TZ-LLM: Protecting On-Device Large Language Models with Arm TrustZone

Xunjie Wang*, Jiacheng Shi*, Zihan Zhao, Yang Yu, Zhichao Hua, Jinyu Gu✉
Institute of Parallel and Distributed Systems, School of Computer Science, Shanghai Jiao Tong University

Abstract

Large Language Models (LLMs) deployed on mobile devices offer benefits like user privacy and reduced network latency, but introduce a significant security risk: the leakage of proprietary models to end users.

To mitigate this risk, we propose a system design for protecting on-device LLMs using Arm Trusted Execution Environment (TEE), TrustZone. Our system addresses two primary challenges: (1) The dilemma between memory efficiency and fast inference (caching model parameters within TEE memory). (2) The lack of efficient and secure Neural Processing Unit (NPU) time-sharing between Rich Execution Environment (REE) and TEE.

Our approach incorporates two key innovations. First, we employ *pipelined restoration*, leveraging the deterministic memory access patterns of LLM inference to prefetch parameters on demand, hiding memory allocation, I/O and decryption latency under computation time. Second, we introduce a *co-driver* design, creating a minimal data plane NPU driver in the TEE that collaborates with the full-fledged REE driver. This reduces the TEE TCB size and eliminates control plane reinitialization overhead during NPU world switches.

We implemented our system on the emerging OpenHarmony OS and the llama.cpp inference framework, and evaluated it with various LLMs on an Arm Rockchip device. Compared to a strawman TEE baseline lacking our optimizations, our system reduces TTFT by up to 90.9% and increases decoding speed by up to 23.2%.

CCS Concepts: • **Security and privacy → Trusted computing**; • **Software and its engineering → Operating systems**.

Keywords: Large Language Model, Mobile computing, Arm TrustZone

* The two authors contributed equally to this work.

EUROSYS '26, Edinburgh, Scotland UK

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769334

ACM Reference Format:
Xunjie Wang, Jiacheng Shi, Zihan Zhao, Yang Yu, Zhichao Hua, Jinyu Gu. 2026. TZ-LLM: Protecting On-Device Large Language Models with Arm TrustZone. In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland UK*. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3767295.3769334

1 Introduction

Intelligent applications based on Large Language Models (LLMs), such as digital assistants, text refinement, and multimodal understanding, have been increasingly deployed on mobile devices [4, 6, 26]. Compared to cloud-based LLMs, on-device LLMs can leverage the semantics of personal data without transmitting it to the cloud, which helps maintain user data privacy, reduce network latency, and lower the cost of LLM inference. However, on-device LLMs introduce a new security challenge because the proprietary models are stored on personal devices. A curious user or a malicious application may compromise the mobile OS to steal the models. Although some model providers protect their models by encrypting the model files, the plaintext model in memory remains at risk during the inference process [77].

Arm TrustZone [25], widely deployed on mobile devices, enforces isolation between a Rich Execution Environment (REE), which runs untrusted applications and a traditional OS, and a Trusted Execution Environment (TEE) for trusted applications (TAs) and a minimal TEE OS. It is intuitive to run LLM inference in the TEE to protect the models. However, traditional TEE software stacks, such as OP-TEE [15], are designed for lightweight TAs that typically require small computational resources. How to efficiently provide the memory and Neural Processing Unit (NPU) resources for LLM inference in the TEE is challenging and lacks research.

This paper introduces TZ-LLM, a system designed to efficiently protect the confidentiality of on-device LLMs with TEE. TZ-LLM overcomes the following two challenges.

The first challenge is a dilemma between memory efficiency and LLM startup time (time-to-first-token, TTFT). On the one hand, if the TEE reserves a static amount of secure memory for caching LLM parameters, the LLM inference can start immediately. However, the memory usage is inefficient due to the large size of LLM parameters (e.g., 8GB for 8-bit quantized Llama-3-8B). Mobile devices are resource constrained (≤24GB memory for commodity smartphones).

On the other hand, scaling secure memory for the LLM on demand results in a long TTFT, because the system needs to

expand secure memory and load LLM parameters from flash storage before the LLM starts inference. Due to the inherent requirement of TrustZone memory protection [2], the system must allocate large contiguous physical memory from REE, which is time-consuming [54, 62, 72]. Meanwhile, confidentiality protection requires model files to be encrypted, resulting in decryption overhead on model loading, in addition to I/O overhead. The allocation, I/O and decryption overheads can increase the TTFT by 11.6s for Llama-3-8B.

The second challenge is the lack of an efficient and secure mechanism for NPU time-sharing between TEE and REE. Existing work has shown that LLM performance can improve by up to 7.3× with NPU [85, 88], while the applications in REE also require NPU for various functionalities [19]. It is feasible to deploy two separated NPU drivers in REE and TEE and dynamically switch the NPU between the two worlds. However, porting the REE NPU driver to TEE will lead to both performance and security concerns. First, the REE driver and its dependencies on the REE OS consist of a large code base (e.g., 60K for the Rockchip NPU [22]), which can significantly bloat the TCB in TEE. Second, when switching the NPU between the two worlds, the REE or TEE driver needs to be reinitialized, incurring a switching overhead (e.g., 32ms for the Rockchip NPU).

To address the first challenge, TZ-LLM proposes *pipelined restoration* to overlap I/O, decryption, memory allocation, and inference, which allows it to dynamically scale secure memory while reducing the overhead on TTFT. The insight is that the memory usage pattern of the DAG-based (directed acyclic graph) inference computation is deterministic, allowing TZ-LLM to accurately prefetch parameters during inference. Specifically, TZ-LLM incrementally extends the contiguous secure memory region with CMA [1] and loads parameters in the topological order of the DAG, while executing inference operators in parallel when the required parameters are loaded. To minimize pipeline bubbles, TZ-LLM uses the following two techniques. ① *Priority-based pipeline scheduling*: When multiple restoration or computation operators compete for the CPUs, TZ-LLM prioritizes the most urgent task that could stall the critical path of the LLM inference. ② *Partial parameter caching*: If memory is sufficient when the LLM is idle, TZ-LLM caches some parameters used by early prefill operators in the secure memory, allowing the next inference to start immediately. The parameters are released in reverse topological order after inference, preserving the contiguity of the secure memory region.

To address the second challenge, TZ-LLM uses a *co-driver* design that separates a data plane driver for TEE from the original NPU driver. The insight is that the workflow of an NPU job is simple and can be secured without relying on the control plane, so that the TEE driver can be tailored (about 1K LoC) and an NPU world switch does not require the reinitialization of the control plane. The TEE driver is only responsible for launching secure NPU jobs, while the REE driver handles control plane operations like NPU job scheduling and device frequency management. The two drivers cooperate: When the REE driver schedules a TEE NPU job, it asks the TEE driver to take over the NPU by configuring the TrustZone hardware and execute the job.

We prototype TZ-LLM based on OpenHarmony (a widely deployed production-grade mobile OS) [16] and the llama.cpp inference framework [13], and evaluate it with various models [12, 28, 89, 95] including Qwen2.5-3B and Llama-3-8B and various real-world benchmarks [36, 52, 83] on an Orange Pi board [17] (Rockchip 3588 CPU/NPU [21]). We compare TZ-LLM to a strawman TEE baseline without pipelined restoration and NPU support, and to an REE baseline optimized with pipelined restoration. Compared to the TEE baseline, TZ-LLM reduces TTFT by 76.1%∼90.9% and increases decoding speed by 0.9%∼23.2%. Compared to the REE baseline, TZ-LLM incurs an average overhead of 5.2%∼28.3% and 1.3%∼4.9% on TTFT and decoding speed, respectively, across different models and benchmarks.

In summary, this paper has the following contributions:

- We identify the challenges of efficient secure memory scaling and TEE-REE NPU time-sharing that hinders efficient LLM inference in TrustZone.
- We address these challenges and make the first attempt to use TrustZone to protect the confidentiality of on-device LLMs.
- Our preliminary evaluations show the promising performance of our system.

2 Background and Motivation

2.1 Confidentiality of On-Device LLMs

Modern mobile applications can automatically incorporate user data into LLM prompts to generate personalized responses [4]. Instead of using cloud systems, there is a growing trend to run LLM inference on mobile devices [29, 85, 88], as it eliminates the network latency of querying cloud services and keeps users' private data on their devices.

However, storing LLM parameters on mobile devices introduces the risk of leaking the proprietary model to untrusted users, as mobile devices are prone to jailbreaking attacks. Model leakage can result in significant financial losses for the model provider, as the development of such models may cost millions of dollars [77]. Additionally, the leakage could severely undermine the model provider's advantage in the highly competitive LLM market [5].

According to the prior study [77], most mobile applications leave their on-device models completely unprotected, while others only encrypt model files but still allow attackers to extract plaintext model parameters from memory.

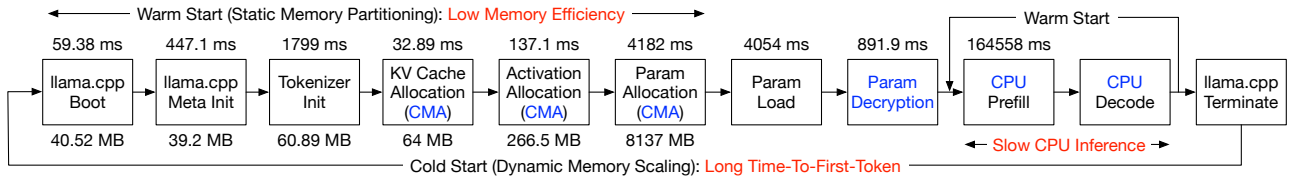

Figure 1. A strawman workflow of LLM inference in TEE (§7 testbed, 8-bit Llama-3-8B, 512-token prompt). Time and memory usage for each step are shown above and below each box. Red texts: challenges. Blue texts: overheads related to TEE protection.

2.2 Arm TrustZone

Our work uses Arm TrustZone [25], a widely deployed hardware isolation mechanism on mobile devices, to protect LLMs. TrustZone separates hardware resources into a Rich Execution Environment (REE) and a Trusted Execution Environment (TEE). The TEE hosts security-critical trusted applications (TAs) and a minimal TEE OS, while the REE runs untrusted applications and a full-fledged OS like Linux.

TrustZone divides the CPU into a secure state and a non-secure state. Software can switch CPU states by calling a security monitor running in EL3 using a Secure Monitor Call (*smc*) instruction. To enforce memory isolation, a TrustZone Address Space Controller (TZASC) protects eight contiguous physical memory regions as secure memory, which cannot be accessed by non-secure CPUs. Peripheral devices are also classified as secure devices and non-secure devices. A TrustZone Protection Controller (TZPC) prohibits any MMIO access to secure devices from non-secure CPUs. The TZASC controls the DMA permission of each device, only allowing secure devices to access secure memory. Moreover, TrustZone directs interrupts from secure devices to the TEE OS with an extension in the generic interrupt controller (GIC).

TrustZone can only protect contiguous physical memory, but contiguous memory allocation at runtime is challenging due to fragmentation [57]. Therefore, existing TEEs typically reserve secure memory at system boot. The Linux kernel provides a Contiguous Memory Allocator (CMA) [1], which reserves a physical memory region. The buddy system can allocate pages from this region, but only *movable* pages can be placed in it. To preserve contiguity, CMA migrates *movable* pages out of the region as follows: the kernel allocates a new destination page outside CMA, unmaps the old page, copies its data to the new page, updates the page table mapping, and releases the old page for CMA allocation.

2.3 Challenges of LLM Inference in TEE

As illustrated in Figure 1, running LLM efficiently in TEE faces the following two challenges.

Challenge #1: The dilemma between memory efficiency and fast inference. Traditional TEEs [15] statically partition memory as secure and non-secure at system boot. However, the LLM requires a large amount of memory for parameters, KV cache, activation, and other data (8.4GB in Figure 1). Using a large secure memory will result in memory shortage in REE as mobile devices are typically resource constrained.

Therefore, the secure memory should be dynamically scaled up and down as the LLM inference starts and completes. However, when scaling up secure memory, a naive "cold start" workflow (Figure 1) for restarting LLM Trusted Application (TA) will incur high overhead on LLM TTFT. This overhead includes the following parts: (1) The inference framework initializes, parses model metadata and creates the tokenizer (2.3s). (2) The TEE allocates memory from REE. Due to the limitation of TZASC, it must allocate contiguous physical memory using Linux CMA, causing high memory migration overhead if the CMA region is occupied (up to 4.2s for 8GB parameters). (3) The system loads LLM parameters from the flash storage. Since the file system is accessible to the untrusted REE applications and OS, the model files must be encrypted, resulting in decryption overhead during loading (0.9s for 8GB parameters). The total cold start overhead is 11.6s in Figure 1.

Thus, a mechanism is needed to *minimize the overhead on TTFT* caused by dynamic scaling of secure memory.

Challenge #2: The lack of efficient and secure NPU time-sharing between REE and TEE. NPUs are widely deployed on mobile devices to support applications such as object detection, OCR, and photo refinement [19]. Since these applications typically run in the REE, the NPU is statically configured as a non-secure device at boot time. However, this design significantly hinders LLM performance in the TEE. As shown in Figure 1, the LLM prefill using CPU takes 164s. Prior work has shown that using the Qualcomm NPU [20] can increase LLM prefill speed by 7.3× compared to the optimal CPU implementation [85]. Our evaluation also shows that the Rockchip NPU provides 12.5× and 1.3× optimizations on the prefill and decoding speed of Llama-3-8B, respectively.

It is intuitive to share the NPU between REE and TEE by deploying one driver in each world. The NPU can be switched between the two worlds by detaching it from one driver and attaching it to another driver. However, this approach has two limitations: (1) The detach-attach incurs substantial switching overhead as it requires full driver reinitialization. The detach-attach of a Rockchip NPU with the Linux driver takes 32ms. The overhead mainly stems from control plane

Table 1. Comparison of existing TEE-based model protection approaches with TZ-LLM.

Approach	Performance		End-to-end Security	Compatibility		Memory Scaling
	Overall	Accelerator Usage		No Model Modification	Quantization Support	
Shielding the entire model[1]	★	No	✔	✔	✔	✘
Obfuscation-based TSLP[2]	★★	REE only	✘	✔	✘	✘
TSQP [75]	★★	REE only	✘	✘	✔	✘
TEESlice [97]	★★	REE only	✘	✘	✘	✘
StrongBox [33]	★★	TEE-REE sharing	✘	✔	✔	✘
SecDeep [66]	★★	TEE only	✔	✔	✔	✘
TZ-LLM (ours)	★★★	TEE-REE sharing	✔	✔	✔	✔

[1] Shielding the entire model [42, 47, 51, 55, 61, 90]. [2] Obfuscation-based TSLP [63, 73, 76, 81, 96].

operations, including NPU power/frequency configuration and interaction with the Linux device framework. (2) Deploying the full-fledged NPU driver, which highly depends on the REE OS, in the TEE bloats the TCB. The Linux driver for Rockchip NPU relies on several Linux subsystems, like device, memory, interrupt, and power management, and the total code base is estimated to be over 60K LoC.

Thus, a mechanism for NPU time-sharing between REE and TEE is needed to *reduce NPU world switching overhead* and *minimize the additional TCB in TEE.*

2.4 Existing Approaches Studies

2.4.1 TEE-based Model Protection. As shown in Table 1, extensive prior work has explored protecting on-device models with TEEs such as Arm TrustZone and Intel SGX. Some work [42, 47, 51, 55, 61, 90] shields the entire model within an accelerator-absent TEE to protect all model parameters and the inference framework. Although these approaches offer end-to-end security guarantees, they only use CPU for inference and incur significant overhead. Consequently, a line of work seeks to mitigate this overhead.

TEE-Shielded LLM Partition (TSLP). TSLP solutions partition models and offload a part of the parameters to REE accelerators for computation. Some TSLP approaches [63, 73, 76, 81, 96] enhance security by obfuscating the offloaded parameters. However, TSQP [75] points out that these approaches are incompatible with quantization, while quantization significantly reduces memory footprint and is well-suited for mobile devices [86, 87]. TSQP enables quantization through quantization-aware model training. Nonetheless, model stealing attacks using public pretrained models can compromise the security of these obfuscation-based TSLP solutions [97]. TEESlice [97] counters this threat with privacy-aware model training, offloading only privacy-irrelevant parameters to REE accelerators. However, it requires model modification, and it still leaves part of the parameters outside the TEE, failing to provide end-to-end security guarantees. In addition, these solutions also incur extra data copying between the REE and TEE, as well as additional computation for deobfuscation or privacy-related parameters.

Accelerator-enabled TEE. Other work attempts to enable accelerators within the TEE. StrongBox [33] builds a GPU TEE with TrustZone by deploying the a single GPU driver in the REE for both secure and non-secure jobs. While it supports page-grained secure memory protection with S2PT, it reserves a static secure memory region and resorts to TZASC for DMA protection, which is not memory efficient. Moreover, it does not safeguard the integrity of the inference framework in the REE, thus lacking end-to-end security guarantees. SecDeep [66] statically configures accelerators as TrustZone secure devices, which restricts the REE functionalities. These approaches also incur frequent encryption and decryption overhead when data is swapped between secure and non-secure memory during inference.

In conclusion, existing TEE-based model protection systems fail to meet performance, security, and compatibility requirements simultaneously, primarily because they lacks REE-TEE accelerator time-sharing or dynamic secure memory scaling for effcient model inference inside the TEE.

2.4.2 Elastic Secure Memory Protection. Previous work uses Stage-2 Page Tables (S2PT) for memory protection at page granularity [33, 46, 48]. Specifically, they run the untrusted OS and applications inside a VM and unmap the secure memory pages from the S2PT. Although this design could support elastic secure memory scaling without the overhead of contiguous memory allocation, we conduct preliminary experiments on the testbed in §7 to demonstrate why we choose not to adopt it.

S2PT incurs continuous overhead on REE applications, while the overhead of CMA allocation is transient. The CPU running REE applications must perform a two-dimensional page table walk for each TLB miss [31, 41, 93]. Although using 2MB or 1GB huge pages in the S2PT reduces overhead, most mappings fall back to 4KB granularity after allocating memory for the LLM due to memory fragmentation. Figure 2 shows that stage-2 translation with 4KB mappings can incur a maximum overhead of 9.8% on Geekbench applications [7], and the average overhead is 2.0%.

Although stage-2 translation can be disabled to avoid the overhead when the LLM is idle, this disables memory protection, requiring all secure memory to be cleaned. To mitigate model loading overhead, parameters can be cached in S2PT-protected memory, at the cost of *continuous* overhead on REE applications.

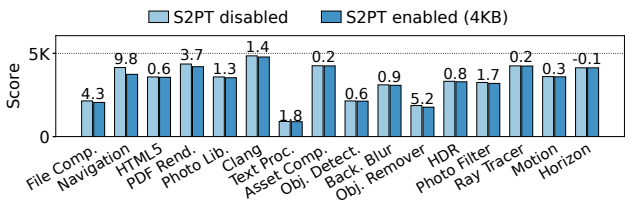

Figure 2. Geekbench scores with S2PT enabled or disabled. The texts are the overheads caused by S2PT (%).

In contrast, while migrating pages from the CMA region also imposes overhead on REE applications, the overhead is *transient* and only exists at the beginning of LLM inference.

CMA allocation overhead is small under low memory pressure, and can be hidden under high memory pressure. We evaluate the time for allocating 8GB memory for 8-bit Llama-3-8B using CMA and buddy system (4KB), respectively. To assess the overhead of page migration, we use stress-ng [23], which maps a portion of memory and generates pressure by running multiple sophisticated memory testing algorithms on the mapped region. Figure 3 shows the results.

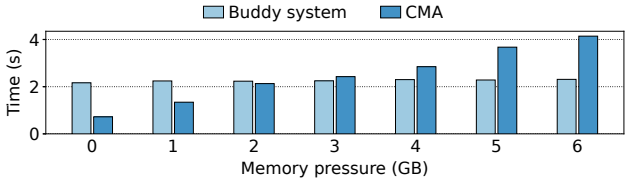

Figure 3. Memory allocation time for Llama-3-8B (8GB) using buddy system or CMA, at different memory pressures.

Under high memory pressure, the CMA allocation throughput is 1.9GB/s, which is similar to the I/O throughput of sequential reads on our platform (2GB/s). Moreover, by using multi-threading, the CMA allocation throughput can reach 3.8GB/s (4 threads). Therefore, we can hide the allocation overhead under the latency of reading the model file.

S2PT protection cannot prevent DMA attacks. S2PT does not control DMA permissions. To prevent DMA attacks on S2PT-protected secure memory, a privileged monitor like the EL3 monitor must intercept every IOMMU configuration operation and unmap the secure memory from I/O page tables [48], or intercept every MMIO operation and verify the DMA addresses [46]. Both designs introduce monitoring overhead on REE and extend the privileged TCB.

3 Overview

3.1 Threat Model

Attack vectors. We consider an attacker attempting to steal on-device LLM parameters, or intermediate inference results that may help model theft, such as activations and KV cache. The attacker might extract parameters by directly accessing memory/flash or exploiting peripheral devices to initiate malicious DMA requests. Attackers may also try to induce the inference framework to exfiltrate model parameters, by exploiting TEE-REE interfaces for Iago attacks (e.g., breaking the integrity of secure NPU jobs). Physical attacks on memory confidentiality are not considered because TrustZone does not enforce memory encryption, and it can be addressed with future hardware [10]. Side-channel and cryptographic attacks fall outside our scope as they are orthogonal to our concern and can be defended with complementary techniques. Denial-of-service (DoS) attacks are also out-of-scope as they do not compromise model confidentiality.

Trusted computing base (TCB). We trust the TEE OS, the TEE NPU driver, and the inference framework (LLM TA). Arm TrustZone hardware, EL3 monitor, and NPU hardware are also trusted. The integrity of these components can be guaranteed with secure boot. Other components within the TEE, such as other TAs and other secure devices, are not trusted. All components in the REE are excluded from the TCB, including the REE OS, the full-fledged REE NPU driver, REE applications, and non-secure peripheral devices.

3.2 System Architecture

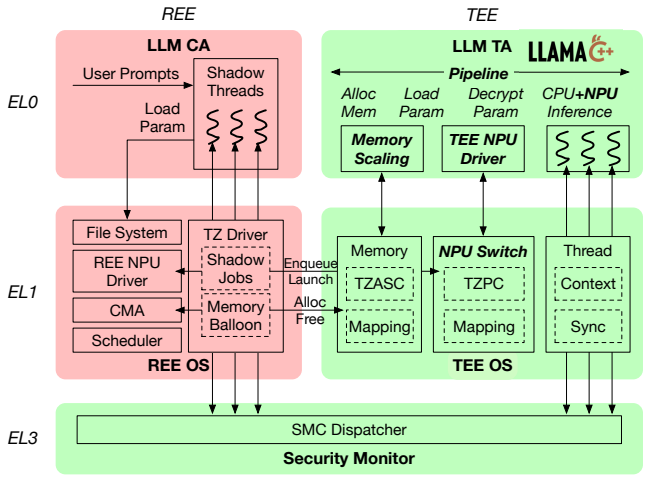

Figure 4. TZ-LLM architecture, S/N: secure/non-secure.

We propose TZ-LLM, a system for protecting on-device LLMs using Arm TrustZone. As shown in Figure 4, TZ-LLM runs the LLM inference framework (e.g. llama.cpp) as a TA, which can be invoked by a client application (CA) in the REE through the TrustZone (TZ) driver in the REE OS (Linux).

The TZ driver also enables interactions between the TEE OS and the CA, Linux CMA, and REE NPU driver to delegate model loading, memory scaling, and NPU job scheduling.

Our design assumes a mobile platform with a hardware platform supporting Arm TrustZone and a software platform consisting of a REE OS with a TEE OS. These assumptions are generally applicable across mobile devices.

Addressing challenge #1: Elastic memory scaling with pipelined restoration. The LLM TA can extend or release secure memory using interfaces provided by the TEE OS. When extending the secure memory, the TEE OS asks the TZ driver to allocate memory from Linux CMA (memory ballooning) and protects it by configuring TZASC. The extended memory can be released with a reverse process.

To mitigate the parameter restoration overhead (allocation, I/O, and decryption) on LLM TTFT (Figure 1), TZ-LLM runs these processes in parallel with LLM inference. The insight for this design is the *determinism* of the memory access pattern of LLMs. Specifically, the computation graph of the LLM is a DAG, in which each node represents an operator like matrix multiplication, and the inference framework schedules the operator in the topological order of the DAG. Each operator only uses a portion of LLM parameters, e.g., operators in LLM layer 1 only use parameters of layer 1. Therefore, when handling one operator, the inference framework can accurately know which parameters will be accessed next and prefetch these parameters in parallel.

If an LLM operator is ready, but the parameters have not been restored, or the hardware is busy, the operator will be blocked, leading to pipeline bubbles. TZ-LLM designs two techniques to minimize such bubbles (§4.1). First, the pipeline is scheduled using a priority-based and preemptive mechanism that prioritizes the most urgent task in the pipeline that may lead to a bubble. Second, the LLM TA uses a partial caching mechanism that gradually releases memory based on the REE memory pressure after the inference is done, and the parameters remaining in memory can be used by the next inference without restoration.

With partial parameter caching, the TA must ensure that the cached secure memory is contiguous. Fortunately, it is optimal to cache the parameters used early during inference, so that the secure memory is released in the reverse topological order of the DAG. This first-in-last-out allocation-deallocation pattern aligns well with the contiguity requirement. We design an *"extend and shrink"* secure memory management interface based on this pattern (§4.2).

Addressing challenge #2: TEE-REE NPU time-sharing with control-data separation. The LLM TA can issue secure NPU jobs with a TEE NPU driver. The TEE and the REE multiplex the NPU with time-sharing. An REE application can run NPU jobs during LLM inference.

For secure and efficient TEE-REE NPU time-sharing, we observe that the workflow of an NPU job (data plane), including setup, launching, and completion, forms a small and self-contained closure. The functionality and security of this workflow does not depend on the control plane state of the full-fledged NPU driver, such as scheduling or power management. This property allows TZ-LLM to use a *co-driver* design (§4.3) by integrating only the tiny data plane of the NPU driver into the TEE, which cooperates with the control plane in the REE NPU driver. Therefore, most control plane code and dependencies can be tailored from the TEE driver and the NPU can switch between the two worlds without reinitializing the control plane. The REE driver manages the unified scheduling of both secure and non-secure NPU jobs, delegating secure jobs to the TEE driver. The TEE driver protects the confidentiality and integrity of the secure jobs based on TrustZone hardware configuration and security checks.

Other techniques for efficient inference. As shown in Figure 1, the initialization of framework, model metadata and tokenizer also takes a long time. We mitigate this overhead by saving a checkpoint of the initialized state in flash and restoring it on each inference request. The KV cache and activation allocation overhead is not mitigated because it is minor compared with the inference time.

In addition to NPU, on-device LLM inference also requires CPU multi-threading for acceleration, but traditional TEEs provide only one thread for each TA. TZ-LLM allows the TA to create multiple threads and schedules them using the REE scheduler. Specifically, each TA thread is paired with a shadow thread in the CA. When a shadow thread is activated, it uses *smc* to start or resume the corresponding TA thread. For security, the contexts of TA threads and the synchronization primitives are managed by the TEE OS.

The file system is managed by the REE, and the LLM TA delegates I/O requests to the CA with *smc* when loading parameters from the flash. To avoid blocking the CPU, the CA issues asynchronous I/O (aio) requests to the file system.

4 Detailed Design

4.1 Pipelined Parameter Restoration

To accelerate the cold start of LLM TA, TZ-LLM adopts a pipeline mechanism that overlaps the parameter restoration operations with the prefill-stage computation of the LLM.

Restoration operators. As shown in Figure 6, with parameter restoration, the LLM computation graph is extended by inserting three restoration operators before a prefill-stage computation operator, representing the memory allocation, parameter loading (flash I/O), and parameter decryption for restoring the parameters used by the computation operator.

The computation and restoration operators run on three types of hardware: CPU, NPU, and I/O engine. Contiguous

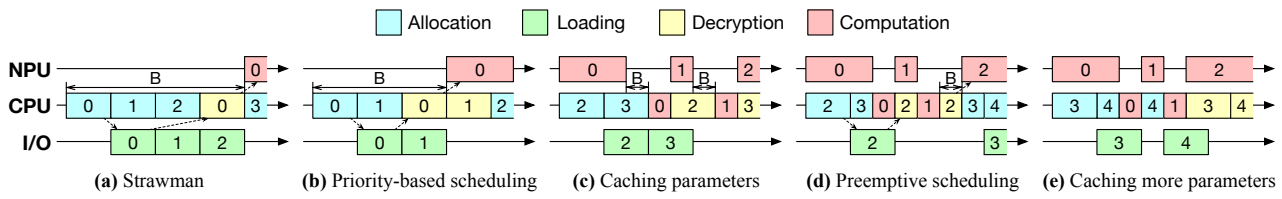

Figure 5. Pipelined restoration timelines. The figure shows the effect of different techniques for reducing bubbles. B: bubble. The number in each box denotes the index of the computation operator that the operator belongs to. The indices follow the topological order of the computation graph. The dashed arrows denote the dependencies of operators, which cause bubbles.

memory allocation (memory migration) and parameter decryption run on CPUs. Some computation operators, such as layer normalization and self-attention, run on CPUs, while others, such as matrix multiplication, run on the NPU. Parameter loading is performed by the I/O engine.

Pipeline scheduling problem. There may be multiple restoration or computation operators ready for the same hardware at the same time. For example, in Figure 6, the green checkmarks indicate that there are four operators ready for the CPUs and two operators ready for the I/O engine. The scheduling problem is to determine the execution order of the operators to minimize the TTFT. For the scheduling of the I/O engine and the NPU, it is intuitive that the best policy is to schedule the loading (I/O) and computation operators in the topological order of the computation graph. Therefore, the main problem is the scheduling of CPU operators.

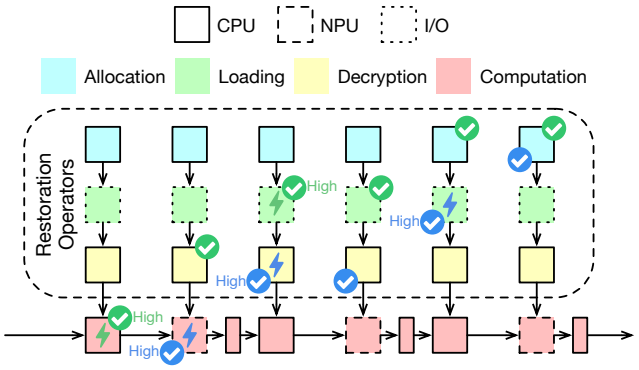

Figure 6. Pipeline scheduling examples. The arrows denote the dependencies of operators. The green and blue marks denote two different scheduling points. The checkmark denotes that the operator is ready. The lightning symbol denotes that the operator is scheduled (highest priority).

Priority-based pipeline scheduling. It is hard to find an optimal scheduling policy for CPU operators because the scheduling goal depends on the pipeline critical path, which varies across different models, prompts, and hardware. There are three potential critical paths: (1) loading (I/O) operators, (2) CPU operators, including allocation, decryption, and computation, and (3) computation operators, including CPU and NPU computation. If loading operators are the critical path, the scheduler should prioritize allocation operators to reduce bubbles on the loading path. If computation operators are the critical path, the scheduler should prioritize computation operators and make restoration operators complete early enough to prevent computation stalls. If CPU operators are the critical path, the scheduler should keep the CPU busy, reducing bubbles caused by waiting for I/O or NPU.

In practice, we observe that the critical path is usually CPU operators or computation operators, instead of loading operators. To meet the scheduling goals of these two common critical paths, TZ-LLM uses a greedy policy that schedules the CPU computation operator if it's ready, or schedules the restoration operator related to the earliest computation operator if no CPU computation operator is ready. This aligns well with the two scheduling goals because (1) it reduces computation stalls by prioritizing computation operators and earlier restoration operators, and (2) it enables CPU computation operators to be ready for scheduling early, thereby keeping the CPU busy. The evaluation shows that the performance of this policy is close to the optimal (§7.2.1).

The scheduler maintains a priority queue of ready operators and executes them according to the priority rule. As shown in Figure 5a and Figure 5b, the scheduler prioritizes decryption operator 0 over allocation operator 2, reducing the bubble before NPU computation operator 0.

Preemptive pipeline scheduling. We find that priority-based scheduling without preemption still suffers from bubbles due to the misalignment of operator execution times. As shown in Figure 5c, CPU computation operator 0 is blocked by allocation operator 3, resulting in a bubble. We eliminate such bubbles with preemptive scheduling, by dividing allocation and decryption operators into smaller micro-operators and introducing preemption points between them. As shown in Figure 5d, allocation operator 3 is preempted as soon as CPU computation operator 0 becomes ready.

Partial parameter caching. As shown in Figure 5b, the pipeline has a bubble at the beginning that waits for the first parameter tensor, regardless of the scheduling policy. To eliminate this bubble, TZ-LLM keeps some secure memory

unrevoked after inference to partially cache the plaintext parameters. With this mechanism, the next inference can resume from the computation stage of the cached parameters, avoiding full restoration. As shown in Figure 5c, by caching parameters of operators 0~1, the initial bubble is eliminated.

Sometimes TZ-LLM needs to cache more parameters, as the bottleneck of the prefill stage may shift to restoration operators when the computation time is short. For example, computation operator 2 in Figure 5d is blocked by decryption operator 2, and this bubble can be eliminated by caching parameters of operators 0~2 (Figure 5e).

It is optimal to cache the parameters used by early computation operators as the later parameters can be restored in parallel with the early computation. To this end, TZ-LLM lazily releases secure memory in the reverse topological order of the computation graph according to the REE memory pressure. The LLM TA provides an interface to the REE OS to revoke secure memory to the REE.

Limitation. A limitation of TZ-LLM is that it may deliver suboptimal performance on non-deterministic workloads. For example, it prefetches all experts in a Mixture of Experts (MoE) model or all layers in an early-exit transformer, including parameters not used in the current inference. The cost of this additional prefetching can be amortized by future inferences that do utilize these parameters.

4.2 Pipeline-Aware Secure Memory Management

The limitation of TZASC mandates that secure memory remain contiguous during scaling. Fortunately, the memory allocation-deallocation pattern of pipelined restoration allows us to design secure memory management interfaces that effectively satisfy this requirement.

Allocation patterns and memory layouts. The LLM TA uses four types of data: LLM parameters, KV cache, activations, and others (libraries, metadata, etc.). TZ-LLM places these data in two contiguous TZASC regions.

One TZASC region is used for LLM parameters. With partial parameter caching (§4.1), LLM parameters are progressively loaded during pipelined restoration and progressively released in the reverse order of allocation after inference. As shown in Figure 7b, this first-in-last-out allocation-deallocation pattern ensures that the in-memory parameters are always stored contiguously.

Another TZASC region is used for KV cache, activations, and other data. The KV cache is initialized to the prompt size during the prefill stage, grows with the number of generated tokens during the decoding stage, and is completely released after inference. The activations and other data are fixed-size buffers allocated at inference start and released at inference completion, so that they can be placed before the KV cache without breaking the contiguity of the TZASC region.

Secure memory management interfaces. Based on the allocation patterns and memory layouts, the TEE OS provides

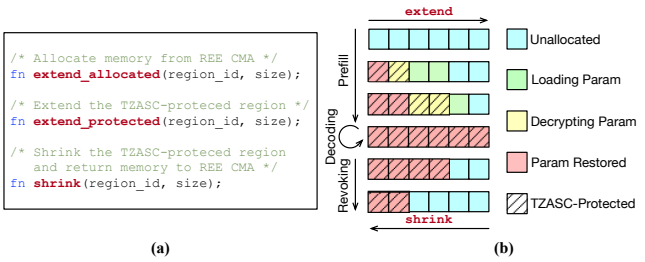

Figure 7. (a) Secure memory management interfaces, (b) memory layout of the CMA region for model parameters.

"extend and shrink" interfaces to the LLM TA for scaling TZASC regions up and down, as shown in Figure 7a.

Each TZASC region is associated with a CMA region in the REE. When extending the secure memory, the TA first calls *extend_allocated*. Then, the TEE OS asks the TZ driver to allocate memory blocks from the CMA region. To ensure the contiguity of the entire allocated memory, CMA allocates new memory blocks adjacent to the previously allocated blocks. The TEE OS verifies this requirement when it receives the allocated memory address from the TZ driver. After allocation, the TA calls *extend_protected*. The TEE OS then extends the end of the TZASC region to protect the newly allocated memory, and maps the new memory into the TA's address space. When revoking secure memory, the TA calls *shrink* to release memory from the end of the TZASC region. The TEE OS unmaps memory from the TA's address space, shrinks the TZASC region and asks the TZ driver to release memory to the CMA. The TEE OS clears all sensitive data before releasing the memory.

The separation of *extend_allocated* and *extend_protected* is designed to eliminate the need for I/O bounce buffers during parameter loading (flash I/O). As shown in Figure 7b, after calling *extend_allocated*, the REE file system can directly load encrypted parameters into the *unprotected* allocated memory, instead of a bounce buffer. After loading, the new memory is protected with *extend_protected* and the parameters are decrypted. This design reduces memory consumption and avoids additional copying overhead.

Minimizing TEE OS modification. The "extend and shrink" interfaces introduce only minor modifications to the TEE OS. In contrast, if the LLM TA is allowed to allocate/deallocate secure memory in an arbitrary order, the TZASC region will become fragmented and the TEE OS needs to defragment the region upon revocation. TZ-LLM leverages the allocation patterns to avoid this complexity.

4.3 TEE-REE NPU Time-Sharing

Inspired by the outsource-and-verify principle [98], which delegates complex operations to an untrusted component while verifying their outcomes, TZ-LLM adopts a co-driver design to enable NPU time-sharing between the REE and the

TEE, as shown in Figure 8. Specifically, a full-fledged NPU driver and a small data plane NPU driver are deployed in the REE and the TEE, respectively, cooperating to manage secure and non-secure NPU jobs. The TEE data plane driver outsources control plane operations to the untrusted REE driver and verifies the returned results.

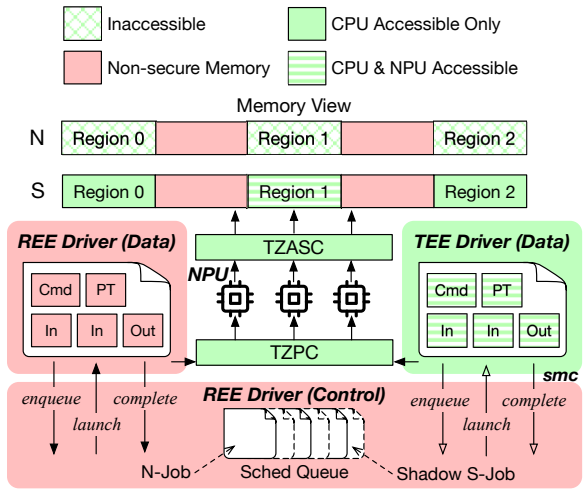

Figure 8. TEE-REE NPU time-sharing, S: secure, N: non-secure, cmd: register commands, PT: I/O page table, in/out: input/output buffers, region: TZASC region.

The goals of the co-driver design are to: (1) separate the NPU driver into isolated domains, (2) provide an isolated execution environment that ensures the confidentiality and integrity of secure NPU jobs, (3) enable NPU time sharing for secure and non-secure NPU jobs with minimal overhead, and (4) minimize the additional TCB introduced to the TEE.

Separating control and data planes. The data plane of the NPU driver performs the following steps for each NPU job: (1) it initializes the execution context of the job, i.e., the *memory* for the I/O page table, register commands (the NPU job code), and input/output buffers; (2) it performs *MMIO* operations to launch the job by specifying the execution context; (3) it handles *interrupts* upon job completion. These steps form a minimal closure that should be integrated into the TEE driver, with the corresponding resources (*memory*, *MMIO*, and *interrupts*) isolated to preserve the confidentiality and integrity of secure NPU jobs.

The control plane of the NPU driver manages device configuration during initialization and power management before and after job execution. As shown in Figure 8, it also handles job scheduling, which interacts with the data plane through scheduling interfaces: (1) it *enqueues* the job into the scheduling queue; (2) it calls the data plane for *launching* when the job is scheduled; (3) it continues to schedule the next job upon *completion* of the current job. Since the control plane does not access the isolated resources during job execution, it can safely reside in the REE driver. The function call interfaces between the REE control plane and the TEE data plane are replaced with *smc*.

Isolated execution environment. The TEE driver switches the NPU between non-secure and secure modes. In non-secure mode, the NPU is prohibited from accessing secure memory. In secure mode, the NPU's *MMIO* region is accessible only to the TEE, its *interrupts* are routed to the TEE, and it is allowed to access secure *memory*. Secure jobs run in secure mode, and the execution contexts of secure jobs are stored in secure memory.

Specifically, the TEE driver performs the following steps when switching the NPU to secure mode. First, it updates the TZPC to isolate the MMIO region of the NPU from the REE and the GIC controller to route NPU interrupts to the TEE. Second, it waits for the ongoing non-secure NPU job, if any, to complete. Third, it sets the TZASC to grant the NPU access to secure memory. The order of these steps is critical to ensure that (1) no new non-secure NPU job can be launched during the sanity check of ongoing non-secure jobs, and (2) any previously launched non-secure NPU job is completed before the NPU is granted access to secure memory.

TEE-REE time-sharing. TZ-LLM reuses the NPU job scheduling mechanism in the REE driver to support NPU time-sharing for secure and non-secure NPU jobs.

As shown in Figure 8, the REE driver is extended to maintain a unified scheduling queue for secure and non-secure NPU jobs. Each time the LLM TA issues a secure NPU job, the TEE driver issues a paired shadow job with an empty execution context to the REE driver. When a shadow job is scheduled, the REE driver proactively transfers NPU control to the TEE driver. The TEE driver then transitions the NPU into secure mode to create an isolated execution environment. To prevent arbitrary launch and replay attacks, the TEE driver ensures that the secure NPU job has been previously initialized but not yet issued. To prevent reordering attacks, the TEE driver assigns each job a monotonic sequence number before issuing it to the REE driver and verifies the number against the current execution sequence number when scheduled. After these checks, the TEE driver launches the secure NPU job and waits for its completion. Upon completion of the secure job (receipt of a secure interrupt), the TEE driver returns the NPU back to non-secure mode and informs the REE driver that the shadow job is complete. Finally, the REE driver discards the shadow job and schedules the next NPU job.

Minimal TCB. Despite the complexity of the REE NPU driver, TZ-LLM minimizes the additional TCB in TEE with two complementary approaches. First, TZ-LLM integrates only the tiny data plane closure into the TEE driver, while excluding control plane components such as job scheduling and dependencies on complex Linux subsystems like device, memory, interrupt, and power management.

Second, TZ-LLM deprivileges the TEE NPU driver to user mode, isolating potential vulnerabilities in the driver from affecting the existing TEE system. The TEE OS strictly confines the privileges of the user-mode NPU driver by enforcing two restrictions. First, the TEE OS only maps the MMIO region of the NPU into the NPU driver's address space, and thus the driver cannot access other secure devices. Second, the TEE OS only allows the NPU to access the execution contexts of secure NPU jobs. This is possible because the parameters, intermediate results, I/O page tables and register commands are placed in independent TZASC regions (§4.2). By configuring the TZASC, the TEE OS only allows the NPU to access these specific regions, while prohibiting NPU access to all other regions. This design follows the broader minimal TCB TEE philosophy [32, 60], retaining only a minimal privileged security monitor for isolation while deprivileging functionality into user mode.

5 Implementation

We prototype TZ-LLM on OpenHarmony OS [16] and its TEE system, which is an open-sourced version of Huawei's commercial HarmonyOS [8]. The LLM TA is built based on llama.cpp [13], a popular on-device inference framework. The REE OS is OpenHarmony v4.1 with Linux v5.10. The NPU driver is Rockchip NPU driver v0.9.8 [22].

The original TEE OS contains 17K LoC for basic functionalities, including thread management, IPC, interrupt dispatching, and memory management. We only extend it with 62 LoC to manage CMA page memory mapping and 50 LoC to support dynamic configuration of TZASC and TZPC. The llama.cpp inference framework is extended with 1.2K LoC for pipelined restoration, 1K LoC for integrating the data plane of the NPU drive, and the OpenSSL library [24] for parameter decryption. Note that the computation graph is directly extracted via internal interfaces of llama.cpp. In the REE OS, we add only 364 LoC to the Linux kernel, which consists of 167 LoC in the NPU driver for shadow job scheduling and 197 LoC in the TZ driver for CMA allocation and deallocation.

The current implementation works on a Rockchip platform, while TZ-LLM design is applicable to other Arm platforms, such as Qualcomm. We investigate the open-source Linux driver for Qualcomm NPUs [18] and confirm that we can also extract a small data plane driver from it.

6 Security Analysis

TZ-LLM protects the confidentiality of LLM parameters from any attacker who compromise the REE OS, the REE applications, or other TAs in the TEE.

Preventing direct access attacks. If an attacker in the REE tries to access plaintext parameters in the secure memory, the TZASC hardware blocks such access attempts. A malicious TA also cannot access the parameters in secure memory as the TEE OS enforces address space isolation between TAs.

If the attacker attempts to read the parameters in flash, he/she will only get content encrypted with a model key. The model key in flash is encrypted with a hardware-protected TEE key. It can only be decrypted by the TEE OS. The TEE OS only allows the LLM TA to access the model key.

Preventing DMA attacks. The attacker may exploit the NPU or other untrusted devices to initiate malicious DMA requests targeting parameters in the secure memory.

For the NPU, the TEE driver enforces two key protections before granting access to secure memory. First, it configures the TZPC to prohibit REE access to MMIO interface of the NPU. Second, it ensures that no NPU job previously launched by the REE driver is still executing. Therefore, the DMA destination can only be a benign address set by the TEE driver, preventing parameter leakage.

For untrusted devices, whether secure or non-secure, the TZASC is configured to reject any access from them to the secure memory regions for LLM parameters.

Preventing Iago attacks. Attackers may attempt to compromise the LLM TA or TEE OS for model theft by exploiting the interface between the TEE and the REE for Iago attacks. TZ-LLM exposes four TEE-REE interfaces vulnerable to Iago attacks: secure memory scaling, NPU job scheduling, model loading, and CPU thread scheduling.

For secure memory scaling, the CMA may return arbitrary memory addresses to the TEE. TZ-LLM counters this by validating the contiguity of the returned address against the previously allocated memory (§4.2). For NPU job scheduling, the REE NPU driver may schedule unauthorized secure jobs, replay previously scheduled jobs, or reorder them. TZ-LLM counters this by validating the job before execution (§4.3). For model loading (§3.2), a malicious REE OS may return forged results. TZ-LLM counters this by verifying the returned content using checksums. For CPU thread scheduling, the REE scheduler may violate the required execution order of TA threads. TZ-LLM counters this by managing synchronization primitives in the TEE (§3.2), ensuring that TA thread follows the execution order enforced by these primitives.

Side-channel and physical attack considerations. Existing side-channel attacks on TrustZone [45, 65, 94] are outside the scope of this paper and have known mitigations [34, 59]. TZ-LLM may introduce two other side channels. First, the parameter tensor sizes are exposed to the REE when the TA scales secure memory. Second, the execution time of secure NPU jobs is exposed to the REE driver when it schedules the jobs. These channels may reveal model structures, but not parameter values. To the best of our knowledge, there are no public reports of side-channel attacks successfully stealing on-device LLM parameters. Additionally, these channels could be mitigated through orthogonal techniques such as dummy parameter loading and dummy computation.

Physical attacks through offline DRAM analysis, such as cold-boot attacks [92], stem from TrustZone's hardware limitations. They can be mitigated by future memory encryption hardware [10] and are orthogonal to TZ-LLM.

Ensuring the security of the existing TEE system. TZ-LLM minimizes its security impact on the existing TAs and TEE OS. First, TZ-LLM minimizes the modification of the privileged TEE OS to about 100 LoC (§5), by running the TEE NPU driver in user space, and designing simple secure memory scaling interfaces (§4.2). Second, even if the LLM TA is compromised, it cannot access the memory of other TAs or the TEE OS with direct read/write or NPU DMA, because the TEE OS enforces address space isolation, and the TZASC configuration only allows the NPU to access the memory regions for NPU job execution contexts (§4.3).

7 Performance Evaluation

In this section, we first evaluate the end-to-end performance of the LLM TA (§7.1) and analyze the optimization effect of pipelined restoration (§7.2). Then, we run the LLM TA in parallel with REE applications to evaluate the overhead of TEE-REE NPU time-sharing (§7.3) and the interference caused by CMA allocation (§7.4).

Testbed. The evaluation is conducted on an Orange Pi 5 Plus board [17] (RK3588 CPU/NPU [21]), equipped with four Cortex-A76 (2.4GHz) CPU cores and four Cortex-A55 (1.8GHz) CPU cores, 16GB of LPDDR4X RAM, an 1TB NVMe SSD (PCIe 3.0 x4), and an NPU with three cores and up to 6 TOPS of computation power.

Baselines. We compare TZ-LLM with the following baselines: (1) *REE-LLM-Memory*: The unmodified llama.cpp framework running in the REE, with all model parameters preloaded in memory. This baseline represents the theoretical best performance, but it is impractical due to memory inefficiency and lacks protection for parameters. (2) *REE-LLM-Flash*: The unmodified llama.cpp framework running in the REE, with model parameters loaded with pipelined restoration at inference start (buddy system allocation, no decryption). This baseline is practical but provides no protection for parameters. (3) *Strawman*: The *"cold start"* strawman in §2.3, which performs cold start and CPU computation for each inference request. This baseline offers security guarantees and memory efficiency but lacks pipelined restoration and NPU support within the TEE.

Models and deployment. We evaluate TZ-LLM with four representative on-device LLMs: TinyLlama-1.1B[95], Qwen2.5-3B[89], Phi-3-3.8B[28], and Llama-3-8B[12]. All models are 8-bit quantized, with parameter sizes of 1.0 GB, 3.3 GB, 3.7 GB, and 7.9 GB, respectively.

The LLM TA runs on the four Cortex-A76 CPU cores and all three NPU cores. For evaluations in §7.1 and §7.2, we simulate the memory pressure in the REE with the stress-ng [23] tool, to trigger memory migration during CMA allocation. To show the worst-case performance, the memory pressure is 13GB, 11GB, 10GB and 6GB for the four models, respectively. The stressing threads and LLM threads are pinned to different CPU cores to avoid interference.

Benchmarks. We use three benchmarks from prior work on on-device LLMs [85, 88]: UltraChat [36] (multi-turn dialogues), PersonaChat [52] (chat summarization tasks), and DroidTask [83] (UI automation tasks).

7.1 End-to-End Performance

In this section, we evaluate the prefill and decoding performance of the LLM TA and explain the source of overhead or optimization compared with the baselines.

7.1.1 Prefill Performance.

We evaluate TZ-LLM's end-to-end prefill performance using both fixed-length prompts and real-world benchmarks.

TTFT under different prompt lengths. Figure 9 presents the TTFT of the evaluated systems and models at prompt lengths of 32, 128 and 512 tokens.

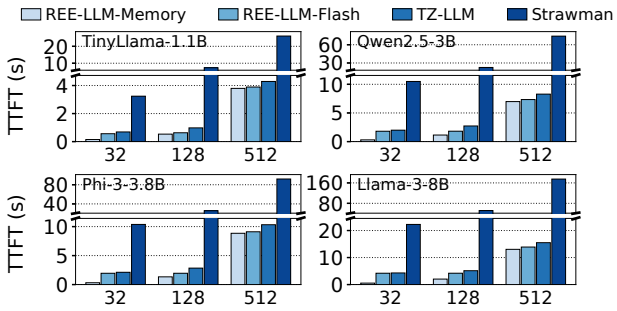

Figure 9. TTFT of different models under different prompt lengths. The x-axis represents the prompt length.

Compared to the strawman baseline, TZ-LLM reduces the TTFT by 77.1%~91.1% across all models and prompt lengths. This improvement stems from the pipelined parameter restoration mechanism, the NPU support in the TEE, and the checkpoint/restoration of the framework initial state (mentioned in §3). The NPU support accelerates prefill computation, reducing TTFT by up to 87.2%. Meanwhile, state checkpoint eliminates the framework initialization overhead, further reducing TTFT by up to 36.8%. Finally, the pipeline mechanism effectively hides the parameter restoration latency, further reducing TTFT by up to 40.6%.

Compared to the REE-LLM-Flash baseline, TZ-LLM incurs 2.5%~22.3%, 22.2%~55.3%, 10.2%~15.2% overhead at prompt lengths of 32, 128, and 512, respectively. This overhead is mainly caused by CMA allocation (memory migration) and the parameter decryption during parameter restoration. The overhead is relatively small for short and long prompt lengths, but more pronounced for medium prompt lengths. For short prompt lengths, the TTFT is bounded by flash

I/O, allowing allocation and decryption to overlap with I/O operators. For long prompt lengths, the TTFT is dominated by CPU and NPU computation, enabling allocation and decryption to overlap with NPU execution. In contrast, for medium prompt lengths, the TTFT is driven by the sum of CPU computation, allocation, and decryption, resulting in only partial overlap of allocation and decryption with NPU execution and I/O operators. Other overheads of TZ-LLM, such as the communication between TEE/REE NPU drivers and decryption of the framework initial state, are minor compared with prefill computation time.

TZ-LLM incurs up to 8.5× overhead compared with the REE-LLM-Memory baseline, due to parameter restoration. However, TZ-LLM is more memory-efficient, and the overhead can be reduced with partial parameter caching (§7.2.3). Moreover, the overhead is only 13.0%~18.9% for the long prompts (512 tokens), as parameter restoration overhead is hidden under the computation time.

Benchmark results. Figure 10 shows the average TTFT of the evaluated systems and models on the three benchmarks.

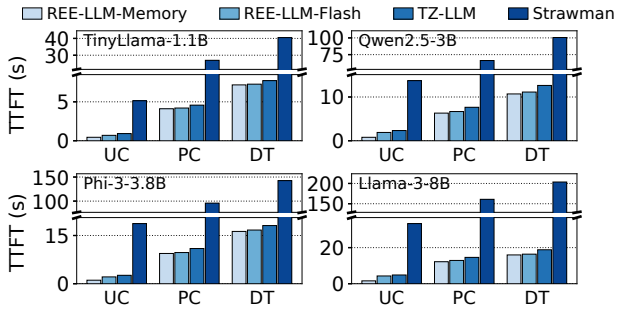

Figure 10. Average TTFT on different real-world benchmarks, UC: UltraChat, PC: PersonaChat, DT: DroidTask.

For each pair of model and benchmark, we calculate the geometric mean of TZ-LLM's overhead/optimization across different prompts. TZ-LLM achieves 76.1%~90.9% TTFT reduction compared to the strawman baseline, while incurring 5.2%~28.3% slowdown compared to the REE-LLM-Flash baseline. Compared to the REE-LLM-Memory baseline, TZ-LLM incurs 2.5×~3.7× overhead on UltraChat and 8.1%~21.2% overhead on PersonaChat and DroidTask. The higher overhead on UltraChat is due to its shorter prompts, where parameter restoration dominates the inference time, but it can be mitigated via partial parameter caching (§7.2.3).

7.1.2 Decoding Performance. Figure 11 shows the decoding speed at a prompt length of 128 and an output length of 64. Results under other prompt and output lengths are similar and are omitted for brevity. The decoding speeds of REE-LLM-Memory and REE-LLM-Flash are the same, so we only show a single bar.

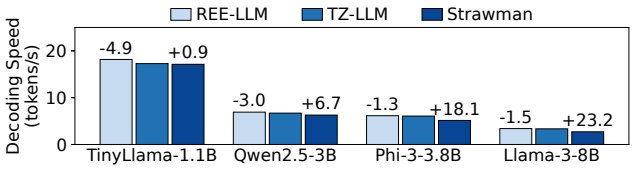

Figure 11. Token generation speeds during decoding for different models. The percentages shown above each bar represent TZ-LLM's relative performance improvement (+) or degradation (-) compared to the respective baseline.

The decoding speed of TZ-LLM shows a modest 0.9%~23.2% improvement over the strawman baseline, thanks to the NPU support in the TEE. In contrast to the more significant gains seen in the prefill stage, this relatively small improvement can be attributed to the single-batch computation pattern of decoding (processing one token in each iteration), which cannot fully utilize the computation power of the NPU. Compared to the REE-LLM baseline, TZ-LLM experiences a 1.3%~4.9% slowdown in decoding speed. This overhead originates from the communication between the TEE and REE NPU drivers for NPU multiplexing (§4.3). The overhead is smaller for larger models because the NPU computation time is longer.

7.2 Effect of Pipelined Restoration

In this section, we comprehensively evaluate the pipelined restoration mechanism in TZ-LLM. First, we evaluate the effectiveness of our pipeline scheduling policy (§7.2.1). Then, we analyze how preemptive scheduling (§7.2.2) and partial parameter caching (§7.2.3) reduce the TTFT.

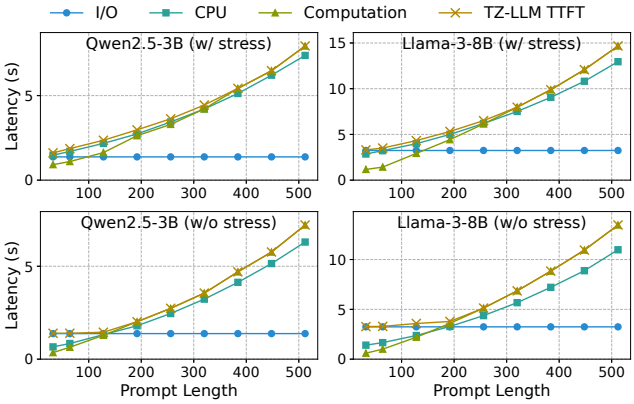

Figure 12. The latency of each critical path and the TTFT of TZ-LLM under different models and prompt lengths, with 20% LLM parameters cached. stress: memory stress. I/O: the total latency of all loading (I/O) operators. CPU: the total latency of CPU computation, allocation, and decryption. Computation: the total latency of CPU and NPU computation.

7.2.1 Scheduling Policy Effectiveness. To evaluate the effectiveness of our priority-based pipeline scheduling policy, we analyze the latency of the three potential critical paths of the pipeline mentioned in §4.1. The maximum latency of them is the theoretical lower bound on TTFT for any scheduling policy. We configure the experiments with 20% of the parameters cached to eliminate initial pipeline bubbles, which is independent of the scheduling policy. Since our scheduling policy favors the scenario with the critical path of CPU or computation operators, we also evaluate the scenario with the critical path of I/O operators by eliminating memory stress (eliminating CPU memory migration overhead) to analyze the worst case of our policy.

Figure 12 shows that TZ-LLM incurs 0.01%~9.9% overhead compared to the theoretical lower bound when memory stress is enabled. When disabling memory stress, the overhead increases to a modest 10.4%. Therefore, our scheduling policy performs close to the optimal one.

7.2.2 Effect of Preemptive Scheduling. Figure 13 shows the effect of preemptive pipeline scheduling on reducing the TTFT. Compared with TZ-LLM without pipelined restoration, the pipeline without preemption reduces the TTFT by up to 31.7%. By enabling preemption on allocation and decryption operators, TZ-LLM eliminates the pipeline bubbles caused by misalignment of operator execution times, further reducing the TTFT by up to 16.2%.

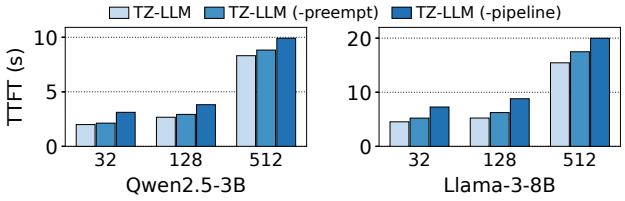

Figure 13. The effect of preemptive pipeline scheduling under different prompt lengths and models.

7.2.3 Effect of Partial Parameter Caching. To assess the effect of partial parameter caching, we vary the proportion of cached parameters from 0% to 100%. Figure 14 illustrates the TTFT of various models across different prompt lengths and cache proportions.

As more parameters are cached, the TTFT decreases approximately linearly up to a threshold. After this threshold, the benefit of additional caching diminishes as the restoration overhead is effectively hidden beneath the computation. This threshold is primarily determined by the NPU computation time, which depends on the model and prompt length. Besides the current mechanism that adjusts the cache size based on REE memory pressure, TZ-LLM can also determine a cache size by identifying the threshold with profiling.

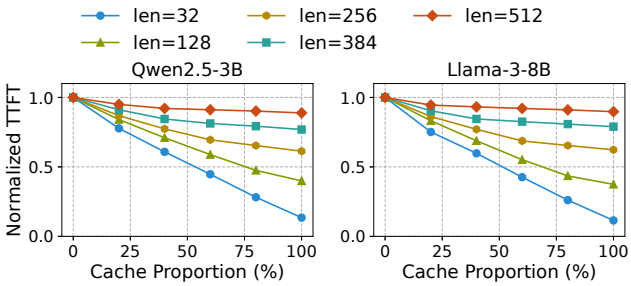

Figure 14. The TTFT of TZ-LLM under different cache proportions. For each model and prompt length, the TTFT is normalized by the TTFT of the 0% cache setup.

7.3 NPU Time-Sharing Performance

We evaluate the NPU time-sharing performance of TZ-LLM by concurrently running mainstream neural network (NN) applications that use the NPU alongside LLM inference. The two evaluated NN applications are YOLOv5 [27] for object detection and MobileNet [49] for image classification. We choose two LLM models with small or large model sizes and use a prompt with 512 tokens. The throughputs of the NN applications and the LLMs are displayed in Figure 15.

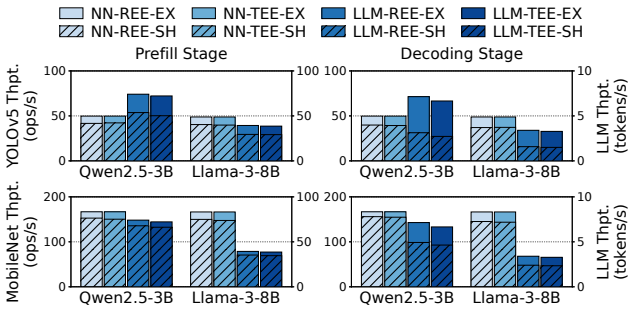

Figure 15. The throughputs of NN applications (left y-axis) and LLMs (right y-axis) with NPU time-sharing. REE: REE-LLM-Memory, TEE: TZ-LLM (100% cached), EX: NN application and LLM run exclusively, SH: NN application and LLM run concurrently with a shared NPU.

As expected, when the NN application and the LLM run concurrently (-SH), the throughputs of both sides are lower compared with their counterparts under exclusive running (-EX), due to NPU multiplexing. Compared with NPU time-sharing within the REE (-REE), the TEE-REE NPU time-sharing mechanism (-TEE) introduces a small additional overhead, with NN applications and LLMs experiencing only up to 3.8% and 3.0% extra slowdown, respectively.

To quantify the overhead of TEE-REE NPU time-sharing, we measure the time spent on (1) *smc* switches for shadow job scheduling, (2) TZASC and TZPC configuration, and (3) GIC configuration. The total time-sharing overhead accounts

for 1.6%∼2.7% and 2.3%∼5.7% of the TTFT and decoding time across all evaluated setups.

7.4 Interference Between CMA and REE Applications

The performance of REE applications may be affected by memory migration during CMA allocation. We evaluate this overhead by concurrently running Geekbench [7] with LLM inference. To show the worst-case overhead, we configure TZ-LLM and REE-LLM-Flash to run the prefill stage, revoke all memory, and then restart the prefill stage. The REE-LLM-Memory baseline only repeats the prefill stage. The benchmark threads and LLM inference threads are pinned to different CPU cores. The results are shown in Figure 16.

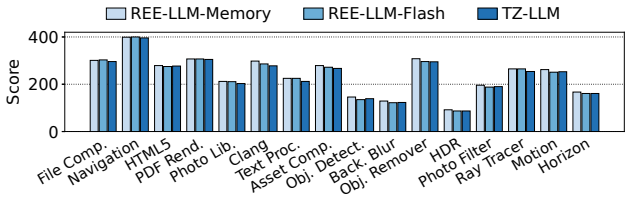

Figure 16. Geekbench scores when running concurrently with LLM prefill stage (Llama-3-8B, 512-token prompt).

Compared to the REE-LLM-Memory and REE-LLM-Flash baselines, the Geekbench scores under TZ-LLM show a degradation of up to 6.7% and 5.8%, respectively. This overhead is comparable to that introduced by S2PT (Figure 2). However, unlike S2PT, the overhead from CMA allocation occurs *only during the prefill stage* and is negligible during the decoding stage or when inference is not running.

8 Other Related Work

On-device LLM inference. On-device LLM inference has garnered significant attention recently [86, 87]. Some prior work runs the LLM with limited memory by swapping parameters between the memory and the flash during inference [29, 88]. Combining TZ-LLM with these parameter offloading techniques is considered future work. Currently, TZ-LLM keeps parameters in memory during decoding. Some prior systems [85, 88] use on-device NPUs to speed up LLMs. TZ-LLM supports NPU acceleration in the TEE.

Some work reduces the parameter sizes of LLMs with quantization, knowledge distillation, or weight pruning [11, 35, 39, 40, 43, 44, 64, 67, 68, 91]. However, even with reduced parameter sizes, dynamic secure memory scaling is still needed for memory efficiency.

Model confidentiality protection. Some prior work protects models with cryptographic techniques like homomorphic encryption [58, 70] and multi-party computation [37, 82]. However, they incur significant performance degradation, particularly on resource-constrained mobile devices. Some systems [71, 78] run LLM inference in confidential virtual machines (CVMs) [3, 9, 10, 14], but CVM hardware is not currently available on mobile devices.

Outsource-and-verify principle. Prior work [98] applies the outsource-and-verify principle to the USB driver, delegating complex USB bus functions to the untrusted OS while verifying their results. TZ-LLM adopts this design principle and addresses the specific challenges of the NPU driver, namely partitioning it into isolated domains and enabling efficient interaction between them. On the one hand, by examining the workflow of the NPU driver, TZ-LLM partitions the driver into two isolated domains: the control plane and the data plane. On the other hand, TZ-LLM develops efficient methods for the trusted data plane to verify the outcomes of the untrusted control plane.

TEEs with accelerators. Some prior work extends TEE protection boundary to accelerators in the cloud [14, 30, 50, 53, 56, 69, 74, 79, 80, 84, 99] or on end devices [33, 38, 66]. StrongBox [33] utilizes the EL3 monitor to protect secure GPU jobs, which extends the highly privileged TCB. TZ-LLM supports NPU in TEE with minimal security impact. sNPU [38] supports fine-grained and dynamic TEE-REE NPU space-sharing, but it requires hardware modifications.

9 Conclusion

TZ-LLM is a novel system designed for protecting on-device LLMs with Arm TrustZone. It satisfies LLM performance, memory efficiency, and security requirements using elastic secure memory scaling with pipelined restoration and TEE-REE NPU time-sharing with control-data separation.

Acknowledgments

We sincerely thank our shepherd Hyungon Moon and the anonymous reviewers, whose reviews, feedback, and suggestions have significantly strengthened our work. This research was supported in part by National Natural Science Foundation of China (No. 62432010), CCF-Huawei Populus Grove Fund, and the Fundamental Research Funds for the Central Universities. Corresponding author: Jinyu Gu (gujinyu@sjtu.edu.cn).

References

[1] A deep dive into CMA. https://lwn.net/Articles/486301/.
[2] About the TZC-400. https://developer.arm.com/documentation/ddi0504/c/introduction/about-the-tzc-400.
[3] AMD Secure Encrypted Virtualization (SEV). https://www.amd.com/en/developer/sev.html.
[4] Apple Intelligence. https://www.apple.com/apple-intelligence/.
[5] Chatbot Arena. https://lmarena.ai.
[6] Galaxy AI. https://www.samsung.com/us/galaxy-ai/.
[7] Geekbench. https://www.geekbench.com.
[8] HarmonyOS. https://www.harmonyos.com/en/.
[9] Intel Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html.

[10] Introducing Arm Confidential Compute Architecture. https://developer.arm.com/documentation/den0125/400/Overview.

[11] K-Quants. https://github.com/ggml-org/llama.cpp/pull/1684.

[12] Llama 3. https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

[13] LLM inference in C/C++. https://github.com/ggerganov/llama.cpp.

[14] NVIDIA Confidential Computing. https://images.nvidia.cn/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf.

[15] OP-TEE Documentation. https://optee.readthedocs.io/en/latest/index.html.

[16] OpenHarmony. https://gitee.com/openharmony.

[17] Orange Pi 5 Plus (32GB). http://www.orangepi.org/html/hardWare/computerAndMicrocontrollers/details/Orange-Pi-5-plus-32GB.html.

[18] QTI MSM NPU driver. https://android.googlesource.com/kernel/msm/+/60319ac47b3d30c81413fb0ebb9a21085a9a0be0/drivers/media/platform/msm/npu/.

[19] Qualcomm AI Hub. https://aihub.qualcomm.com/mobile/models.

[20] Qualcomm Hexagon NPU. https://www.qualcomm.com/products/technology/processors/hexagon.

[21] RK3588. https://www.rock-chips.com/a/en/products/RK35_Series/2022/0926/1660.html.

[22] rknpu-driver. https://github.com/airockchip/rknn-llm/tree/main/rknpu-driver.

[23] stress-ng (stress next generation). https://github.com/ColinIanKing/stress-ng.

[24] TLS/SSL and crypto library. https://github.com/openssl/openssl.

[25] TrustZone for Cortex-A. https://www.arm.com/technologies/trustzone-for-cortex-a.

[26] Voice Assistant Celia - HUAWEI Global. https://consumer.huawei.com/en/emui/celia/.

[27] YOLOv5: A state-of-the-art real-time object detection system. https://docs.ultralytics.com.

[28] Marah I Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat S. Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Caio César Teodoro Mendes, Weizhu Chen, Vishrav Chaudhary, Parul Chopra, Allie Del Giorno, Gustavo de Rosa, Matthew Dixon, Ronen Eldan, Dan Iter, Amit Garg, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Jamie Huynh, Mojan Javaheripi, Xin Jin, Piero Kauffmann, Nikos Karampatziakis, Dongwoo Kim, Mahoud Khademi, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Chen Liang, Weishung Liu, Eric Lin, Zeqi Lin, Piyush Madan, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Xia Song, Masahiro Tanaka, Xin Wang, Rachel Ward, Guanhua Wang, Philipp Witte, Michael Wyatt, Can Xu, Jiahang Xu, Sonali Yadav, Fan Yang, Ziyi Yang, Donghan Yu, Chengruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. Phi-3 technical report: A highly capable language model locally on your phone. *CoRR*, abs/2404.14219, 2024.

[29] Keivan Alizadeh, Seyed-Iman Mirzadeh, Dmitry Belenko, S. Khatamifard, Minsik Cho, Carlo C. del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. LLM in a flash: Efficient large language model inference with limited memory. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 12562–12584. Association for Computational Linguistics, 2024.

[30] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A security architecture with customizable and resilient enclaves. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1073–1090. USENIX Association, 2021.

[31] Shai Bergman, Mark Silberstein, Takahiro Shinagawa, Peter R. Pietzuch, and Lluís Vilanova. Translation pass-through for near-native paging performance in vms. In Julia Lawall and Dan Williams, editors, *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, pages 753–768. USENIX Association, 2023.

[32] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, 2016.

[33] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, and Fengwei Zhang. Strongbox: A GPU TEE on arm endpoints. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 769–783. ACM, 2022.

[34] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. Hybcache: Hybrid side-channel-resilient caches for trusted execution environments. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 451–468. USENIX Association, 2020.

[35] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3.int8(): 8-bit matrix multiplication for transformers at scale. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[36] Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. Enhancing chat language models by scaling high-quality instructional conversations, 2023.

[37] Ye Dong, Wen jie Lu, Yancheng Zheng, Haoqi Wu, Derun Zhao, Jin Tan, Zhicong Huang, Cheng Hong, Tao Wei, and Wenguang Chen. Puma: Secure inference of llama-7b in five minutes, 2023.

[38] Erhu Feng, Dahu Feng, Dong Du, Yubin Xia, and Haibo Chen. snpu: Trusted execution environments on integrated npus. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*, pages 708–723. IEEE, 2024.

[39] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 10323–10337. PMLR, 2023.

[40] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: accurate post-training quantization for generative pre-trained transformers. *CoRR*, abs/2210.17323, 2022.

[41] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. Agile paging: Exceeding the best of nested and shadow paging. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 707–718. IEEE Computer Society, 2016.

[42] Karan Grover, Shruti Tople, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure dnn inference with enclaves. *arXiv preprint arXiv:1810.00602*, 2018.

[43] Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. Minillm: Knowledge distillation of large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria,*

May 7-11, 2024. OpenReview.net, 2024.

[44] Hui Guan, Shaoshan Liu, Xiaolong Ma, Wei Niu, Bin Ren, Xipeng Shen, Yanzhi Wang, and Pu Zhao. Cocopie: enabling real-time AI on off-the-shelf mobile devices via compression-compilation co-design. *Commun. ACM*, 64(6):62–68, 2021.

[45] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 38–55. IEEE Computer Society, 2016.

[46] Seung-Kyun Han and Jinsoo Jang. Mytee: Own the trusted execution environment on embedded devices. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[47] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Maximilian Augustin, Michael Backes, and Mario Fritz. Mlcapsule: Guarded offline deployment of machine learning as a service. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2021, virtual, June 19-25, 2021*, pages 3300–3309. Computer Vision Foundation / IEEE, 2021.

[48] Alexander Van't Hof and Jason Nieh. Blackbox: A container security monitor for protecting containers on untrusted operating systems. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 683–700. USENIX Association, 2022.

[49] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[50] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. Telekine: Secure computing with cloud gpus. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 817–833. USENIX Association, 2020.

[51] Md Shihabul Islam, Mahmoud Zamani, Chung Hwan Kim, Latifur Khan, and Kevin W. Hamlen. Confidential execution of deep learning inference at the untrusted edge with ARM trustzone. In Mohamed Shehab, Maribel Fernández, and Ninghui Li, editors, *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy, CODASPY 2023, Charlotte, NC, USA, April 24-26, 2023*, pages 153–164. ACM, 2023.

[52] Pegah Jandaghi, XiangHai Sheng, Xinyi Bai, Jay Pujara, and Hakim Sidahmed. Faithful persona-based conversational dataset generation with large language models, 2023.

[53] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity gpus. 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 455–468. ACM, 2019.

[54] Jinkyu Jeong, Hwanju Kim, Jeaho Hwang, Joonwon Lee, and Seungryoul Maeng. Daac: device-reserved memory as an eviction-based file cache. In Ahmed Jerraya, Luca P. Carloni, Vincent John Mooney III, and Rodric M. Rabbah, editors, *Proceedings of the 15th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*, pages 191–200. ACM, 2012.

[55] Zhaolong Jian, Xu Liu, Qiankun Dong, Longkai Cheng, Xueshuo Xie, and Tao Li. Smartzone: Runtime support for secure and efficient on-device inference on arm trustzone. *IEEE Transactions on Computers*, 2025.

[56] Jianyu Jiang, Ji Qi, Tianxiang Shen, Xusheng Chen, Shixiong Zhao, Sen Wang, Li Chen, Gong Zhang, Xiapu Luo, and Heming Cui. CRONUS: fault-isolated, secure and high-performance heterogeneous computing for trusted execution environment. In *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*, pages 124–143. IEEE, 2022.

[57] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In Simon L. Peyton Jones and Richard E. Jones, editors, *International Symposium on Memory Management, ISMM '98, Vancouver, British Columbia, Canada, 17-19 October, 1998, Conference Proceedings*, pages 26–36. ACM, 1998.

[58] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha P. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1651–1669. USENIX Association, 2018.

[59] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, pages 974–987. IEEE Computer Society, 2018.

[60] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[61] 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 Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, and Vassilis Kostakos, editors, *The 25th Annual International Conference on Mobile Computing and Networking, MobiCom 2019, Los Cabos, Mexico, October 21-25, 2019*, pages 46:1–46:17. ACM, 2019.

[62] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. Twinvisor: Hardware-isolated confidential virtual machines for ARM. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 638–654. ACM, 2021.

[63] Qinfeng Li, Zhiqiang Shen, Zhenghan Qin, Yangfan Xie, Xuhong Zhang, Tianyu Du, Sheng Cheng, Xun Wang, and Jianwei Yin. Translinkguard: safeguarding transformer models against model stealing in edge deployment. In *Proceedings of the 32nd ACM International Conference on Multimedia*, pages 3479–3488, 2024.

[64] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: activation-aware weight quantization for on-device LLM compression and acceleration. In Phillip B. Gibbons, Gennady Pekhimenko, and Christopher De Sa, editors, *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*. mlsys.org, 2024.

[65] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 549–564. USENIX Association, 2016.

[66] Renju Liu, Luis Garcia, Zaoxing Liu, Botong Ou, and Mani B. Srivastava. Secdeep: Secure and performant on-device deep learning inference framework for mobile and iot devices. In *IoTDI '21: International Conference on Internet-of-Things Design and Implementation, Virtual Event / Charlottesville, VA, USA, May 18-21, 2021*, pages 67–79. ACM, 2021.

[67] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The era of 1-bit llms: All large language models are in 1.58 bits. *CoRR*, abs/2402.17764, 2024.

[68] Xinyin Ma, Gongfan Fang, and Xinchao Wang. Llm-pruner: On the structural pruning of large language models. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[69] Haohui Mai, Jiacheng Zhao, Hongren Zheng, Yiyang Zhao, Zibin Liu, Mingyu Gao, Cong Wang, Huimin Cui, Xiaobing Feng, and Christos Kozyrakis. Honeycomb: Secure and efficient GPU executions via static validation. In Roxana Geambasu and Ed Nightingale, editors, *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 155–172. USENIX Association, 2023.

[70] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2505–2522. USENIX Association, August 2020.

[71] Apoorve Mohan, Mengmei Ye, Hubertus Franke, Mudhakar Srivatsa, Zhuoran Liu, and Nelson Mimura Gonzalez. Securing ai inference in the cloud: Is cpu-gpu confidential computing ready? In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pages 164–175, 2024.

[72] Seongjae Park, Minchan Kim, and Heon Y. Yeom. GCMA: guaranteed contiguous memory allocator. *IEEE Trans. Computers*, 68(3):390–401, 2019.

[73] Tianxiang Shen, Ji Qi, Jianyu Jiang, Xian Wang, Siyuan Wen, Xusheng Chen, Shixiong Zhao, Sen Wang, Li Chen, Xiapu Luo, Fengwei Zhang, and Heming Cui. SOTER: guarding black-box inference for general neural networks at the edge. In Jiri Schindler and Noa Zilberman, editors, *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 723–738. USENIX Association, 2022.

[74] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, Mark Kuhne, Fabio Aliberti, and Shweta Shinde. ACAI: extending arm confidential computing architecture protection from cpus to accelerators. *CoRR*, abs/2305.15986, 2023.

[75] Yu Sun, Gaojian Xiong, Jianhua Liu, Zheng Liu, and Jian Cui. Tsqp: Safeguarding real-time inference for quantization neural networks on edge devices. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 2114–2132. IEEE, 2025.

[76] Zhichuang Sun, Ruimin Sun, Changming Liu, Amrita Roy Chowdhury, Long Lu, and Somesh Jha. Shadownet: A secure and efficient on-device model inference system for convolutional neural networks. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1596–1612. IEEE, 2023.

[77] Zhichuang Sun, Ruimin Sun, Long Lu, and Alan Mislove. Mind your weight(s): A large-scale study on insufficient machine learning model protection in mobile apps. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1955–1972. USENIX Association, 2021.

[78] Yifan Tan, Cheng Tan, Zeyu Mi, and Haibo Chen. Pipellm: Fast and confidential large language model services with speculative pipelined encryption. *arXiv preprint arXiv:2411.03357*, 2024.

[79] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on gpus. 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 681–696. USENIX Association, 2018.

[80] Chenxu Wang, Fengwei Zhang, Yunjie Deng, Kevin Leach, Jiannong Cao, Zhenyu Ning, Shoumeng Yan, and Zhengyu He. CAGE: complementing arm CCA with GPU extensions. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024.

[81] Pengli Wang, Bingyou Dong, Yifeng Cai, Zheng Zhang, Junlin Liu, Huanran Xue, Ye Wu, Yao Zhang, and Ziqi Zhang. Game of arrows: On the ({In-) Security} of weight obfuscation for {On-Device}{TEE-Shielded}{LLM} partition algorithms. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 279–298, 2025.

[82] Yongqin Wang, G. Edward Suh, Wenjie Xiong, Benjamin Lefaudeux, Brian Knott, Murali Annavaram, and Hsien-Hsin S. Lee. Characterization of mpc-based private inference for transformer-based models. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 187–197, 2022.

[83] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. Autodroid: Llm-powered task automation in android. In Weisong Shi, Deepak Ganesan, and Nicholas D. Lane, editors, *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking, ACM MobiCom 2024, Washington D.C., DC, USA, November 18-22, 2024*, pages 543–557. ACM, 2024.

[84] Xiaolong Wu, Dave Jing Tian, and Chung Hwan Kim. Building GPU tees using CPU secure enclaves with gevisor. In *Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC 2023, Santa Cruz, CA, USA, 30 October 2023 - 1 November 2023*, pages 249–264. ACM, 2023.

[85] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. Fast on-device LLM inference with npus. In Lieven Eeckhout, Georgios Smaragdakis, Kaitai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach, editors, *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pages 445–462. ACM, 2025.

[86] Jiajun Xu, Zhiyuan Li, Wei Chen, Qun Wang, Xin Gao, Qi Cai, and Ziyuan Ling. On-device language models: A comprehensive review. *CoRR*, abs/2409.00088, 2024.

[87] Mengwei Xu, Wangsong Yin, Dongqi Cai, Rongjie Yi, Daliang Xu, Qipeng Wang, Bingyang Wu, Yihao Zhao, Chen Yang, Shihe Wang, Qiyang Zhang, Zhenyan Lu, Li Zhang, Shangguang Wang, Yuanchun Li, Yunxin Liu, Xin Jin, and Xuanzhe Liu. A survey of resource-efficient LLM and multimodal foundation models. *CoRR*, abs/2401.08092, 2024.

[88] Zhenliang Xue, Yixin Song, Zeyu Mi, Le Chen, Yubin Xia, and Haibo Chen. Powerinfer-2: Fast large language model inference on a smartphone. *CoRR*, abs/2406.06282, 2024.

[89] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *CoRR*, abs/2412.15115, 2024.

[90] Mengda Yang, Wenzhe Yi, Juan Wang, Hongxin Hu, Xiaoyang Xu, and Ziang Li. Penetralium: Privacy-preserving and memory-efficient neural network inference at the edge. *Future Generation Computer Systems*, 156:30–41, 2024.

[91] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual*

Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022, 2022.

[92] Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Reetuparna Das, and Todd M. Austin. Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors. In *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*, pages 313–324. IEEE Computer Society, 2017.

[93] Jiyuan Zhang, Weiwei Jia, Siyuan Chai, Peizhe Liu, Jongyul Kim, and Tianyin Xu. Direct memory translation for virtualized clouds. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 287–304. ACM, 2024.

[94] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. Truspy: Cache side-channel information leakage from the secure world on ARM devices. *IACR Cryptol. ePrint Arch.*, page 980, 2016.

[95] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. Tinyllama: An open-source small language model, 2024.

[96] Zheng Zhang, Na Wang, Ziqi Zhang, Yao Zhang, Tianyi Zhang, Jianwei Liu, and Ye Wu. Groupcover: a secure, efficient and scalable inference framework for on-device model protection based on tees. In *Forty-first international conference on machine learning*, 2024.

[97] Ziqi Zhang, Chen Gong, Yifeng Cai, Yuanyuan Yuan, Bingyan Liu, Ding Li, Yao Guo, and Xiangqun Chen. No privacy left outside: On the (in-) security of tee-shielded dnn partition for on-device ml. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3327–3345. IEEE, 2024.

[98] Zongwei Zhou, Miao Yu, and Virgil D Gligor. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *2014 IEEE symposium on security and privacy*, pages 308–323. IEEE, 2014.

[99] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, and Dan Meng. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1450–1465. IEEE, 2020.

A Artifact Appendix

A.1 Abstract

The artifact contains the source code of our prototype system and the scripts for conducting the experiments presented in the paper.

A.2 Description & Requirements

The artifact is available at https://doi.org/10.5281/zenodo.17213486.

A.2.1 Hardware Dependencies. The artifact requires an Orange Pi 5 Plus board [17] (RK3588 CPU/NPU [21]). A standalone machine is required to build the artifact and communicate with the board using a USB-to-USB cable. The machine should have at least 8 GB of memory and 100 GB of free disk space.

A.2.2 Software Dependencies. The standalone machine should operate on a Linux OS (Ubuntu 22.04.3 tested) and have the following dependencies installed: OpenHarmony Device Connector for board communication, Python3 and its matplotlib library for plotting and analysis, and Docker for containerized builds.

A.3 Setup

The user needs to download the source code. Please refer to the `README.md` for details.

A.4 Evaluation Workflow

A.4.1 Major Claims.

- Claim **C1**: TZ-LLM can reduce TTFT by 76.1%∼90.9% compared to the Strawman baseline and incurs 5.2%∼28.3% overhead compared to the REE-LLM-Memory baseline. This is demonstrated by experiment **E1**, whose results are reported in Figure 10.
- Claim **C2**: TZ-LLM can increase decoding speed by 0.9%∼23.2% compared to the Strawman baseline and incurs 1.3%∼4.9% overhead compared to the REE-LLM baseline. This is demonstrated by experiment **E2**, whose results are reported in Figure 11.
- Claim **C3**: As more parameters are cached, partial parameter caching can reduce TTFT approximately linearly up to a threshold. This is demonstrated by experiment **E3**, whose results are reported in Figure 14.

A.4.2 Experiments.

- Experiment **E1** (approximately 60 compute-minutes): Run script `scripts/1-end-to-end-prefill.sh`, which evaluates the TTFT of TZ-LLM and other baselines across different benchmarks. The results are displayed in `plots/figure10.pdf`.
- Experiment **E2** (approximately 20 compute-minutes): Run script `scripts/2-end-to-end-decoding.sh`, which evaluates the decoding speed of TZ-LLM and other baselines across different models. The results are displayed in `plots/figure11.pdf`.
- Experiment **E3** (approximately 60 compute-minutes): Run script `scripts/3-caching.sh`, which evaluates the effect of partial parameter caching on the TTFT of TZ-LLM across different cache proportions and prompt lengths. The results are displayed in `plots/figure14.pdf`.