBERT: Sentence-Level Energy Optimizations for Latency-Aware Multi-Task NLP Inference. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 830–844. ACM.

[209] Alejandro Cartas, Martin Kocour, Aravindh Raman, Ilias Leontiadis, Jordi Luque, Nishanth Sastry, José Núñez-Martínez, Diego Perino, and Carlos Segura. A reality check on inference at mobile networks edge. In *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking, EdgeSys@EuroSys 2019, Dresden, Germany, March 25, 2019*, pages 54–59. ACM, 2019.

[210] Toine Bogers, Ammar Ali Abdelrahim Al-Basri, Claes Ostermann Rytlig, Mads Emil Bak Møller, Mette Juhl Rasmussen, Nikita Katrine Bates Michelsen, and Sara Gerling Jørgensen. A Study of Usage and Usability of Intelligent Personal Assistants in Denmark. In Natalie Greene Taylor, Caitlin Christian-Lamb, Michelle H. Martin, and Bonnie Nardi, editors, *Information in Contemporary Society*, Lecture Notes in Computer Science, pages 79–90. Springer International Publishing.

[211] Juan Pablo Carrascal and Karen Church. An in-situ study of mobile app & mobile search interactions. In Bo Begole, Jinwoo Kim, Kori Inkpen, and Woontack Woo, editors, *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI 2015, Seoul, Republic of Korea, April 18-23, 2015*, pages 2739–2748. ACM, 2015.

[212] Paul Michel, Omer Levy, and Graham Neubig. Are Sixteen Heads Really Better than One?

[213] Haonan Yu, Sergey Edunov, Yuandong Tian, and Ari S. Morcos. Playing the lottery with rewards and multiple languages: lottery tickets in RL and NLP. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[214] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.

[215] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter.

[216] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[217] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 4510–4520. Computer Vision Foundation / IEEE Computer Society, 2018.

[218] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 6848–6856. Computer Vision Foundation / IEEE Computer Society, 2018.

[219] Hugging face:nlptown/bert-base-multilingual-uncased-sentiment. `https://huggingface.co/nlptown/bert-base-multilingual-uncased-sentiment`. (Accessed on 07/07/2022).

[220] Kasturi Bhattacharjee, Miguel Ballesteros, Rishita Anubhai, Smaranda Muresan, Jie Ma, Faisal Ladhak, and Yaser Al-Onaizan. To BERT or not to BERT: comparing task-specific and task-agnostic semi-supervised approaches for sequence tagging. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 7927–7934. Association for Computational Linguistics, 2020.

[221] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. End the senseless killing: Improving memory management for mobile operating systems. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 873–887. USENIX Association, 2020.

[222] Soyoon Lee and Hyokyung Bahn. Characterization of android memory references and implication to hybrid memory management. *IEEE Access*, 9:60997–61009, 2021.

[223] Android. Android: Low memory killer daemon. `https://source.android.com/devices/tech/perf/lmkd/`, 2022.

[224] Huoran Li, Xuan Lu, Xuanzhe Liu, Tao Xie, Kaigui Bian, Felix Xiaozhu Lin, Qiaozhu Mei, and Feng Feng. Characterizing smartphone usage patterns from millions of android users. In Kenjiro Cho, Kensuke Fukuda, Vivek S. Pai, and Neil Spring, editors, *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*, pages 459–472. ACM, 2015.

[225] Yuan Tian, Ke Zhou, Mounia Lalmas, and Dan Pelleg. Identifying tasks from mobile app usage patterns. In Jimmy Huang, Yi Chang, Xueqi Cheng, Jaap Kamps, Vanessa Murdock, Ji-Rong Wen, and Yiqun Liu, editors, *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*, pages 2357–2366. ACM, 2020.

[226] Rongjie Yi, Ting Cao, Ao Zhou, Xiao Ma, Shangguang Wang, and Mengwei Xu. Understanding and optimizing deep learning cold-start latency on edge devices, 2022.

[227] Hongyu Miao and Felix Xiaozhu Lin. Enabling large nns on tiny mcus with swapping. *CoRR*, abs/2101.08744, 2021.

[228] Hanrui Wang. *Efficient Algorithms and Hardware for Natural Language Processing*. PhD dissertation, Massachusetts Institute of Technology, 2020.

[229] Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. DynaBERT: Dynamic BERT with Adaptive Width and Depth.

[230] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. GOBO: Quantizing Attention-Based NLP Models for Low Latency and Energy Efficient Inference. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 811–824. IEEE.

[231] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need.

[232] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[233] Xiantao Chen, Moli Zhou, Renzhen Wang, Yalin Pan, Jiaqi Mi, Hui Tong, and Daisong Guan. Evaluating response delay of multimodal interface in smart device. In Aaron Marcus and Wentao Wang, editors, *Design, User Experience, and Usability. Practice and Case Studies - 8th International Conference, DUXU 2019, Held as Part of the 21st HCI International Conference, HCII 2019, Orlando,*

FL, USA, July 26-31, 2019, Proceedings, Part IV, volume 11586 of *Lecture Notes in Computer Science*, pages 408–419. Springer, 2019.

[234] Pytorch. https://pytorch.org/. (Accessed on 03/14/2022).

[235] Tensorflow. https://www.tensorflow.org/, 2021. (Accessed on 04/22/2021).

[236] Luting Yang, Bingqian Lu, and Shaolei Ren. A Note on Latency Variability of Deep Neural Networks for Mobile Inference.

[237] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112, 2019.

[238] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for DNN training. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 1–15. ACM, 2019.

[239] Suchita Pati, Shaizeen Aga, Nuwan Jayasena, and Matthew D. Sinclair. Demystifying BERT: Implications for Accelerator Design.

[240] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In Anna Korhonen, David R. Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 5797–5808. Association for Computational Linguistics, 2019.

[241] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. MobileBERT: A Compact Task-Agnostic BERT for Resource-Limited Devices.

[242] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations.

[243] Haotong Qin, Yifu Ding, Mingyuan Zhang, Qinghua Yan, Aishan Liu, Qingqing Dang, Ziwei Liu, and Xianglong Liu. BIBERT: ACCURATE FULLY BINARIZED BERT. page 24.

[244] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient Transformers: A Survey.

[245] Yangyang Shi, Yongqiang Wang, Chunyang Wu, Ching-Feng Yeh, Julian Chan, Frank Zhang, Duc Le, and Mike Seltzer. Emformer: Efficient memory transformer based acoustic model for low latency streaming speech recognition. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6783–6787. IEEE, 2021.

[246] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. RecSSD: Near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 717–729. ACM.

[247] Hu Wan, Xuan Sun, Yufei Cui, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. FlashEmbedding: Storing embedding tables in SSD for large-scale recommender systems. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 9–16. ACM.

[248] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using Non-volatile Memory for Storing Deep Learning Models. page 13.

[249] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach.

[250] Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. HAT: Hardware-Aware Transformers for Efficient Natural Language Processing.

[251] Michael Frederick McTear, Zoraida Callejas, and David Griol. *The conversational interface*, volume 6. Springer, 2016.

[252] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[253] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In Anna Korhonen, David R. Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 5797–5808. Association for Computational Linguistics, 2019.

[254] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. What does BERT look at? an analysis of bert's attention. In Tal Linzen, Grzegorz Chrupala, Yonatan Belinkov, and Dieuwke Hupkes, editors, *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@ACL 2019, Florence, Italy, August 1, 2019*, pages 276–286. Association for Computational Linguistics, 2019.

[255] Chengyue Gong, Zixuan Jiang, Dilin Wang, Yibo Lin, Qiang Liu, and David Z. Pan. Mixed Precision Neural Architecture Search for Energy Efficient Deep Learning. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7.

[256] Zhen Dong, Zhewei Yao, Amir Gholami, Michael Mahoney, and Kurt Keutzer. HAWQ: Hessian AWare Quantization of Neural Networks With Mixed-Precision. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 293–302. IEEE.

[257] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8604–8612. IEEE.

[258] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8BERT: Quantized 8Bit BERT.

[259] Thomas Lorenser. The dsp capabilities of arm cortex-m4 and cortex-m7 processors. *ARM White Paper*, 29, 2016.

[260] Heejin Park and Felix Xiaozhu Lin. Gpureplay: a 50-kb gpu stack for client ml. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–170, 2022.

[261] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. AsyMo: Scalable and efficient deep-learning inference on asymmetric mobile CPUs. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 215–228. ACM.

[262] Pytorch 1.11, torchdata, and functorch are now available — pytorch. https://pytorch.org/blog/pytorch-1.11-released/. (Accessed on 07/07/2022).

[263] Version 0.23.2 — scikit-learn 1.1.1 documentation. https://scikit-learn.org/stable/whats_new/v0.23.html. (Accessed on 07/07/2022).

[264] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[265] Angela Fan, Edouard Grave, and Armand Joulin. Reducing Transformer Depth on Demand with Structured Dropout.

[266] Haoli Bai, Wei Zhang, Lu Hou, Lifeng Shang, Jing Jin, Xin Jiang, Qun Liu, Michael Lyu, and Irwin King. BinaryBERT: Pushing the Limit of BERT Quantization.

[267] Zhenhua Liu, Xinfeng Zhang, Shanshe Wang, Siwei Ma, and Wen Gao. Evolutionary Quantization of Neural Networks with Mixed-Precision. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2785–2789.

[268] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In Rajeev Shorey, Rohan Murty, Yingying (Jennifer) Chen, and Kyle Jamieson, editors, *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking, MobiCom 2018, New Delhi, India, October 29 - November 02, 2018*, pages 115–127. ACM, 2018.

[269] Yang Hu, Connor Imes, Xuanang Zhao, Souvik Kundu, Peter A. Beerel, Stephen P. Crago, and John Paul N. Walters. Pipeline Parallelism for Inference on Heterogeneous Edge Computing.

[270] Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. Pipetransformer: Automated elastic pipelining for distributed training of large-scale models. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 4150–4159. PMLR, 2021.

[271] Trusty tee uses and examples. https://source.android.com/security/trusty#uses_examples. (Accessed on 02/08/2022).

[272] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM trustzone to build a trusted language runtime for mobile applications. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, pages 67–80. ACM, 2014.

[273] Fan Mo, Hamed Haddadi, Kleomenis Katevas, Eduard Marin, Diego Perino, and Nicolas Kourtellis. PPFL: privacy-preserving federated learning with trusted execution environments. In Suman Banerjee, Luca Mottola, and Xia Zhou, editors, *MobiSys '21: The 19th Annual International Conference on Mobile Systems, Applications, and Services, Virtual Event, Wisconsin, USA, 24 June - 2 July, 2021*, pages 94–108. ACM, 2021.

[274] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. Darknetz: Towards model privacy at the edge using trusted execution environments. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, MobiSys '20, page 161–174, New York, NY, USA, 2020. Association for Computing Machinery.

[275] Chang Min Park, Donghwi Kim, Deepesh Veersen Sidhwani, Andrew Fuchs, Arnob Paul, Sung-Ju Lee, Karthik Dantu, and Steven Y. Ko. Rushmore: Securely displaying static and animated images using trustzone. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '21, page 122–135, New York, NY, USA, 2021. Association for Computing Machinery.

[276] commit 148fec63133d755259feac338d50a747976dbf37 - trusty/lk/trusty - git at google. https://android.googlesource.com/trusty/lk/trusty/+/148fec63133d755259feac338d50a747976dbf37. (Accessed on 02/21/2022).

[277] optee_os/changelog.md at 0.1.0 · op-tee/optee_os · github. https://github.com/OP-TEE/optee_os/blob/0.1.0/CHANGELOG.md. (Accessed on 02/21/2022).

[278] optee_os/changelog.md at master · op-tee/optee_os · github. https://github.com/OP-TEE/optee_os/blob/master/CHANGELOG.md. (Accessed on 08/12/2021).

[279] Sd standard overview — sd association. https://www.sdcard.org/developers/sd-standard-overview/. (Accessed on 02/21/2022).

[280] e.mmc — jedec. https://www.jedec.org/standards-documents/technology-focus-areas/flash-memory-ssds-ufs-emmc/e-mmc. (Accessed on 02/21/2022).

[281] I2c broadcom bug workaround. · issue #254 · raspberrypi/linux. https://github.com/raspberrypi/linux/issues/254. (Accessed on 10/09/2021).

[282] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter R. Pietzuch. SGX-LKL: securing the host OS interface for trusted execution. *CoRR*, abs/1908.11143, 2019.

[283] National Institute of Standards and Technology (NIST). Defending against software supply chain attacks. https://www.cisa.gov/sites/default/files/publications/defending_against_software_supply_chain_attacks_508_1.pdf.

[284] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: an application-level kernel for record and replay. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 193–208. USENIX Association, 2008.

[285] Xianzheng Dou, Peter M. Chen, and Jason Flinn. Knockoff: Cheap versions in the cloud. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 73–88, Santa Clara, CA, 2017. USENIX Association.

[286] Daniel Hein, Johannes Winter, and Andreas Fitzek. Secure block device – secure, flexible, and efficient data storage for arm trustzone systems. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 222–229, 2015.

[287] Bernard Dickens III, Haryadi S. Gunawi, Ariel J. Feldman, and Henry Hoffmann. Strongbox: Confidentiality, integrity, and performance using stream ciphers for full drive encryption. In Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 708–721. ACM, 2018.

[288] Tiago Brito, Nuno O. Duarte, and Nuno Santos. ARM trustzone for secure image processing on the cloud. In *35th IEEE Symposium on Reliable Distributed Systems Workshops, SRDS 2016 Workshop, Budapest, Hungary, September 26, 2016*, pages 37–42. IEEE Computer Society, 2016.

[289] Ardalan Amiri Sani. Schrodintext: Strong protection of sensitive textual content of mobile applications. In Tanzeem Choudhury, Steven Y. Ko, Andrew Campbell, and Deepak Ganesan, editors, *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'17, Niagara Falls, NY, USA, June 19-23, 2017*, pages 197–210. ACM, 2017.

[290] Mengwei Xu, Tiantu Xu, Yunxin Liu, Xuanzhe Liu, Gang Huang, and Felix Xiaozhu Lin. Supporting video queries on zero-streaming cameras. *arXiv preprint arXiv:1904.12342*, 2019.

[291] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[292] Device power management basics — the linux kernel documentation. https://www.kernel.org/doc/html/v4.18/driver-api/pm/devices.html. (Accessed on 10/09/2021).

[293] Usb hotplugging — the linux kernel documentation. https://www.kernel.org/doc/html/v4.18/driver-api/usb/hotplug.html. (Accessed on 10/09/2021).

[294] optee_os/serial8250_uart.c at master · op-tee/optee_os · github. https://github.com/OP-TEE/optee_os/blob/master/core/drivers/serial8250_uart.c. (Accessed on 02/21/2022).

[295] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with Termite. In *Proc. of the 22nd symposium on Operating systems principles (SOSP)*, pages 73–86. ACM, 2009.

[296] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. User-guided device driver synthesis. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 661–676, 2014.

[297] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 488–501, New York, NY, USA, 2017. ACM.

[298] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. Building trusted path on untrusted device drivers for mobile devices. In *Asia-Pacific Workshop on Systems, APSys'14, Beijing, China, June 25-26, 2014*, pages 8:1–8:7. ACM, 2014.

[299] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pages 253–264. ACM, 2013.

[300] NVIDIA. Tegra2 Family: Technical reference manual, 2011.

[301] Raspberry Pi 3. Raspberry Pi 3 SDHost Driver: SDEDM_FSM. https://github.com/raspberrypi/linux/blob/rpi-5.10.y/drivers/mmc/host/bcm2835-sdhost.c.

[302] Linux. USB Linux Kernel Driver: OTG-FSM. https://github.com/raspberrypi/linux/blob/rpi-5.10.y/drivers/usb/common/usb-otg-fsm.c.

[303] Joseh Yiu. White paper: Software based Finite State Machine (FSM) with general purpose processors.

[304] Bcm4329 preliminary data sheet. https://www.mouser.jp/datasheet/2/100/Radio%20with%20Integrated%20Bluetooth%202.1%20_%20EDR%20and%20FM%20T-961654.pdf. (Accessed on 02/21/2022).

[305] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, number CONF, 2009.

[306] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 246–256. IEEE, 2013.

[307] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR. CHECKER: A soundy analysis for linux kernel drivers. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1007–1024. USENIX Association, 2017.

[308] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In Rajiv Gupta and Todd C. Mowry, editors, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 265–278. ACM, 2011.

[309] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 291–307. USENIX Association, 2018.

[310] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. DECAF++: elastic whole-system dynamic taint analysis. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*, pages 31–45. USENIX Association, 2019.

[311] Lok Kwong Yan and Heng Yin. Sok: On the soundness and precision of dynamic taint analysis. *Formal. Taint*, 2017:1–15, 2017.

[312] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. Symdrive: Testing drivers without devices. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 279–292. USENIX Association, 2012.

[313] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with revnic. In *Proceedings of the 5th European conference on Computer systems*, pages 167–180, 2010.

[314] Heejin Park, Shuang Zhai, Long Lu, and Felix Xiaozhu Lin. Streambox-tz: Secure stream analytics at the edge with trustzone. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 537–554, Renton, WA, July 2019. USENIX Association.

[315] Cve-2016-5195: Race condition in mm/gup.c in the linux kernel 2.x through 4.x before 4.8.3 allows local users to gain privileges by leveraging incorrect handling of a copy-on-write (cow) feature to write to a read-only memory mapping, as exploited in the wild in october 2016, aka dirty cow.: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195. (Accessed on 09/22/2021).

[316] Karsten Nohl and Jakob Lell. Badusb - on accessories that turn evil. *Black Hat USA*, 1(9):1–22, 2014.

[317] CVE-2016-7389. Available from MITRE, CVE-ID CVE-2016-7389., September 9 2016.

[318] Cve-2016-6775 : An elevation of privilege vulnerability in the nvidia gpu driver could enable a local malicious application to execute arbitrary code. https://www.cvedetails.com/cve/CVE-2016-6775/. (Accessed on 09/22/2021).

[319] Cve-2016-9120 : Race condition in the ion_ioctl function in drivers/staging/android/ion/ion.c in the linux kernel before 4.6 allows loca. https://www.cvedetails.com/cve/CVE-2016-9120/. (Accessed on 09/22/2021).

[320] Rpmb file system performance · issue #1033 · op-tee/optee_os. https://github.com/OP-TEE/optee_os/issues/1033. (Accessed on 10/09/2021).

[321] Keystone — an open framework for architecting tees. https://keystone-enclave.org/. (Accessed on 02/10/2022).

[322] Samuel Weiser and Mario Werner. SGXIO: generic trusted I/O path for intel SGX. In Gail-Joon Ahn, Alexander Pretschner, and Gabriel Ghinita, editors, *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, pages 261–268. ACM, 2017.

[323] Trusty tee — android open source project. https://source.android.com/security/trusty. (Accessed on 03/13/2022).

[324] Guard your data with the qualcomm snapdragon mobile platform. https://www.qualcomm.com/media/documents/files/guard-your-data-with-the-qualcomm-snapdragon-mobile-platform.pdf. (Accessed on 03/13/2022).

[325] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 33–50. USENIX Association, 2018.

[326] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 267–280. USENIX Association, 2008.

[327] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, December 2003.

[328] Ang Chen, W. Brad Moore, Hanjun Xiao, Andreas Haeberlen, Linh Thi Xuan Phan, Micah Sherr, and Wenchao Zhou. Detecting covert timing channels with time-deterministic replay. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 541–554, Broomfield, CO, October 2014. USENIX Association.

[329] 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)*, pages 1–14, 2016.

[330] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1705–1722, Baltimore, MD, August 2018. USENIX Association.

[331] 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)*, pages 72–81, 2013.

[332] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 571–582, New York, NY, USA, 2016. Association for Computing Machinery.

[333] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, Santa Clara, CA, July 2015. USENIX Association.

[334] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[335] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. Binrec: dynamic binary lifting and recompilation. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 36:1–36:16. ACM, 2020.

[336] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In Niels Provos, editor, *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*. USENIX Association, 2007.

[337] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 923–934, New York, NY, USA, 2016. Association for Computing Machinery.

[338] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. Occlumency: Privacy-preserving remote deep-learning inference using sgx. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, New York, NY, USA, 2019. Association for Computing Machinery.

[339] SeungSeob Lee, Hang Shi, Kun Tan, Yunxin Liu, SuKyoung Lee, and Yong Cui. S2net: Preserving privacy in smart home routers. *IEEE Trans. Dependable Secur. Comput.*, 18(3):1409–1424, 2021.

[340] Heejin Park, Shuang Zhai, Long Lu, and Felix Xiaozhu Lin. Streambox-tz: Secure stream analytics at the edge with trustzone. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 537–554. USENIX Association, 2019.

[341] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. Secloak: ARM trustzone-based mobile peripheral control. In Jörg Ott, Falko Dressler, Stefan Saroiu, and Prabal Dutta, editors, *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2018, Munich, Germany, June 10-15, 2018*, pages 1–13. ACM, 2018.

[342] Mengmei Ye, Jonathan Sherman, Witawas Srisa-an, and Sheng Wei. Tzslicer: Security-aware dynamic program slicing for hardware isolation. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2018, Washington, DC, USA, April 30 - May 4, 2018*, pages 17–24, 2018.

[343] Latanya Sweeney. K-anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

[344] Pravin Shankar, Vinod Ganapathy, and Liviu Iftode. Privately querying location-based services with sybilquery. In *UbiComp 2009: Ubiquitous Computing, 11th International Conference, UbiComp 2009, Orlando, Florida, USA, September 30 - October 3, 2009, Proceedings*, pages 31–40, 2009.

[345] Madhav Suresh, Zuohao She, William Wallace, Adel Lahlou, and Jennie Rogers. Kloakdb: A platform for analyzing sensitive data with k-anonymous query processing. *CoRR*, abs/1904.00411, 2019.

[346] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[347] Sushil Jajodia, Anup K. Ghosh, Vipin Swarup, Cliff Wang, and Xiaoyang Sean Wang, editors. *Moving Target Defense - Creating Asymmetric Uncertainty for Cyber Threats*, volume 54 of *Advances in Information Security*. Springer, 2011.

[348] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM trustzone to build a trusted language runtime for mobile applications. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 67–80, 2014.

[349] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 105–120. USENIX, 2005.

[350] Ming-Chang Yang, Yu-Ming Chang, Che-Wei Tsao, Po-Chun Huang, Yuan-Hao Chang, and Tei-Wei Kuo. Garbage collection and wear leveling for flash memory: Past and future. In *International Conference on Smart Computing, SMARTCOMP 2014, Hong Kong, China, November 3-5, 2014*, pages 66–73. IEEE Computer Society, 2014.

[351] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. *Trans. Storage*, 10(1):3:1–3:32, January 2014.

[352] Kernel.org bugzilla - bug list. https://bugzilla.kernel.org/buglist.cgi?chfield=%5BBug%20creation%5D&chfieldfrom=7d.

[353] Jon Geater. Usable hardware security for android on arm devices. page 35, 2012.

[354] Donald Lewine. *POSIX programmers guide.* " O'Reilly Media, Inc.", 1991.

[355] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 640–656, Washington, DC, USA, 2015. IEEE Computer Society.

[356] Shujie Cui, Sana Belguith, Ming Zhang, Muhammad Rizwan Asghar, and Giovanni Russello. Preserving access pattern privacy in sgx-assisted encrypted search. In *27th International Conference on Computer Communication and Networks, ICCCN 2018, Hangzhou, China, July 30 - August 2, 2018*, pages 1–9. IEEE, 2018.

[357] Roy Friedman and David Sainz. File system usage in android mobile phones. In *Proceedings of the 9th ACM International on Systems and Storage Conference, SYSTOR 2016, Haifa, Israel, June 6-8, 2016*, pages 16:1–16:11. ACM, 2016.

[358] Saba Eskandarian and Matei Zaharia. Oblidb: Oblivious query processing using hardware enclaves, 2017.

[359] Qingqing Cao, Noah Weber, Niranjan Balasubramanian, and Aruna Balasubramanian. Deqa: On-device question answering. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 27–40, 2019.

[360] Weizhe Hua, Zhiru Zhang, and G. Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 4:1–4:6. ACM, 2018.

[361] ROS developers. Ros: Recording and playing back data. http://wiki.ros.org/Bags/Format, 2020.

[362] Nitin Agrawal, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Emulating goliath storage systems with david. In *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, pages 203–216, 2011.

[363] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, Boston, MA, 2017. USENIX Association.

[364] Kevin D. Fairbanks, Christopher P. Lee, and Henry L. Owen III. Forensic implications of ext4. In Frederick T. Sheldon, Stacy J. Prowell, Robert K. Abercrombie, and Axel W. Krings, editors, *Proceedings of the 6th Cyber Security and Information Intelligence Research Workshop, CSIIRW 2010, Oak Ridge, TN, USA, April 21-23, 2010*, page 22. ACM, 2010.

[365] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.

[366] Jan Beirlant, Edward J Dudewicz, László Györfi, Edward C Van der Meulen, et al. Nonparametric entropy estimation: An overview. *International Journal of Mathematical and Statistical Sciences*, 6(1):17–39, 1997.

[367] Edmund B. Nightingale and Jason Flinn. Energy-efficiency and storage flexibility in the blue file system. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 363–378. USENIX Association, 2004.

[368] SQLite: Speedtest1.c. https://sqlite.org/src/file/test/speedtest1.c, 2019. (Accessed on 08/16/2019).

[369] Apache lucy. http://lucy.apache.org/, 2020. (Accessed on 11/24/2020).

[370] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi, editors, *Machine Learning: ECML 2004, 15th European Conference on Machine Learning, Pisa, Italy, September 20-24, 2004, Proceedings*, volume 3201 of *Lecture Notes in Computer Science*, pages 217–226. Springer, 2004.

[371] Youtube live streaming: Downtown bangor. https://www.youtube.com/watch?v=LIQnvi2FmUg, 2021. (Accessed on 05/06/2021).

[372] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *CoRR*, abs/1410.7747, 2014.

[373] David Wheeler. Sloccount. *http://www. dwheeler. com/sloccount/*, 2001.

[374] Min Hong Yun and Lin Zhong. Ginseng: Keeping secrets in registers when you distrust the operating system. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[375] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: The missing OS abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 1:1–1:17, 2019.

[376] Deeksha Dangwal, Weilong Cui, Joseph McMahan, and Timothy Sherwood. Safer program behavior sharing through trace wringing. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 1059–1072. ACM, 2019.

[377] Raspberry pi 3 model b+. https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/, 2021. (Accessed on 04/22/2021).

[378] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, 2015. USENIX Association.

[379] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In Keith A. Smith and Yuanyuan Zhou, editors, *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013*, pages 73–80. USENIX, 2013.

[380] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 299–310, 2013.

[381] Kisung Lee and Youjip Won. Smart layers and dumb result: IO characterization of an android-based smartphone. In Ahmed Jerraya, Luca P. Carloni, Florence Maraninchi, and John Regehr, editors, *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*, pages 23–32. ACM, 2012.

[382] Sandeep Chinchali, Apoorva Sharma, James Harrison, Amine Elhafsi, Daniel Kang, Evgenya Pergament, Eyal Cidon, Sachin Katti, and Marco Pavone. Network offloading policies for cloud robotics: a learning-based approach. *arXiv preprint arXiv:1902.05703*, 2019.

[383] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 41–46. Association for Computing Machinery, 2011.

[384] Ruth Brand. Microdata protection through noise addition. In *Inference control in statistical databases*, pages 97–116. Springer, 2002.

[385] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association.

[386] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst.*, 33(3):8:1–8:26, August 2015.

[387] Chia-che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 645–658, 2017.

[388] Shweta Shinde, Shengyi Wang, Pinghai Yuan, Aquinas Hobor, Abhik Roychoudhury, and Prateek Saxena. Besfs: A POSIX filesystem for enclaves with a mechanized safety proof. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 523–540. USENIX Association, 2020.

[389] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB linux applications with SGX enclaves. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, 2017.

[390] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. Bitvisor: a thin hypervisor for enforcing i/o device security. In Antony L. Hosking, David F. Bacon, and Orran Krieger, editors, *Proceedings of the 5th International Conference on Virtual Execution Environments, VEE 2009, Washington, DC, USA, March 11-13, 2009*, pages 121–130. ACM, 2009.

[391] Aravind Menon, Simon Schubert, and Willy Zwaenepoel. Twindrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest OS drivers. In Mary Lou Soffa and Mary Jane Irwin, editors, *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, pages 301–312. ACM, 2009.