ASGARD: Protecting On-Device Deep Neural Networks with Virtualization-Based Trusted Execution Environments

Myungsuk Moon, Minhee Kim, Joonkyo Jung, Dokyung Song*
Department of Computer Science
Yonsei University

***Abstract*—On-device deep learning, increasingly popular for enhancing user privacy, now poses a serious risk to the privacy of deep neural network (DNN) models. Researchers have proposed to leverage Arm TrustZone's trusted execution environment (TEE) to protect models from attacks originating in the rich execution environment (REE). Existing solutions, however, fall short: (i) those that fully contain DNN inference within a TEE either support inference on CPUs only, or require substantial modifications to closed-source proprietary software for incorporating accelerators; (ii) those that offload part of DNN inference to the REE either leave a portion of DNNs unprotected, or incur large run-time overheads due to frequent model (de)obfuscation and TEE-to-REE exits.**

We present ASGARD, the first virtualization-based TEE solution designed to protect on-device DNNs on legacy Armv8-A SoCs. Unlike prior work that uses TrustZone-based TEEs for model protection, ASGARD's TEEs remain compatible with existing proprietary software, maintain the trusted computing base (TCB) minimal, and incur near-zero run-time overhead. To this end, ASGARD (i) securely extends the boundaries of an existing TEE to incorporate an SoC-integrated accelerator via secure I/O passthrough, (ii) tightly controls the size of the TCB via our aggressive yet security-preserving platform- and application-level TCB debloating techniques, and (iii) mitigates the number of costly TEE-to-REE exits via our exit-coalescing DNN execution planning. We implemented ASGARD on RK3588S, an Armv8.2-A-based commodity Android platform equipped with a Rockchip NPU, without modifying Rockchip- nor Arm-proprietary software. Our evaluation demonstrates that ASGARD effectively protects on-device DNNs in legacy SoCs with a minimal TCB size and negligible inference latency overhead.

I. Introduction

On-device deep learning is being increasingly adopted to enhance user privacy [52], [51], [20], [65], [72]. The idea is to conduct deep neural network (DNN) inference on mobile devices, which, by construction, prevents a *direct* exposure of sensitive user data such as user faces and fingerprints to remote servers. However, this shift is raising a new concern: model privacy. The leakage of DNN models could cause (at least) two harms, (i) first financially, as they constitute valuable (often mission-critical) intellectual properties, and (ii) in terms of user privacy, as the models could indirectly leak information about their training data [54], [16], [17]. Researchers in response proposed a range of on-device DNN model protection solutions [52], [72], [65], [45], [67], which predominantly use Arm TrustZone to create a trusted execution environment (TEE) and host DNN inference inside the TEE.

Unfortunately, however, these solutions have not seen adoption in practice. A recent study revealed that a significant portion of on-device DNN models remain completely unprotected [73], [63]. Some mobile applications do employ some form of protection, but such protection is ineffective while the models are *in use* for inference. Many of them are susceptible to dynamic model extraction attacks that simply dump the models during inference, launched from the rich execution environment (REE). If existing TEE-based on-device model protection solutions were adopted [52], [72], [65], [45], such dynamic model extraction attacks would have been impossible.

The challenges hindering a real-world adoption of existing model protection solutions stem primarily from the inherent limitations of Arm TrustZone. First, TrustZone requires *static* partitioning of physical resources into Normal World and Secure World. To make resources widely available to different applications, most of the resources are typically assigned to Normal World, leaving Secure World resource-constrained: it has (i) limited memory (e.g., typically 10-32MB [57]) and (ii) no access to accelerators. Due to this limitation, a line of prior work does not support using accelerators for DNN inference at all [20], or offloads a subset of DNN operators to REE-assigned accelerators either unprotected [52], or after costly encryption or obfuscation [65], [72], [95], [79]. Another line of work instead extends its TEE to accelerators [24], but substantially modifies the proprietary, privileged platform software, i.e., the secure monitor, undermining compatibility and increasing the trusted computing base (TCB) of the entire platform. Second, TrustZone's Secure World was designed to host vendor-specific, security-oriented OSs [18], which provide vendor-tailored execution environments for trusted applications. These environments, unfortunately, are not binary-compatible with the proprietary accelerator software stack (e.g., user-mode and kernel-mode drivers) available only for mainstream OSs in the REE, requiring substantial modifications to [24] or even complete rewriting of [59] the accelerator

*Corresponding author.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240449
www.ndss-symposium.org

software stack to host them inside a TEE.

Most recently, Siby et al. proposed an on-device DNN protection solution [67] that leverages Arm Confidential Compute Architecture (Arm CCA) [8], a planned extension to Armv9-A, instead of Arm TrustZone. While this solution does not suffer from TrustZone's limitations, no real hardware with CCA is available yet. Hence, the solution cannot be deployed in widespread Armv8-A devices.

This paper proposes ASGARD, a new on-device DNN protection solution built upon *virtualization-based TEEs* emerging in mobile platforms [50], [40], [26], [43]. ASGARD's virtualization-based approach addresses the performance and compatibility problems of existing on-device DNN protection solutions, as follows. First, by directly exposing the accelerator hardware to virtualization-based TEEs (or enclaves) via I/O passthrough, ASGARD can fully accelerate DNN inference. Second, ASGARD's monitor that enforces isolation between enclaves can be implemented completely at the hypervisor level in EL2, without requiring modification to the proprietary EL3 secure monitor. Third, by using commodity operating systems within the virtual machines, which have access to the raw accelerator interface, ASGARD can use an unmodified, proprietary accelerator software stack. Last, unlike DNN protection solutions that require hardware extensions [67], ASGARD's TEE was designed to protect DNNs on legacy Armv8-A SoCs. To summarize, ASGARD is a solution that can be readily adopted to protect DNNs used by existing mobile applications, without requiring modification to proprietary software nor causing significant performance degradation.

The high-level approach of ASGARD may seem straightforward, but it introduces a unique set of problems prior work has not fully addressed. First, designing a secure I/O passthrough for virtualization-based TEEs on *legacy* Armv8-A SoCs with limited TEE primitives remains an unresolved problem. Existing virtualization-based TEEs in mobile devices offer secure virtualization for CPUs only [40], [26], do not provide an end-to-end I/O passthrough design [41], or do not support legacy Armv8-A SoCs [69], [85], [43]. Second, the virtual machine abstraction and secure I/O passthrough used by ASGARD brings TCB overheads. An entire IOMMU driver and its dependencies, and even an entire virtual machine should be included in the TCB for model protection. Third, despite ASGARD's use of hardware-accelerated processor virtualization and direct access to the accelerator, it could still degrade DNN inference latency. DNN accelerators frequently raise interrupts whenever they encounter operators that they do not support. Since interrupts are managed by the REE-side host, such interrupts lead to costly TEE-to-REE exits, degrading the performance of DNN inference.

Our key contribution lies in addressing these problems introduced by the use of virtualization-based TEEs for model protection, with (i) a concrete, end-to-end design of secure I/O passthrough for virtualization-based TEEs that can be implemented on legacy Armv8-A SoCs, (ii) aggressive yet security-preserving hypervisor and enclave debloating techniques that can reduce the size of the TCB for model protection, and (iii) an exit-coalescing DNN execution planning technique that can mitigate costly TEE-to-REE exits.

We implemented a prototype[1] of ASGARD on Rockchip RK3588S, a commodity Armv8.2-A SoC that is equipped with an integrated neural processing unit (NPU). We based our implementation of virtualization-based TEEs on *protected* KVM (or *p*KVM for short) [23], a virtualization-based *software-secure* TEE that has recently been implemented in Linux KVM for Armv8-A [58], [22]. We extended *p*KVM with ASGARD's secure I/O passthrough for the Rockchip NPU. For ASGARD's exit-mitigating DNN execution planning, we implemented a custom DNN partitioning tool and DNN execution runtime atop Rockchip's proprietary DNN compiler and runtime.

Our prototype confirms ASGARD's compatibility with existing proprietary software on a legacy SoC, and our thorough security analysis validates that the TEE was indeed securely extended to the NPU. We also show that ASGARD's TCB size and DNN inference latency optimization techniques are all effective. Through our aggressive virtual machine image specialization, we show that ASGARD can achieve an image size of 17.439MB for hosting accelerated DNN inference. We applied ASGARD's exit-coalescing DNN execution planning to multiple DNNs commonly used for object detection and text understanding in mobile devices. The results show that ASGARD can significantly reduce virtualization overheads on DNN inference latency by minimizing a number of costly TEE-to-REE exits. In the best-performing instance, ASGARD-protected DNN inference achieved a latency even lower than unprotected inference (-1.36% on SSD-MobileNetV1) by reducing the number of TEE-to-REE exits from 13 to 2.

In summary, our contributions are as follows:

- We propose ASGARD, a new on-device DNN protection solution that, for the first time, uses virtualization-based TEEs with secure I/O passthrough to address the performance and compatibility issues of existing TrustZone-based solutions.
- We propose both TCB size and performance optimization techniques, which significantly reduce the TCB size and DNN latency overheads of virtualization-based TEEs.
- We implemented ASGARD on a legacy Armv8-A hardware platform by extending *p*KVM that offers software-secure virtualization-based TEEs, and thoroughly analyzed ASGARD's security and performance properties.

II. Background & Motivation

A. Arm Architecture Basics

Privilege Levels and Address Spaces. Armv8-A and Armv9-A CPUs execute in one of the four privilege levels called Exception Levels (ELs): EL0 for application code, EL1 for OS kernel, EL2 for hypervisor, and EL3 for secure monitor [7]. They employ two stages of address space translation. Stage-1 translation uses page tables controlled by the OS kernel at EL1, and translates virtual address (VA) to intermediate physical address (IPA). Page tables for Stage-2 translation are controlled by software running at EL2 or higher. The hypervisor at EL2 uses them to control the view of physical memory in a virtual machine [9]. The guest OS thinks that an IPA is an address in physical memory, but Stage-2 translation actually converts IPAs to physical addresses.

[1] Artifact available at: https://github.com/yonsei-sslab/asgard

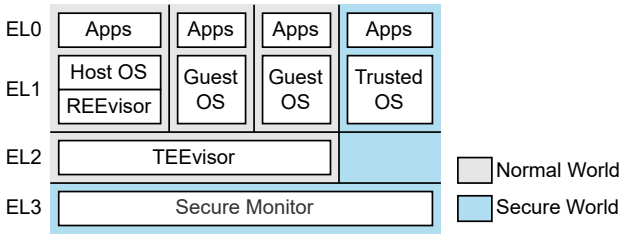


Fig. 1: Architecture of *protected* KVM on Armv8-A.

Arm TrustZone. TrustZone horizontally separates the CPU execution mode into two, Normal World (NW) and Secure World (SW), and security-critical services like key management are placed in SW [6]. Both NW and SW span over EL0 to EL2, and, in the case of Armv8-A, EL3 belongs to SW; we refer to each world's exception level as N-ELx and S-ELx for NW and SW respectively. SW software stack is composed of secure monitor at S-EL3, hypervisor at S-EL2 (only supported since Armv8.4-A), trusted OS kernel at S-EL1 and trusted applications at S-EL0. Resources such as physical memory are partitioned between NW and SW via TrustZone hardware, such as TZASC and TZPC [85].

B. Virtualization-Based Trusted Execution Environments

TEEs (or "enclaves") refer to software execution environments protected from (i) privileged software adversaries residing in the REE, and (ii) physical adversaries having varying degrees of physical tampering capabilities. *Virtualization-based* TEE solutions provide enclaves in the form of virtual machines, to which access is strictly controlled by a small, trusted hypervisor [41] (henceforth "TEEvisor"). These TEE solutions typically employ an additional, unprivileged hypervisor (henceforth "REEvisor") alongside the TEEvisor. The REEvisor implements the rich functionalities of traditional hypervisors including resource management, which do not require elevated privileges. Researchers have proposed a variety of virtual machine enclave design on Arm [50], [40], [43], [23], which have varying security and performance characteristics.

Of these, we detail one concrete design relevant to our work: *protected* Kernel Virtual Machines (*p*KVM) [23]. *p*KVM provides *software-secure* virtual machine enclaves, meaning that they are protected from privileged *software* adversaries on the REE side, but not from adversaries with physical access to hardware. As depicted in Fig. 1, *p*KVM accomplishes this by introducing a small TEEvisor in N-EL2, which strongly isolates enclaves. The host kernel running at N-EL1 includes the REEvisor that comprises the majority of the KVM hypervisor module and operates without requiring elevated privileges.

*p*KVM controls access to page frames of physical memory through an ownership concept. Each page frame is exclusively owned by either a guest, the host, or the TEEvisor. The owner of a frame can transfer the ownership by *donating* it, and a non-owner can *borrow* it from the owner for temporary access. The TEEvisor fully mediates donation and borrowing, enforcing a discretionary access control policy where the owner decides whether to donate or allow borrowing the frame. However, the REEvisor still *manages* the page frames. The REEvisor can always reclaim the ownership of any enclave-owned frame, but the TEEvisor ensures that the frame is wiped before transferring the ownership to the REEvisor. The TEEvisor enforces this policy by configuring the MMU's Stage-2 translation.

In addition to protecting enclave memory from CPU-side adversaries, the TEEvisor also protects it from DMA-capable peripherals by configuring the translation of the IOMMU (also called SMMU on Arm). *p*KVM requires all DMA-capable peripherals in an SoC to be placed behind an IOMMU to prevent DMA attacks to enclaves. To this end, IOMMU drivers, originally running in N-EL1, are split into the REEvisor and TEEvisor [26], [15]. *p*KVM's TEEvisor assigns all peripherals to the host, by exposing only host-owned page frames to them via IOMMU page table configuration.

Table I: Limitations of existing TrustZone-based on-device model protection solutions, in comparison with our solution.

	P1. Efficiency		**P2. Compatibility**	
	Fully Accelerated	**Negligible Overheads**	**Unmodified Accelerator Driver**	**Unmodified Secure Monitor**
DarkneTZ [52]	✗	✓	Not applicable[1]	✓
ShadowNet [72]	✗[2]	✗	✓	✓
GPUReplay [59]	✓	✓	✗	✓
StrongBox [24]	✓	✓	✗	✗
ASGARD (Ours)	✓	✓	✓	✓

[1] Acceleration not supported.
[2] Acceleration only supported for linear DNN layers.

C. Limitations of Prior Work

Existing approaches to on-device DNN protection can be classified into (i) those that offload part of DNN computation to REE-assigned accelerators [52], [72], and (ii) those that run DNN inference entirely within a TEE using TEE-assigned accelerators [59], [24]. Unfortunately, as summarized in Table I, they suffer various efficiency (referred to as **P1**) and compatibility problems (**P2**). These problems stem mainly from their use of TrustZone, and, specifically, from the following properties TrustZone: (i) static partitioning of physical resources making SW resource-constrained, and (ii) uses of proprietary, specialized OSs in SW [18], as detailed below.

P1: Low Efficiency. TrustZone requires accelerators be statically assigned to either NW or SW, and accelerators are typically left in NW to make it widely available to different applications. To utilize NW-assigned accelerators, researchers proposed *partition-and-offload* approaches [79], [72], [52], [95], which offload a portion of DNN computation to NW (i.e., REE). A line of work offloads the *initial* layers of a DNN to the REE [52], while executing the last layers—which are more susceptible to white-box membership inference attacks [66]—in SW on CPUs. The execution of REE-offloaded layers can be accelerated, but they are left completely unprotected.

Another line of work proposed to protect the REE-offloaded layers via encryption or obfuscation [79], [72], [95]: Specifically, this line of work offloads the *linear* layers to REE-assigned accelerator after encrypting or obfuscating their weights, while running non-linear layers on CPUs in SW. This approach, however, could incur non-negligible run-time overheads (see §VI-C). The overheads of these encryption- or obfuscation-based *REE-offloading* solutions come from

(i) costly world switches between SW and NW required for every non-linear-to-linear layer transition, and (ii) blinding and unblinding (e.g., encrypting and decrypting) of the input and output of each NW-offloaded computation.

P2: Low Backward Compatibility. Researchers also proposed to use an accelerator after statically assigning it to SW [59], [24]. In contrast to partition-and-offload approaches, this approach can fully accelerate DNN inference, and without incurring REE-offloading overheads. The SW OS in TrustZone, however, is a specialized OS (e.g., OP-TEE OS [57]) that is incompatible with an existing accelerator driver stack written for the NW OS. Park and Lin proposed to employ a minimal driver stack written specifically for the SW OS [59], which supports replaying recorded (i.e., already seen) accelerator tasks only. Though this approach only marginally increases the size of the TCB (by introducing new code only to S-EL0 and S-EL1), it does not support arbitrary (i.e., unseen) DNN inference tasks, nor any tasks submitted from NW.

Deng et al. later demonstrated that dynamically yet securely serving NW- and SW-originating DNN inference requests on a single accelerator is possible in TrustZone [24]. However, this approach requires nontrivial modifications to the secure monitor at EL3, the most privileged component that isolates NW and SW. Not being compatible with vendor's proprietary firmware raises the bar for adoption. Moreover, it bloats the TCB of the entire platform: accelerator-specific code must be added to the secure monitor executing at EL3.

III. ASGARD Overview

We now present ASGARD, a new TEE-based on-device DNN protection approach for Armv8-A SoCs, which can address the aforementioned problems. Based on our observation that these problems stem mainly from the inherent limitations of TrustZone, we designed ASGARD such that it uses *virtualization-based TEEs* [50], [41] to protect on-device DNNs.

The virtual machine abstractions significantly alleviate the problems of TrustZone-based solutions, as follows. First, by directly exposing an accelerator to virtual machine enclaves via I/O passthrough, DNN inference can be fully accelerated, without requiring costly obfuscation and deobfuscation of DNN layer activations [72] (addressing **P1** for the most part). Second, the raw hardware abstraction of the CPU *and* the accelerator exposed to the virtual machines allows them to use an unmodified accelerator software stack. In addition, enforcing isolation between the REE and TEEs can be done entirely by the TEEvisor running at EL2, without requiring any modification to the EL3 secure monitor (addressing **P2**).

A. Design Goals

To make ASGARD a practical and readily adoptable TEE solution that goes beyond merely addressing the aforementioned problems, we introduce three concrete design goals.

G1: Strong Protection on Armv8-A SoCs. ASGARD should protect DNNs at all times——while they are in transit, at rest, and in use for DNN inference—against the same on-device, REE-side adversaries considered by TrustZone's Trusted Applications, on legacy Armv8-A SoCs.

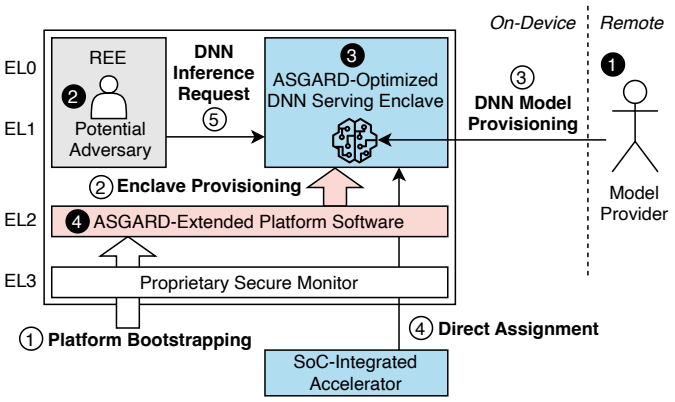


Fig. 2: High-level overview of ASGARD.

G2: Minimal Trusted Computing Base Overheads. ASGARD's TEE should incorporate minimal, controlled code within the TCB. ASGARD's DNN protection guarantees should rely upon a TCB substantially smaller than that of unprotected DNN inference using REE-assigned accelerators, and only marginally larger than that of existing virtualization-based TEEs.

G3: Near-Zero Run-Time Overheads. ASGARD should achieve DNN inference latency significantly lower than that of existing REE-offloading solutions, and approach near-zero run-time overheads with respect to the latency of unprotected, accelerated DNN inference in the REE.

B. Threat Model & Assumptions

There are three parties involved in ASGARD's on-device DNN protection: (i) a remote model provider who wants to protect their model (❶ in Fig. 2), (ii) an on-device, REE-side privileged adversary who aims to steal the model (❷), and (iii) a trusted DNN serving enclave (❸) built upon ASGARD-extended platform software (❹). We assume that the adversary fully controls the REE including the REEvisor. The trusted platform includes the TEEvisor, which enforces isolation between virtualization domains. This is in line with existing approaches to virtualization-based TEEs on Arm-based platforms [35], [42], [40], [26], [41], in that they trust their own privileged hypervisors. Like prior TrustZone-based on-device model protection solutions [52], [65], [72], [24], we also trust SW, as privileged software in SW can freely access the entire NW (including the TEEvisor) in Armv8-A platforms.

In summary, our threat model assumes *software* adversaries in the REE, and we aim to provide *software-secure* enclaves on *commodity* Armv8-A platforms. We note, however, that our threat model can be made stronger, thereby providing *hardware-secure* enclaves, when additional hardware support is available. We further discuss how ASGARD can provide further model protection with hardware support in §VII.

We also assume that an SoC-integrated accelerator such as an NPU is present and behind a dedicated IOMMU. When the accelerator performs direct memory access (DMA), the IOMMU translates I/O virtual addresses into physical memory addresses. The SoC is assumed to have an accelerator reset interface, which can be used to reset all mutable state of the

accelerator to a known good state (e.g., the initial state). We do not consider physical adversaries who can tamper with the accelerator or the bus it is attached to; for example, an attacker cannot attach a rogue accelerator to the bus. We also assume that the IOMMU and the accelerator reset interface cannot physically be tampered with. We describe, however, how certain classes of physical attacks can be defeated when the accelerator supports measured boot and attestation (see §VII). Finally, following prior work on TEE-based model protection [52], [72], we put side-channel attacks out-of-scope, though we extensively discuss existing and potential side-channel-based model extraction attacks in §VII.

C. Key Techniques

We achieve the aforementioned design goals by designing and implementing a set of techniques within both the DNN serving enclave (❸ in Fig. 2) and the platform software at EL2 (❹), which we summarize below.

Secure Accelerator I/O Passthrough (§IV-A). Existing virtualization-based TEEs for Armv8-A either offer secure virtualization for CPUs only [40], [23], or do not provide an end-to-end design for secure I/O passthrough [41]. To fill this gap, we propose a concrete, end-to-end design of secure accelerator I/O passthrough that can be realized on commodity Armv8-A SoCs. The key components of our design include Stage-2 control, IOMMU control, and accelerator reset control in the TEEvisor, which are used to enforce a set of security invariants. By establishing trust in these components that can be traced back to the platform's hardware root-of-trust via chain-of-trust, ASGARD systematically defeats a wide range of attacks from strong REE-side adversaries.

TCB[2] and Attack Surface Reduction (§IV-B). ASGARD's TCB overheads come from both (i) the new code introduced by ASGARD to the TEEvisor, contributing to the platform-level TCB, and (ii) the image of the enclave that serves DNN inference requests, which contributes to the application-level TCB that model protection guarantees rely upon. In particular, virtual machine enclaves span both user- and kernel-mode (i.e., EL0 and EL1, respectively), unlike user-mode enclaves that only use EL1 [14], [94]. This can easily bloat the application-level TCB, making it challenging to achieve **G2**. We propose a suite of aggressive TCB reduction techniques, which work by optimizing ASGARD's enclave as a single application, and by delegating resource management to the REE.

DNN Inference Latency Optimization (§IV-C). The virtual machine abstraction used by ASGARD introduces run-time overheads that can inadvertently increase DNN inference latency, posing challenges in achieving **G3**. The overheads of the virtual machine abstraction increase further when using virtualization techniques designed specifically for TEEs [50], [43], [40], [41], due to its higher trustworthiness and protection guarantees than those of traditional virtualization. Based on our finding that TEE-to-REE exits during DNN inference are the main source of overheads, we propose exit-coalescing DNN execution planning, an application-level TEE-to-REE exit mitigation technique.

[2]We classify the TCB into an application- and platform-level TCB. The application-level TCB encompasses all components responsible for enforcing DNN model confidentiality. This expands upon the platform-level TCB, comprising the secure monitor at EL3 and the TEEvisor at EL2, which are responsible for isolation between the REE and TEEs.

D. Operational Overview

ASGARD is a virtualization-based TEE designed to protect on-device DNN models at all times—while they are in transit, at rest, and in use for DNN inference. We sketch how ASGARD achieves this, by describing the entire operation of ASGARD in chronological order, starting from platform bootstrapping to DNN inference serving (i.e., from ① to ⑤ in Fig. 2).

① **Platform Bootstrapping.** ASGARD uses the platform's standard secure bootstrapping, and ASGARD's DNN protection guarantees can therefore be traced back to the platform root-of-trust. Commodity Armv8-A SoCs support hardware root-of-trust, and, together with chained measurements of the trusted platform software, an authentic image of the TEEvisor starts running at EL2, when the secure boot process finishes. The bootloader verifies the host kernel image (which contains both the REEvisor and TEEvisor) and the device tree, after which the host kernel starts booting. During the early boot of the host kernel, and before the control is first transferred to the REE, ASGARD reserves security-sensitive resources that need to be owned by the TEEvisor for secure accelerator I/O passthrough.

② **Enclave Provisioning.** ASGARD then provisions a virtual machine enclave. The REE populates the initial content—the kernel and root file system—of the enclave. The enclave successfully boots only when (i) it is an enclave offered by the genuine platform, (ii) the device is in an OEM-locked state, and (iii) the enclave image is signed by the model provider. The platform, during enclave creation, secretly provisions an attestation key (i.e., a signing key bound to the enclave by the platform) and a sealing key (i.e., a per-enclave secret that remains the same across enclave boots) to the enclave. ASGARD uses the former to authenticate itself to the remote model provider, and the latter to seal DNNs before enclave shuts down.

③ **DNN Model Provisioning.** ASGARD can now load the DNN into the private memory of the enclave. If the enclave is booted for the first time, the DNN is obtained from the remote model provider. Specifically, the model provider engages in a standard authenticated key exchange with the enclave, authenticating the enclave using the enclave's attestation key that is anchored to the platform root-of-trust. Then the remote model provider transmits the DNN to the enclave through this secure, authenticated channel, ensuring model confidentiality during transmission. Before the enclave shuts down, ASGARD seals the DNN and stores it in an untrusted local storage. On subsequent boots of the same enclave, a sealed DNN is retrieved from this local storage, and then unsealed into the private memory.

④ **Direct Accelerator Assignment.** Then, ASGARD securely assigns an accelerator to the virtual machine enclave so that ASGARD-protected DNN inference serving can be accelerated. ASGARD extends the TEEvisor to expose the accelerator to the enclave, such that strong isolation between enclaves is maintained (see §IV-A). ASGARD's extension to the TEEvisor, however, increases the size of the TCB; we employ multiple strategies to mitigate both platform- and application-level TCB bloat (see §IV-B).

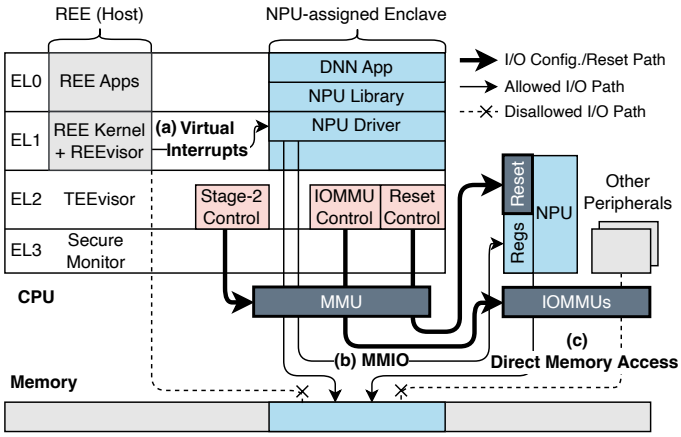


Fig. 3: ASGARD's passthrough virtualization of an NPU. The NPU-assigned enclave and the NPU interact via (a) virtual interrupts, (b) MMIO, and (c) DMA. The TEEvisor at EL2 enforces isolation between the REE and enclaves by controlling MMU's Stage-2, IOMMU, and NPU's reset interface.

⑤ **Serving DNN Inference Requests.** Now that ASGARD has securely provisioned the model to the virtual machine enclave and also has the accelerator assigned to it, a user residing in the REE can make inference requests to the enclave by sending an input to it. ASGARD executes DNNs per so-called exit-coalescing planning we propose to mitigate the overhead of TEE-to-REE exits (see §IV-C). Following prior work [52], [95], the enclave, after performing inference in it, returns the end result (e.g., prediction labels) to the user without revealing the confidence scores. Model confidentiality is maintained throughout this process against REE-side adversaries attempting model exfiltration.

IV. Detailed Design

A. Secure Accelerator I/O Passthrough

ASGARD employs passthrough I/O virtualization illustrated in Fig. 3, where (i) an accelerator-assigned enclave has direct access to the accelerator's interface, and (ii) the accelerator has direct access to the enclave-owned page frames. This allows ASGARD to fully accelerate DNN inference and be compatible with an unmodified accelerator software stack.

Although the initial line of work proposing virtualization-based TEEs for Armv8-A platforms outlines how to support I/O passthrough by protecting the IOMMU interface [50], [41], [40], they do not provide a concrete, end-to-end design. Sun et al. recently proposed LEAP [71], which provides a concrete virtual-machine-like TEE design that supports I/O passthrough. However, as discussed in §VIII, LEAP requires an entire physical CPU core to be dedicated to its enclaves, severely limiting efficient CPU core utilization.

Our contribution here is the first concrete, end-to-end design of secure accelerator I/O passthrough tailored for virtualization-based TEEs on legacy Armv8-A SoCs. To ensure that the design systematically mitigates all typical threats posed by REE-side software adversaries to the confidentiality of DNN models during inference, we introduce three invariants:

INV1 (Complete Mediation). The TEEvisor must have a full, exclusive control over all *sensitive* interfaces of the CPU cores and peripherals.[3]

INV2 (Exclusive Assignment). The accelerator should be assigned to no more than one virtual machine enclave at any point in time.

INV3 (No Residual State). After the accelerator has been used by a virtual machine enclave, there should be no residual states or leftover data within the accelerator.

Exclusive Stage-2 & IOMMU Control. After exposing the accelerator to enclaves via I/O passthrough, ASGARD's TEEvisor at EL2 shown in Fig. 3 should maintain exclusive control over all virtual machine isolation mechanisms, i.e., sensitive interfaces (**INV1**), comprising:

- *The Stage-2 translation of MMUs*: Armv8-A assigns control of this only to EL2 (i.e., TEEvisor) or higher.
- *The register address spaces of* all *IOMMUs, and all memory regions storing their page tables*: These can be protected by using the Stage-2 page table management interface (which the TEEvisor has exclusive access to) that we detail below.
- *The sensitive interface of the accelerator itself*: This can be protected by configuring Stage-2 page tables as well, although commodity accelerators including the NPU we used in our evaluation do not have sensitive interfaces.

The TEEvisor initially establishes control over sensitive interfaces during the platform's secure boot, i.e., before adversaries could interact with the platform. During secure boot, the TEEvisor reads the physical addresses of the sensitive MMIO regions from the verified device tree. Then, the TEEvisor assigns initial ownership of the page frames corresponding to these sensitive regions to itself, and never shares them with the host nor any enclave. Also, the TEEvisor creates the IOMMU page tables, another type of sensitive resources, in page frames it owns, and never shares them with others either. Exclusive ownership over these page frames is maintained, because TEEvisor-owned page frames, unlike enclave-owned ones (see §IV-B), cannot be reclaimed by the REEvisor.

Trustworthy Reset Control. Since ASGARD time-shares the accelerator across different security domains, TEEvisor's reset control must be able to clear all the residual state of the accelerator at domain transitions (i.e., **INV3**). The TEEvisor achieves this by using the accelerator's reset interface. Specifically, ASGARD assigns ownership of the reset interface to the TEEvisor during secure boot to ensure that it can always access the authentic reset interface. In addition, ASGARD requires the trustworthiness of this reset interface itself to be carefully assessed for each given accelerator. Reset-based time-sharing typically demands hardware support (i) to *fully* reset all dynamic states, including registers and memory as well as microarchitectural and physical ones [10], [13], and (ii) to attest to the freshness of the reset [90]. However, since our target is a legacy SoC, we can only modify the accelerator *software* (i.e., firmware) to add a function that wipes all of its software-writable registers and memory, and have the trusted

[3]The interface of CPU cores and peripherals consists of both their registers and instruction set. It is called *sensitive* if accessing the registers or executing certain instructions affects the isolation between virtual machines [60], [64].

Table II: New hypercalls introduced by ASGARD in the guest-side TEEvisor interface.

Hypercall	Description
`acquire_device (d_id, d_gpa)`	Request TEEvisor to assign the given accelerator (specified by `d_id`) to the calling enclave, exposing the accelerator's register space at the given guest-physical address (specified by `d_gpa`).
`release_device (d_id)`	Request TEEvisor to remove assignment of the given accelerator (specified by `d_id`) from the calling enclave.

TEEvisor's reset control (i) reload the authentic firmware, and (ii) invoke this wipe function as part of its reset procedure. Though physical or microarchitectural states could persist even after such software-based wiping, our threat model does not consider physical adversaries nor side-channel attacks.

Secure Accelerator Assignment. With exclusive ownership securely established over the sensitive interfaces of the platform (**INV1**), ASGARD's TEEvisor can now securely assign the accelerator to an enclave, as exemplified in Fig. 3.

- *Establishing Protected I/O Paths*: Exclusive assignment of the accelerator (**INV2**) involves establishing enclave-accelerator I/O paths isolated from any other enclaves nor the host. First, ASGARD creates protected DMA paths by exposing the page frames owned by the enclave to only the accelerator, and not any other peripherals. ASGARD does so by configuring every IOMMU on the platform such that only the IOMMU of the accelerator (and no other IOMMU) has page table entries that translate I/O virtual addresses (IOVAs) to the enclave-owned pages frames. Second, only a single, accelerator-assigned enclave must be able to access the accelerator's MMIO region (except for its *sensitive* registers, if any). This is done by donating the ownership of their corresponding page frames to the enclave.
- *Secure Assignment Switching*: When ASGARD switches accelerator assignments, the TEEvisor re-establishes all the protected I/O paths by reconfiguring the MMU, IOMMU, and interrupt forwarding. To ensure exclusive assignment during this transition (**INV2**), the TEEvisor updates the MMU translation entries, and flushes all relevant entries cached in the TLB. Similarly, to re-establish DMA I/O paths, the TEEvisor updates the IOMMU translation entries, and flushes the IOTLB. To reduce switching latency, it reuses existing entries initialized during enclave boot rather than repopulating them. In addition, ASGARD's TEEvisor resets the accelerator during the switch, ensuring that no residual state remains inside the accelerator (**INV3**). Most of SoC-integrated accelerators, including Rockchip NPU we used to prototype ASGARD, expose a reset interface. To ensure that the TEEvisor always has access to the *authentic* reset interface (therefore it can reliably trigger accelerator resets), ASGARD, during secure boot, assigns the TEEvisor the initial ownership of its corresponding page frame. The TEEvisor never donates this page frame, but permits borrowing of it, because borrowing does not hinder its access to the reset interface.
- *Controlling Assignments*: We introduce two hypercalls to the TEEvisor's guest interface for controlling accelerator assignments: `acquire_device()` and `release_device()`, which we summarize in Table II. An enclave that has an accelerator added at startup (see §IV-B) can voluntarily request to *acquire* (by invoking the former), and *release* the accelerator (the latter) at any time, and the TEEvisor handles the requests per the following policy: it assigns the accelerator to the calling enclave only if the accelerator (i) was added to the enclave, and (ii) has currently not been acquired by any enclave; it removes accelerator assignment only if the calling enclave has acquired the accelerator.

Secure Accelerator Reclamation. A key design principle of TEEs is the separation of privileges: unprivileged resource management and privileged resource isolation (see §II-B). The REEvisor is thus responsible for managing the enclave-owned I/O resources (e.g., the page frames corresponding to the accelerator's MMIO regions and DMA pages) like any other resources. This means that the REEvisor can always reclaim the accelerator. To preserve model confidentiality during reclamation, the TEEvisor takes the following measures before the REEvisor can start accessing the reclaimed page frames: (i) the page frames being reclaimed are wiped, (ii) the Stage-2 and IOMMU page table entries of the enclave that map to the reclaimed page frames are removed, and (iii) when the accelerator's MMIO regions are being reclaimed, the accelerator assignment is forcefully switched to the REE. To prevent the DNN serving enclave from misbehaving (e.g., leaking models) upon reclamation, ASGARD relies on existing liveness-checking mechanisms (e.g., watchdogs) employed in accelerator drivers. By using this mechanism, the enclave can detect reclamation and safely terminate the DNN inference process, preventing any inadvertent leakage of the model.

B. TCB and Attack Surface Reduction

ASGARD's virtualization-based DNN protection introduces platform- and application-level TCB overheads. At the platform level, the TCB increases, because secure I/O passthrough adds additional components to the TEEvisor at EL2 (though not to EL3's secure monitor). Specifically, the IOMMU driver needs to be included. A naive approach would be to add the IOMMU driver along with all of its dependencies (i.e., all the kernel-provided symbols referenced by the driver) to the TEEvisor. This approach, however, would unnecessarily bloat the TCB. An alternative approach would be to retain the dependencies within the REEvisor, and allow the IOMMU driver to switch to the REEvisor as needed. This approach, however, would increase the TEEvisor's attack surface.

The TCB further increases at the application level, because ASGARD runs a commodity OS (e.g., Linux) within an accelerator-assigned, DNN-serving enclave. Although this ensures that the proprietary driver stack for the accelerator can run without requiring modification, it expands the TCB responsible for ensuring model confidentiality to include the entire guest OS in addition to the platform-level TCB.

Unprivileged IOMMU Power and Clock Management. To debloat the TEEvisor, the power and clock management of the IOMMU can be delegated to the REEvisor. The rationale here is that power and clock management fall under resource *management* rather than resource *isolation*, which does not directly affect the confidentiality and integrity guarantees of the enclave. This delegation not only removes the power and clock management component of the IOMMU driver, but also their

dependencies. To preserve the security invariants described in §IV-A after delegation, when the REEvisor turns off the IOMMU, ASGARD must ensure that the accelerator cannot bypass ASGARD's IOMMU-based memory protection [15]. Our finding is that this is not possible in the Armv8-A SoC we used to prototype ASGARD, because each pair of the peripheral and IOMMU share the same power and clock domains. That is, when the REEvisor turns off an IOMMU, its corresponding peripheral is also turned off, preventing the peripherals from bypassing IOMMU-based memory protection. ASGARD thus safely delegates the power and clock management of the IOMMU to the REEvisor.

Coarse-Grained IOMMU Control. We then employ coarse-grained IOMMU control to reduce the TEEvisor's attack surface. Prior work on I/O passthrough exposes a fine-grained IOMMU control interface to guests by (para)virtualizing IOMMUs [5], [78], [11]. Adding such support requires new hypercalls (e.g., for mapping and unmapping of DMA buffers) to be exposed to individual guests. This expands the TEEvisor's attack surface, without providing additional protection guarantees for the model. For this reason, ASGARD does not expose any IOMMU interface to enclaves. Instead, when the accelerator is assigned to the enclave, ASGARD exposes the entire enclave-owned memory to the accelerator. To this end, ASGARD installs a small IOMMU driver inside the guest enclave, which simply maintains identity mappings between the IOVAs (i.e., DMA addresses) and guest physical addresses. The accelerator driver operates with this IOMMU driver, by using guest physical addresses as DMA addresses. These DMA addresses, as they are used by the accelerator, translate to host physical addresses by the TEEvisor-controlled IOMMU.

Still, IOMMU-mapped pages are *managed* by the REEvisor. For this, we expose the existing IOMMU driver interface of the Linux kernel to the REEvisor as hypercalls, namely: `iommu_map_page()` and `iommu_unmap_page()`. When an enclave starts with an accelerator added, the REEvisor first pins the entire guest physical memory, and then invokes the former hypercall to have the TEEvisor create IOMMU translations from guest physical addresses to enclave-owned physical memory addresses. These translations are only activated when the enclave successfully acquires the device. When the REEvisor later invokes the latter hypercall, the TEEvisor deactivates and removes the IOMMU translations, and then the REEvisor subsequently unpins the guest physical memory.

Relaxed Intra-Enclave Privilege Separation. We then relax the privilege separation inside the DNN serving enclave to reduce its size. The user-mode DNN inference runtime already has access to the model, so the user-kernel mode separation within ASGARD's enclaves does not enhance the protection of the model any further. Also, since the enclave runs a single application, no separation between users is needed. We therefore disable user-kernel attack surface reduction and advanced access control features of the guest Linux kernel, including Seccomp [75] and SELinux [29]. We still keep, however, other security features such as block device integrity protection (e.g., `dm-verity` [28]) and exploit hardening features (e.g., control flow integrity) enabled to protect and harden the enclave from potential attacks from outside the enclave.

Removing Unused Code from Enclave Images. Executing accelerated DNN inference inside the enclave requires only a minimal set of kernel features. We identify and enable the bare minimum of the features in the kernel, and compile it with full link-time optimization to aggressively eliminate dead code and further reduce the TCB. As we show in §VI-B, applying these specialization strategies results in a substantially smaller kernel image compared to existing general-purpose kernel images optimized for smaller size, such as Google's Microdroid [26] and Firecracker [3], while still supporting DNN inference. In addition to the kernel, we specialize the root file system used by the guest enclave to further debloat the TCB. The root file system image commonly contains various binaries that are installed as part of the default packages. Most of them are unnecessary for ASGARD-protected DNN inference. By designating the DNN application as the `init` program that runs as the first program after an enclave boot, ASGARD can eliminate most of the binaries from the root file system image, retaining only those actually used during DNN inference.

C. DNN Inference Latency Optimization

ASGARD exposes the accelerator directly yet securely to enclaves via secure I/O passthrough (see §IV-A), resulting in full acceleration of DNN inference while maintaining compatibility with existing proprietary software. Unfortunately, however, ASGARD's virtualization-based design could inadvertently impose run-time overheads on DNN inference latency, due to costly virtual machine enclave exits. An enclave exit followed by an entry under ASGARD necessitates multiple exception level transitions back and forth, which involve costly context save, cleanup, and restore operations [21]. To achieve minimal run-time overheads on DNN inference (**G3**), ASGARD incorporates a new DNN execution planning strategy, an application-level optimization that we propose to significantly reduce the frequency of voluntary TEE-to-REE exits during DNN inference inside enclaves.

Coalescing CPU-Fallback DNN Operators. Handling hardware interrupts entails multiple TEE-to-REE exits. Exits occur in both the delivery and acknowledgement paths of interrupts (also known as end-of-interrupt) [8], [23]: An interrupt, when raised, is initially delivered to the host kernel on the REE side (requiring a TEE-to-REE exit when the CPU was serving a TEE), and then forwarded to the guest enclave as a *virtual* interrupt entering back to the TEE from the REE. Once the virtual interrupt is acknowledged by the guest enclave, it must then be acknowledged by the REE-side host kernel, requiring a voluntary TEE-to-REE exit again. This means that each interrupt raised by the accelerator causes one or more exits from and entries into the enclave.

When the accelerator encounters so-called *CPU-fallback operators* during a DNN inference, it raises an interrupt to fall back to the CPU. *CPU-fallback operators* [38] refer to the operators not supported by an accelerator, requiring accelerator-to-CPU fallbacks during DNN inference. Such operators are unfortunately prevalent in modern DNNs, because new operators are invented for better prediction performance, and certain operators do not exhibit parallelism. Since interrupts trigger TEE-to-REE exits, the cost of accelerator-to-CPU fallback becomes more pronounced with ASGARD's protection.

To address this problem, we propose an exit-coalescing DNN execution planning. The idea is to schedule CPU-fallback

Algorithm 1 Exit-coalescing DNN partitioning.

```
1: function PARTITION(G)
2:   ▷ G: An input DNN computation graph
3:   ExitFlag ← False
4:   Subgraphs ← ∅
5:   while G ≠ ∅ do          ▷ Construct a partition in each iteration
6:     P ← ∅                 ▷ Stores DNN operators in the current partition
7:     V ← GETNODES(G, InDegree=0, ExitFlag=ExitFlag)
8:     while V ≠ ∅ do
9:       P ← P ∪ V           ▷ Add V to the current partition
10:      G ← G − V           ▷ Remove from the graph
11:      V ← GETNODES(G, InDegree=0, ExitFlag=ExitFlag)
12:    end while
13:    if P ≠ ∅ then
14:      Subgraphs ← Subgraphs ∪ {P}
15:    end if
16:    ExitFlag ← ¬ExitFlag
17:  end while
18:  return Subgraphs
19: end function
```

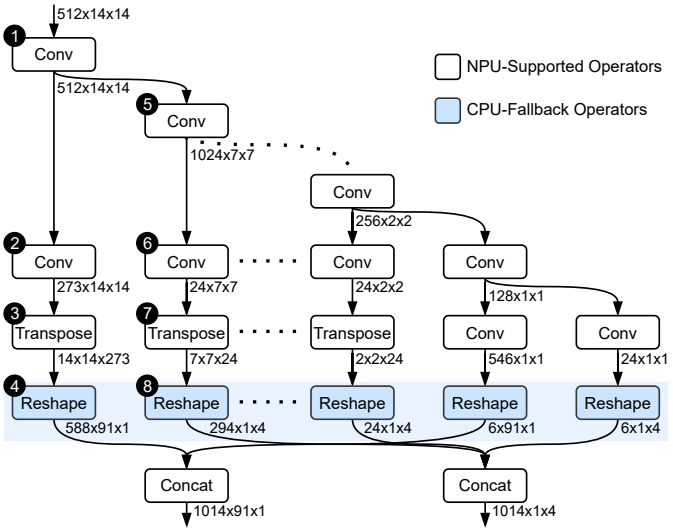


Fig. 4: A simplified computation graph of SSD-MobileNetV1, where the operators are (partially) numbered based on a depth-first operator execution planning.

operators that are not data-dependent on each other, such that they can be processed collectively in a single accelerator-to-CPU fallback, thus requiring a single accelerator interrupt. To this end, ASGARD partitions a given DNN computation graph, as formally described in Algorithm 1. This algorithm (i) takes as input a DNN computation graph, (ii) greedily constructs subgraphs, each composed exclusively of either exit-causing nodes or those not causing exits, (iii) produces an ordered list of subgraphs as output. ASGARD's exit-coalescing planning enforces this order between subgraphs during DNN execution, which effectively reduces the number of exits by necessitating exits only at a subgraph level. We illustrate the idea with an example DNN computation graph shown in Fig. 4. Executing this DNN in a depth-first order (starting from ❶ to ❽), which is the default execution order of many DNN runtimes, would trigger multiple exits (at, e.g., ❹ and ❽). Our execution planning, in contrast, puts all CPU-fallback operators in a single subgraph, requiring only a single exit.

Optimizable DNN Architectures. DNN models can benefit from our exit-coalescing optimization, when their computation graphs exhibit parallel data flows. Parallel data flows are prevalent in DNNs designed for mobile and embedded devices, as they often employ multiple feature extractors for more efficient feature extraction. In the computer vision domain, SSD models [46], for instance, have multiple feature maps, which function as parallel feature extractors. Multiple and parallel feature extractors are also found in the natural language processing domain [68], [87]. For instance, Lite Transformer [87], a Transformer [83] model designed for resource-constrained mobile applications, has parallel multi-branch feature extractors, each responsible for capturing short- and long-range attention. We show that our technique reduces many TEE-to-REE exits produced by these models in §VI-D.

Interoperability with Other Planning Strategies. ASGARD can be combined with other DNN execution planning strategies, because it only enforces a *partial* ordering between operators in a DNN computation graph. Specifically, ASGARD enforces an ordering between subgraphs produced as a result of Algorithm 1. The operators *within* each subgraph can freely be reordered per a different execution planning strategy. One can employ, for example, operator reordering strategies within each subgraph, which optimize the execution order of DNN operators for lower memory usage [4], [70], [44].

V. Implementation Details

We prototyped ASGARD on RK3588S, an Armv8.2-A SoC, equipped with a vendor-proprietary NPU (i.e., Rockchip NPU). We based our implementation of virtualization-based TEEs on Android 13's CROSVM [27], a user-mode virtual machine monitor running at EL0, and *p*KVM [23], a kernel-mode hypervisor in the Android 13's kernel (v5.10), whose REEvisor runs at EL1 and TEEvisor at EL2.

Secure I/O Passthrough. To establish secure MMIO paths between the DNN-serving enclave and the NPU, we integrated the description of the NPU in the guest device tree. The enclave uses the MMIO register addresses specified in the device tree when invoking `acquire_device()` to acquire the NPU. To configure the NPU's DMA I/O paths, we used VFIO [76], with our own modifications to interface with the IOMMU driver running at EL2. We split the existing Rockchip IOMMU driver into an unprivileged and privileged component, which we incorporated into the REEvisor and TEEvisor, respectively. We replaced the EL2 IOMMU driver's references to EL1 kernel symbols with EL2-equivalents, such as locking primitives, or provided EL2's own definition of those symbols. To configure interrupt forwarding (i.e., to deliver virtual interrupts to the NPU-assigned enclave), the TEEvisor was modified to control `ICH_LRn_EL2` list registers. Finally, to reset the Rockchip NPU during assignment switching, we had the TEEvisor assert the reset registers of the NPU, which clears all of its other registers. Additionally, by disabling the Rockchip NPU's optional use of SRAM, we eliminate the risk of inadvertently leaking models through registers nor SRAM across assignment switches. We did not reload any firmware, because the Rockchip NPU does not provide an interface for dynamic firmware loading.

NPU Bootstrapping. ASGARD employs an unmodified NPU driver in both the host and the DNN-serving guest enclaves. During the host boot process, the NPU driver on the host side bootstraps the NPU by initializing its power and clock resources, ensuring it is operational. Once bootstrapped, this host-side NPU driver becomes inactive. ASGARD employs an unmodified NPU driver inside DNN-serving enclaves as well, which controls the bootstrapped NPU while it is assigned to the enclave. However, the driver in enclaves does not control the power and clock resources of the NPU. ASGARD ensures this by providing the enclave with fake addresses for the power and clock control registers of the NPU. Any access to these registers inside the enclave therefore has no effect.

Exit-Coalescing DNN Inference. We implemented our exit-coalescing DNN execution by statically partitioning models compiled for Rockchip NPU, because the closed-source, proprietary Rockchip NN (RKNN) compiler and runtime do not support custom execution plans (i.e., operator reordering). We ran the partitioned models on our own DNN execution runtime, which executes (i) NPU-supported operators with the proprietary RKNN runtime, and (ii) CPU-fallback operators with the TensorFlow Lite runtime. During DNN inference, the physical memory shared between the host and guest enclave was used to send inference inputs and receive their outputs.

VI. Evaluation

A. Security Analysis

We conduct a security analysis of ASGARD using concrete attacks on model confidentiality. We assume a privileged, REE-side adversary (without physical tampering capabilities), following prior TrustZone-based work on protecting accelerator workloads including DNN inference [24], [59], [52], [72].

DMA Attacks. We first consider an REE adversary who controls one or more DMA-capable peripherals. ASGARD thwarts DMA attacks through such malicious peripherals, because (i) every peripheral is behind an IOMMU (see §III-B), (ii) all the IOMMUs are exclusively controlled by the TEEvisor (**INV1** in §IV-A), and (iii) the TEEvisor does not create any IOMMU page table entries that could grant malicious peripherals' access to memory pages owned by an enclave, unless it explicitly requests to do so (**INV2**).

Attacks on Sensitive Interfaces. The CPU and accelerator expose sensitive registers (see **INV1** in §IV-A) responsible for configuring the MMU and IOMMU, respectively, and both use a portion of physical memory to store page tables. An REE adversary could attempt to access these sensitive interfaces. The Arm architecture restricts the REE in N-EL0 and N-EL1 from accessing any Stage-2 control registers of the MMU, and, using Stage-2 translation, the TEEvisor prevents the adversary from accessing IOMMU registers or all MMU/IOMMU page tables residing in physical memory. Since TEEvisor owns the page frames corresponding to sensitive physical memory regions, any request to map these into the REE's address spaces is rejected. Also, as this ownership is established during secure boot, the adversary cannot fake the physical addresses of sensitive interfaces.

Attacks by Faking Reset Interface. To prevent information leakage and ensure a benign state of the accelerator at assignment (see **INV3** in §IV-A), the TEEvisor's accelerator reset should always take effect. An REE adversary could attempt to fake the reset interface to nullify the effect of the TEEvisor-invoked resets. ASGARD eliminates this threat by establishing the TEEvisor's ownership of the page frames during secure boot which the adversary cannot tamper with.

Attacks on Assignment Interface. An attacker could abuse the two hypercalls we added (see §IV-A), by invoking them with wrong arguments and at an unexpected point in time. Accelerator acquire and release requests with wrong device identifiers are rejected, and race conditions between multiple requests are prevented through mutual exclusion. These hypercalls do not introduce any attack surface to Iago attacks [19], because they are fully handled by the TEEvisor, and not passed to the REEvisor. In other words, an REE adversary cannot manipulate the result of these hypercalls invoked by enclaves.

Attacks by Raising Rogue NPU Interrupts. Since REE is responsible for interrupt management, REE adversaries can tamper with NPU interrupts. Specifically, they can (i) trigger interrupts at times not anticipated by the NPU driver, and (ii) suppress interrupts that the NPU driver expects. Their implications in DNN inference serving are: (i) prematurely executing subsequent operators with incorrect or incomplete intermediate computation results, and (ii) availability compromise, respectively. The former could have further implications with respect to model protection: If the adversaries can force the intermediate result to be the identity elements for subsequent operators (e.g., 1 for multiplication), the operators could degenerate to identity functions that simply pass their raw weight values to next operators. We believe, still, that an end-to-end model extraction attack is going to be challenging, considering the depth of DNN layers and the presence of non-invertible layers such as ReLU. Also, such a small possibility could even be eliminated by employing interrupt authenticity checks in TEEvisor similarly to Devlore [12].

Attacks by Observing NPU Interrupts. Interrupts raised by the NPU are first delivered to the REE. By passively monitoring these interrupts, an adversary on the REE side could learn that the NPU has encountered an operator it does not support. Observe that this information provides only *architectural* insights into the model, not the raw model weights. The adversary might learn further architectural details by measuring the time between two NPU interrupts, potentially revealing which operators executed in that interval. Leaking information about the model weights would require other side channels that are sensitive to the operands (e.g., model weight and bias values) of DNN operators, such as floating-point side channels [25], though such architectural side channels are beyond the scope of this work (see §III-B).

B. TCB Size Analysis

TEEvisor. For secure I/O passthrough (see §IV-A), ASGARD adds sizable components to the TEEvisor. The most significant addition is the IOMMU driver and its dependencies (see §IV-B). The original TEEvisor is 12 kLoC, and our implementation of the EL2 Rockchip IOMMU driver only introduces 2 kLoC to the TEEvisor. If the kernel's power and clock management modules were included in the TEEvisor, it could have additionally introduced 6 and 7 kLoC, respectively. Besides,

Table III: Enclave image size before and after specialization, measured in MB.

Mode	Component	Size		
		Before	After	(Reduction)
User	Core Utilities	0.831	**0.000**	**(–100.0%)**
	DNN Application	0.023	0.023	(–0.0%)
	DNN Runtime & User-Mode NPU Driver	5.610	5.610	(–0.0%)
	C/C++ Standard Libraries & Linker	4.015	**3.870**	**(–3.6%)**
	Total	10.479	**9.503**	**(–9.3%)**
Kernel	Security Features	7.000	**3.928**	**(–43.9%)**
	Application-Specific Features	6.564	**0.039**	**(–99.4%)**
	Kernel-Mode NPU Driver	0.284	0.284	(–0.0%)
	Core Kernel	3.685	3.685	(–0.0%)
	Total	17.533	**7.936**	**(–54.7%)**

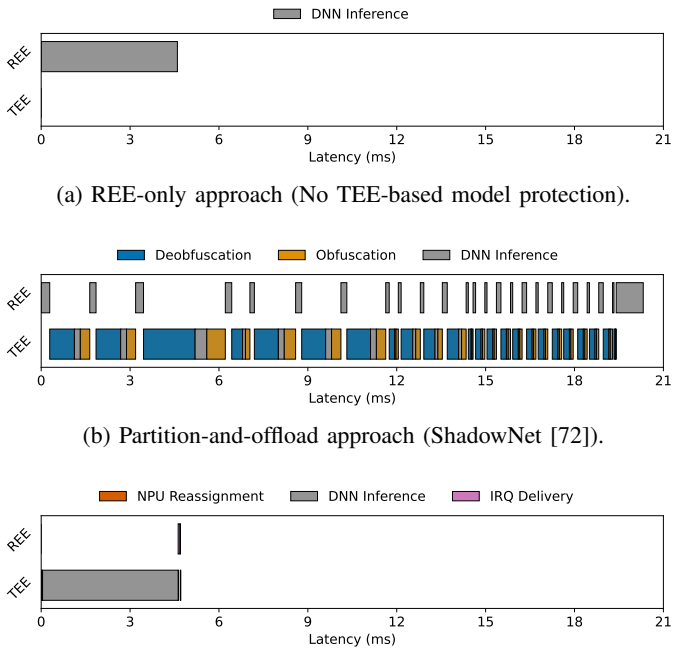
implementing secure accelerator assignment and reclamation policies (e.g., triggering accelerator resets at switching) also adds around 200 LoC. Of these, the two new guest-side hypercall handlers ASGARD introduces—which (i) configure interrupt routing paths, (ii) map and unmap NPU registers, and (iii) configure DMA paths—add less than 50 LoC each to the TEEvisor.

DNN-Serving Enclaves. Table III presents the results of our enclave debloating (see §IV-B). Starting from Google's Microdroid kernel [26], a general-purpose kernel for enclaves optimized for size, we disabled 471 out of 915 configurations, resulting in a reduction of 9.597MB from the kernel image. Of these, 422 were application-specific configurations not needed by ASGARD—including debugging, real-time processing, networking, etc. Relaxing user-kernel separation removes 1.008MB from the image. We find that enabling control flow integrity for kernel hardening adds 3.258MB to ASGARD's image; however, it does not really bloat the TCB, because it is mostly the same compiler-inserted code. In comparison, the binary size of the Trusted OS is 1.50MB for Google Pixel XL and 0.21MB for Samsung Galaxy S7 [18]. While these specialized OSs have smaller images, they are incompatible with existing accelerator runtime and drivers. As for the root file system, we trimmed 0.990MB from a minimal `buildroot` image. The image largely consists of four libraries—`libc.so`, `libm.so`, `libgcc_s.so`, and `libstdc++.so`—linked to the proprietary user-mode NPU driver.

C. End-to-End DNN Inference Latency

Experimental Setup. We ran experiments on four Cortex-A76 cores whose frequency is fixed at 2.208GHz, by disabling the other Cortex-A55 cores to offset the heterogeneity of the CPU cores. We fixed the memory controller at 1.560GHz and the NPU at 1GHz. We assigned 1 vCPU and 512MB of RAM to each guest enclave, and pinned the vCPU to a physical core to prevent migration across cores. Following prior work [85], we checked `CNTPCT_EL0`, a hardware timer, for inference latency measurements.

ASGARD vs. REE-Only & Partition-and-Offload Approaches. Fig. 5 depicts our end-to-end DNN inference latency measurements of the two baselines—an REE-only approach and a state-of-the-art *partition-and-offload* approach (i.e., ShadowNet [72])—and ASGARD for MobileNetV1 [33]. Our prototypes of ASGARD and the baselines rely on



(a) REE-only approach (No TEE-based model protection).

(b) Partition-and-offload approach (ShadowNet [72]).

(c) Secure I/O passthrough approach (ASGARD).

Fig. 5: End-to-end inference latency of MobileNetV1, under (a) an REE-only approach without any TEE-based model protection, (b) a partition-and-offload approach that simulates ShadowNet [72], and (c) ASGARD's passthrough approach. Averaged over 1,000 trials.

VSOCK [31] for communication between the TEE and REE during DNN inference, but we exclude VSOCK's overheads in our measurements. Note that this effectively makes the partition-and-offload baseline (i.e., ShadowNet) a conservative comparison, because it incurs more frequent switches between the TEE and REE than ASGARD during inference. The two baselines keep the accelerator assigned to the REE, thereby saving the accelerator assignment cost, and achieves faster NPU interrupt handling than ASGARD. Although ASGARD-protected DNN inference incurs a 2.01% overhead over the REE-only approach, it is 4.333x faster than ShadowNet, and 1.430x faster than ShadowNet without any obfuscation. This is because ShadowNet's cost of offloading the linear layers to the REE-assigned accelerator, due to costly (de)obfuscation of activations and an increased number of TEE-to-REE exits, significantly outweighs the cost saved.

NPU Assignment Cost Breakdown. We now analyze the latency of the `acquire_device()` and `release_device()` hypercalls, and depict the results in Fig. 6. In both hypercalls, the TEEvisor configures the IOMMU via an MMIO write to either set or remove the base address of the page tables used for translation. When handling the latter hypercall, the TEEvisor additionally flushes the IOTLB, which incurs marginal latency. The mapping and unmapping of NPU registers for configuring direct MMIO involves mapping and unmapping four 4KB pages from Stage-2 page tables. In the case of unmapping, there is an additional step of invalidating TLB entries for the unmapped MMIO region, but this adds negligible latency. The `release_device()` additionally includes the NPU reset

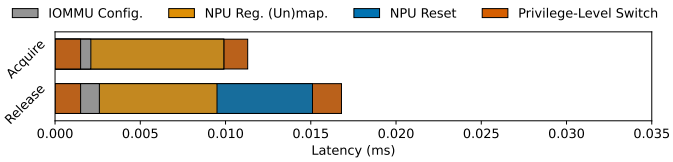


Fig. 6: Latency breakdown of ASGARD's NPU assignment, averaged over 1,000 trials.

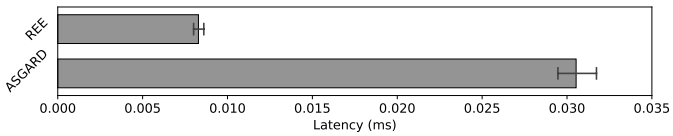


Fig. 7: Latency of delivering an NPU-generated interrupt from the EL1 host interrupt handler to the NPU driver executing either in the REE and in ASGARD's TEE, averaged over 1,000 trials with 95% confidence intervals.

latency, which is similarly negligible for the Rockchip NPU.

Interrupt Delivery Cost of Enclave-Assigned NPUs. In Fig. 7, we compare the latency of delivering an NPU-generated interrupt from the EL1 host interrupt handler to the NPU driver, with and without ASGARD's protection. As expected, this latency is significantly higher (3.684x) with ASGARD's protection due to the TEE-to-REE exits and entries involved in the delivery path (see §IV-C). The interrupt delivery from the REE to ASGARD's TEE requires 2 REE-to-TEE entries and 1 TEE-to-REE exit: (i) the host initially deactivates the physical interrupt and injects a virtual interrupt to the guest, (ii) the guest deactivates the virtual interrupt, (iii) the host accepts the deactivated virtual interrupt and unmasks (i.e., allowing the delivery of physical interrupts again) the physical interrupt, and (iv) the guest begins executing the NPU driver code.

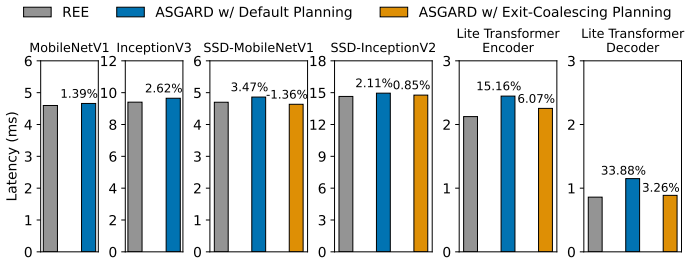
D. Run-Time Overheads During DNN Inference

Experimental Setup. We now evaluate the run-time overhead imposed by virtualization-based TEEs *during* DNN inference, with six DNN models, including those used to evaluate prior work [72], [95], [65], [52]. For image classification and object detection models, we use a 224x224 image as their input. For text understanding and generation models (i.e., Lite Transformer encoder and decoder models [87], respectively), we use 16 tokens as their input; for the decoder model, we perform a single generation step. The image classification models were retrieved from Keras Applications [1], the SSD models from TensorFlow Model Garden [92], and the Lite Transformer models from RKNN Model Zoo [2].

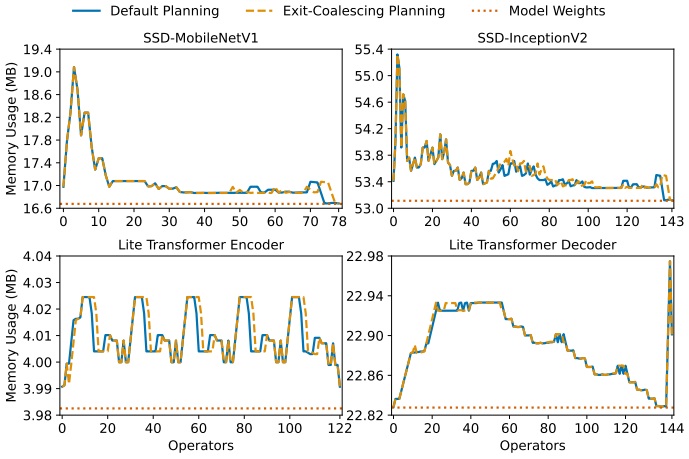
Latency and Memory Usage Impact of Exit-Coalescing Planning. Figs. 8a and 8b show the impact of ASGARD's exit-coalescing DNN execution planning on TEE-to-REE exit counts and inference latency, respectively, on six models. Of these, four models—i.e., the two SSD models [46] and the encoder and decoder models of Lite Transformer [87]—have multiple feature extractors (see §IV-C). The impact of TEE-to-REE exit overheads are more pronounced when the models require more exits during their execution due to CPU-fallback operators, or when the models exhibit lower inference latency. However, after applying our exit-coalescing planning, we remove up to 18 exits triggered when executing CPU-fallback operators (i.e., the decoder model of Lite Transformer), and this significantly reduces the TEE-to-REE fallback overheads. In addition to evaluating the latency impact, we also calculated the memory usage over time during DNN inference under the two planning strategies for the four models where exit-coalescing was effective, and depict the results in Fig. 8c. Specifically, we calculated the memory consumption by summing the sizes of the model weights and intermediate tensors (i.e., input and output activations) at each operator. The results show that applying our exit-coalescing planning alters memory usage only locally; the peak memory usage of DNN inference remains unchanged for all four models.

Planning	MobileNet V1	Inception V3	SSD-MobileNet V1	SSD-Inception V2	Lite Transformer Encoder	Lite Transformer Decoder
Default	1	10	13	18	26	38
Exit-Coalescing **(Difference)**	1 **(0)**	10 **(0)**	2 **(-11)**	10 **(-8)**	16 **(-10)**	20 **(-18)**

(a) TEE-to-REE exit counts before & after exit-coalescing planning.



(b) Inference latency of ASGARD before & after exit-coalescing planning, compared to the the REE baseline. The percentages above each bar represent the latency increase or decrease relative to the REE baseline.



(c) Memory usage before & after exit-coalescing planning.

Fig. 8: Impact of ASGARD's exit-coalescing DNN execution planning on (a) TEE-to-REE exit counts, (b) inference latency, and (c) memory usage.

VII. Discussion & Limitations

Porting ASGARD across SoCs. The main effort in porting ASGARD to another SoC involves implementing both IOMMU and reset control within TEEvisor (see §IV-A). To illustrate the process, along with ASGARD's potential TCB and run-time overheads in other SoCs, we examined Pixel 6, a production SoC featuring EdgeTPU as a DNN inference accelerator, Trusty [30] as a SW OS, and its own proprietary EL3 firmware. Implementing the IOMMU control can be done by splitting Pixel 6's own IOMMU driver into REEvisor and TEEvisor. Resetting the EdgeTPU, however, must be requested to Trusty in SW, specifically via the VIRTIO interface. While implementing this reset control within TEEvisor is feasible, it could increase the reset latency due to world switches, and TEEvisor's size due to the need to incorporate VIRTIO drivers.

Model Extraction via Side Channels. Most prior attacks that extract models through side channels primarily focus on inferring either (i) the *architectural-* or *hyper*-parameters of the model [88], [86], [47], or (ii) the *functionality* of the victim model by training it themselves using partial information obtained from side channels [61], [93], rather than the raw model weights. Side-channel attacks that recover raw weights tend to require stronger adversarial capabilities, such as physical access to the victim [34], [89]. For example, Hua et al. demonstrated that memory access patterns observed via *physical* side channels can leak the ratio between the weight and bias values of the victim model [34]. Another line of prior work has shown that interrupt capabilities could amplify existing probabilistic side channels [53], or even be used to create new ones [82]. No existing work, however, has yet exploited interrupt capabilities to exfiltrate model weights [55]. An REE adversary could delay the NPU's execution by manipulating its clock management delegated to the REE (see §IV-B), which could potentially amplify existing side channels that can expose model weights. We leave the exploration of this approach to future work.

Compatibility with Arm CCA. While ASGARD protects on-device DNNs against REE-side, privileged *software* adversaries, the same can be achieved against physical and/or stronger software adversaries with an upcoming CCA support in Armv9-A. CCA Realms can support ASGARD's entire operation described in §III-D, because (i) each Realm spans EL1 and EL2 just like a virtual machine, and (ii) CCA supports both enclave and secret data provisioning protocols. By incorporating ASGARD's secure I/O passthrough in RMM, accelerators can be securely assigned to Realms. ASGARD on CCA provides at least two security benefits. First, ASGARD can protect DNN models against physical memory probing attacks [32], because CCA encrypts memory owned by Realms using CCA's memory protection engine. Second, ASGARD can protect models against SW compromise (e.g., SW kernel compromise), because RW and SW in CCA are isolated from each other by the secure monitor in CCA's Root World.

Besides, ASGARD's exit-coalescing DNN execution planning could also be effective under CCA. (Round-trip) transitions between a TEE context and the REE context involves many operations in CCA [43], including a trap to and exception return from the EL3 secure monitor, mapping and unmapping temporary pages for transferring exit and entry information, saving and restoring the context, etc. In other words, TEE-to-REE exits become more expensive under CCA than *p*KVM. This means that, when applying ASGARD to CCA-backed TEEs, ASGARD's exit-coalescing optimization, because it reduces the number of voluntary TEE-to-REE exits, becomes more effective at reducing DNN inference latency.

Accelerator Attestation. If the accelerator peripheral has hardware root-of-trust and supports an attestation protocol, the reset operation can be more trustworthily verified through device attestation. ASGARD trusts that triggering a reset via the accelerator's reset interface in the SoC always brings back the accelerator into a known good state (see §IV-A). There is a risk, however, that the reset interface is physically compromised, which could result in (i) leakage of residual information between enclaves, or (ii) the accelerator being left in a malicious state. To establish a more direct and trustworthy confirmation of the accelerator's integrity, hardware-based attestation support can be integrated into the accelerator. For this, a hardware-isolated privileged domain (or layer [80]) serving as the root-of-trust must be integrated into the accelerator. Various attestation protocols could be used, e.g., TEE Device Interface Security Protocol, PCIe-5's Integrity and Data Encryption (in the case of PCIe devices), or more lightweight ones such as DICE [81], [48].

VIII. Related Work

Extending TEEs to Peripherals. A line of prior work has explored software-only approaches to extending TEEs to peripherals [71], [24]. LEAP [71], for instance, dedicates one or more CPU cores exclusively to hosting a TEE, and directly assigns a peripheral to it. This design offers high backward compatibility akin to ASGARD. Its requirement for *spatial* isolation of CPU cores (also known as static partitioning [49], [62]), however, limits a flexible use of compute resources in resource-constrained mobile devices. ASGARD, in contrast, supports *temporal* sharing and isolation of the REE and TEEs on the same core, using a full-fledged hypervisor split into REEvisor and TEEvisor.

StrongBox [24], another software-only approach that incorporates a peripheral (GPUs, in particular) into TEEs, separates GPU tasks into so-called secure tasks and non-secure tasks, and enforces isolation between them by using TrustZone primitives such as TZASC. Its primary advantage is that the kernel driver is excluded from the TCB. The design, however, bloats the platform-level TCB, namely the EL3 monitor, and requires substantial modifications to the kernel driver in support of secure and non-secure task abstractions. ASGARD exhibits stronger backward compatibility than StrongBox, as it does not require any change in the EL3 monitor, nor kernel driver.

There are hardware approaches as well. Arm CCA provides multiple hardware-isolated TEEs [8], and ACAI [69] and CAGE [85] extend CCA to peripheral devices. ACAI securely attaches a PCIe device by extending RMM [69]. CAGE uses a shadow-task mechanism to expose accelerators to Realms [85]. Hardware approaches on the x86 architecture offer secure I/O by requiring modifications either on the GPU side (e.g., Graviton [84] or NVIDIA H100 [56]), or on the CPU side (e.g., HIX [37] or Intel TDX [36]). In contrast to these hardware approaches, ASGARD works on legacy Armv8-A SoCs with an existing NPU hardware and drivers.

Accelerator I/O Virtualization. I/O virtualization for accelerators can be classified into direct assignment (i.e., I/O passthrough), emulation, paravirtualization, and API remoting. As full virtualization of accelerators via emulation is slow, prior work proposed various paravirtualization optimizations for GPU virtualization [77], [74]. AvA uses API remoting to more flexibly share GPUs between applications [91]. Though these approaches generally offer a higher degree of accelerator sharing than direct assignment approaches, it remains challenging yet to ensure isolation between accelerator execution contexts owned by different enclaves [39].

IX. Conclusion

DNNs are increasingly seen as valuable, often mission-critical business assets. On-device DNNs must therefore be protected as strongly possible like other traditional security-critical programs and data. We identified various efficiency and compatibility issues of prior TrustZone-based on-device model protection, and proposed the first virtualization-based TEE solution that can address them. Our design includes techniques to reduce TCB size and inference latency, making ASGARD a highly optimized and unique TEE solution for on-device DNN protection. Our evaluation shows that ASGARD can offer a trustworthy on-device environment for an end-to-end DNN protection, and that the cost of virtualization can be contained.

Acknowledgment

The authors would like to thank the anonymous shepherd and reviewers for their insightful feedback. We also extend our gratitude to Hyung-Chan An at Yonsei University for the valuable discussion on graph partitioning. This material is based upon work supported by Samsung Electronics under grant nr. IO240514-09973-01 and National Research Foundation (NRF) of Korea under grant nr. RS-2024-00334395. We also gratefully acknowledge the support provided from the Okawa Foundation through their Research Grant award.

References

[1] "Keras applications," 2024. [Online]. Available: https://keras.io/api/applications

[2] "RKNN model zoo," 2024. [Online]. Available: https://github.com/airockchip/rknn_model_zoo/blob/main/examples/lite_transformer/README.md

[3] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *Proceedings of the USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2020.

[4] B. H. Ahn, J. Lee, J. M. Lin, H.-P. Cheng, J. Hou, and H. Esmaeilzadeh, "Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices," in *Proceedings of the Conference on Machine Learning and Systems (MLSys)*, 2020.

[5] N. Amit, M. Ben-Yehuda, D. Tsafrir, A. Schuster *et al.*, "vIOMMU: Efficient IOMMU emulation," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

[6] Arm Limited, "Learn the architecture - TrustZone for AArch64 (version 1.1)," 2021. [Online]. Available: https://developer.arm.com/documentation/102418/0101

[7] ——, "Learn the architecture - AArch64 exception model (version 1.3)," 2022. [Online]. Available: https://developer.arm.com/documentation/102412/0103

[8] ——, "Introducing Arm confidential compute architecture (version 3.0)," 2023. [Online]. Available: https://developer.arm.com/documentation/den0125/0300

[9] ——, "Learn the architecture - AArch64 virtualization guide (version 1.0)," 2024. [Online]. Available: https://developer.arm.com/documentation/102142/0100

[10] A. Athalye, A. Belay, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Notary: A device for secure transaction approval," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[11] E. Auger, "vIOMMU/ARM: full emulation and virtio-iommu approaches," *KVM Forum*, 2017.

[12] A. Bertschi, S. Sridhara, F. Groschupp, M. Kuhne, B. Schlüter, C. Thorens, N. Dutly, S. Capkun, and S. Shinde, "Devlore: Extending Arm CCA to integrated devices," *arXiv preprint arXiv:2408.05835*, 2024.

[13] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "MI6: Secure enclaves in a speculative out-of-order processor," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[14] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with user-space enclaves," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[15] J.-P. Brucker, "KVM: Arm SMMUv3 driver for pKVM," 2023. [Online]. Available: https://lwn.net/Articles/921869

[16] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson, A. Oprea, and C. Raffel, "Extracting training data from large language models," in *Proceedings of the USENIX Security Symposium (Security)*, 2021.

[17] N. Carlini, J. Hayes, M. Nasr, M. Jagielski, V. Sehwag, F. Tramer, B. Balle, D. Ippolito, and E. Wallace, "Extracting training data from diffusion models," in *Proceedings of the USENIX Security Symposium (Security)*, 2023.

[18] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "SoK: Understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems," in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.

[19] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[20] J. Choi, J. Kim, C. Lim, S. Lee, J. Lee, D. Song, and Y. Kim, "GuardiaNN: Fast and secure on-device inference in TrustZone using embedded SRAM and cryptographic hardware," in *Proceedings of the ACM/IFIP International Middleware Conference (Middleware)*, 2022.

[21] C. Dall, S.-W. Li, J. T. Lim, J. Nieh, and G. Koloventzos, "ARM virtualization: Performance and architectural implications," in *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[22] C. Dall and J. Nieh, "KVM/ARM: The design and implementation of the Linux ARM hypervisor," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[23] W. Deacon, "Virtualization for the masses: Exposing KVM on Android," *KVM Forum*, 2020.

[24] Y. Deng, C. Wang, S. Yu, S. Liu, Z. Ning, K. Leach, J. Li, S. Yan, Z. He, J. Cao, and F. Zhang, "StrongBox: A GPU TEE on Arm endpoints," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2022.

[25] C. Gongye, Y. Fei, and T. Wahl, "Reverse-engineering deep neural networks using floating-point timing side-channels," in *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2020.

[26] Google, "Android virtualization framework (AVF) overview," 2024. [Online]. Available: https://source.android.com/docs/core/virtualization

[27] ——, "crosvm - the ChromeOS virtual machine monitor," 2024. [Online]. Available: https://android.googlesource.com/platform/external/crosvm

[28] ——, "Implementing dm-verity," 2024. [Online]. Available: https://source.android.com/docs/security/features/verifiedboot/dm-verity

[29] ——, "Security-enhanced Linux in Android," 2024. [Online]. Available: https://source.android.com/docs/security/features/selinux

[30] ——, "Verified boot," 2024. [Online]. Available: https://source.android.com/docs/security/features/trusty

[31] S. Hajnoczi, "virtio-vsock: Zero-configuration host/guest communication," *KVM Forum*, 2015.

[32] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *Proceedings of the USENIX Security Symposium (Security)*, 2008.

[33] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[34] W. Hua, Z. Zhang, and G. E. Suh, "Reverse engineering convolutional neural networks through side-channel information leaks," in *Proceedings of the Annual Design Automation Conference (DAC)*, 2018, pp. 1–6.

[35] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM TrustZone," in *Proceedings of the USENIX Security Symposium (Security)*, 2017.

[36] Intel, "Intel Trust Domain Extensions," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html

[37] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity GPUs," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[38] J. S. Jeong, J. Lee, D. Kim, C. Jeon, C. Jeong, Y. Lee, and B.-G. Chun, "Band: Coordinated multi-DNN inference on heterogeneous mobile processors," in *Proceedings of the Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2022.

[39] H. Lefeuvre, D. Chisnall, M. Kogias, and P. Olivier, "Towards (really) safe and fast confidential I/O," in *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2023.

[40] D. Li, Z. Mi, Y. Xia, B. Zang, H. Chen, and H. Guan, "TwinVisor: Hardware-isolated confidential virtual machines for ARM," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[41] S.-W. Li, J. S. Koh, and J. Nieh, "Protecting cloud virtual machines from hypervisor and host operating system exploits," in *Proceedings of the USENIX Security Symposium (Security)*, 2019.

[42] W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang, "TEEv: Virtualizing trusted execution environments on mobile platforms," in *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2019.

[43] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell, "Design and verification of the Arm confidential compute architecture," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[44] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, "On-device training under 256KB memory," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[45] R. Liu, L. Garcia, Z. Liu, B. Ou, and M. Srivastava, "SecDeep: Secure and performant on-device deep learning inference framework for mobile and IoT devices," in *Proceedings of the International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 2021.

[46] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[47] Z. Liu, Y. Yuan, Y. Chen, S. Hu, T. Li, and S. Wang, "DeepCache: Revisiting cache side-channel attacks in deep neural networks executables," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2024.

[48] A. Marochko, D. Mattoon, P. England, R. Aigner, R. S. (CELA), and S. Thom, "Cyber-resilient platforms overview," Microsoft, Tech. Rep. MSR-TR-2017-40, 2017.

[49] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems," in *Workshop on next generation real-time embedded systems (NG-RES)*, 2020.

[50] S. Mirzamohammadi and A. A. Sani, "The case for a virtualization-based trusted execution environment in mobile devices," in *Proceedings of the Asia-Pacific Workshop on Systems (APSys)*, 2018.

[51] F. Mo, H. Haddadi, K. Katevas, E. Marin, D. Perino, and N. Kourtellis, "PPFL: Privacy-preserving federated learning with trusted execution environments," in *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2021.

[52] F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi, "DarkneTZ: Towards model privacy at the edge using trusted execution environments," in *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2020.

[53] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX amplifies the power of cache attacks," in *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017.

[54] M. Nasr, R. Shokri, and A. Houmansadr, "Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning," in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019.

[55] T. Nayan, Q. Guo, M. Al Duniawi, M. Botacin, S. Uluagac, and R. Sun, "SoK: All you need to know about on-device ML model extraction - the gap between research and practice," in *Proceedings of the USENIX Security Symposium (Security)*, 2024.

[56] NVIDIA, "Confidential Compute on NVIDIA Hopper H100," 2023. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf

[57] OP-TEE core maintainers, "OP-TEE trusted OS," 2024. [Online]. Available: https://github.com/OP-TEE/optee_os

[58] Open Virtualization Alliance, "Linux kernel virtual machine." [Online]. Available: https://www.linux-kvm.org

[59] H. Park and F. X. Lin, "GPUReplay: A 50-kb GPU stack for client ML," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[60] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[61] A. S. Rakin, M. H. I. Chowdhuryy, F. Yao, and D. Fan, "DeepSteal: Advanced model extractions leveraging efficient weight stealing in memories," in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2022.

[62] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no VM exits! (almost)," in *Proceedings of the Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2017.

[63] P. Ren, C. Zuo, X. Liu, W. Diao, Q. Zhao, and S. Guo, "DEMISTIFY: Identifying on-device machine learning models stealing and reuse vulnerabilities in mobile apps," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2024.

[64] A. A. Sani, L. Zhong, and D. S. Wallach, "Glider: A GPU library driver for improved system security," Rice University, Tech. Rep. 2014-11-14, 2014.

[65] T. Shen, J. Qi, J. Jiang, X. Wang, S. Wen, X. Chen, S. Zhao, S. Wang, L. Chen, X. Luo, F. Zhang, and H. Cui, "SOTER: Guarding black-box inference for general neural networks at the edge," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2022.

[66] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2017.

[67] S. Siby, S. Abdollahi, M. Maheri, M. Kogias, and H. Haddadi, "GuaranTEE: Towards attestable and private ML with CCA," in *Proceedings of the Workshop on Machine Learning and Systems (EuroMLSys)*, 2024.

[68] D. So, Q. Le, and C. Liang, "The evolved Transformer," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2019.

[69] S. Sridhara, A. Bertschi, B. Schlüter, M. Kuhne, F. Aliberti, and S. Shinde, "ACAI: Protecting accelerator execution with Arm confidential computing architecture," in *Proceedings of the USENIX Security Symposium (Security)*, 2024.

[70] B. Steiner, M. Elhoushi, J. Kahn, and J. Hegarty, "MODeL: Memory

optimizations for deep learning," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.

[71] L. Sun, S. Wang, H. Wu, Y. Gong, F. Xu, Y. Liu, H. Han, and S. Zhong, "LEAP: TrustZone based developer-friendly TEE for intelligent mobile apps," *IEEE Transactions on Mobile Computing*, vol. 22, no. 12, pp. 7138–7155, 2023.

[72] Z. Sun, R. Sun, C. Liu, A. R. Chowdhury, L. Lu, and S. Jha, "ShadowNet: A secure and efficient on-device model inference system for convolutional neural networks," in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2023.

[73] Z. Sun, R. Sun, L. Lu, and A. Mislove, "Mind your weight(s): A large-scale study on insufficient machine learning model protection in mobile apps," in *Proceedings of the USENIX Security Symposium (Security)*, 2021.

[74] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "GPUvm: Why not virtualizing GPUs at the hypervisor?" in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2014.

[75] The kernel development community, "Seccomp BPF (secure computing with filters)," 2019. [Online]. Available: https://www.kernel.org/doc/html/v5.10/userspace-api/seccomp_filter.html

[76] ——, "VFIO - "virtual function I/O"," 2019. [Online]. Available: https://www.kernel.org/doc/html/v5.10/driver-api/vfio.html

[77] K. Tian, Y. Dong, and D. Cowperthwaite, "A full GPU virtualization solution with mediated pass-through," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2014.

[78] K. Tian, Y. Zhang, L. Kang, Y. Zhao, and Y. Dong, "coIOMMU: A virtual IOMMU with cooperative DMA buffer tracking for efficient memory management in direct I/O," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020.

[79] F. Tramer and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[80] Trusted Computing Group, "DICE layering architecture," 2020. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/DICE-Layering-Architecture-r19_pub.pdf

[81] ——, "DICE attestation architecture," 2021. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/DICE-Attestation-Architecture-r23-final.pdf

[82] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[83] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[84] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on GPUs," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[85] C. Wang, F. Zhang, Y. Deng, K. Leach, J. Cao, Z. Ning, S. Yan, and Z. He, "CAGE: Complementing Arm CCA with GPU extensions," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2024.

[86] J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. Al Faruque, "Leaky DNN: Stealing deep-learning model secret with GPU context-switching side-channel," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020.

[87] Z. Wu, Z. Liu, J. Lin, Y. Lin, and S. Han, "Lite Transformer with long-short range attention," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[88] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures," in *Proceedings of the USENIX Security Symposium (Security)*, 2020.

[89] D. Yang, P. J. Nair, and M. Lis, "Huffduff: Stealing pruned DNNs from sparse accelerators," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[90] Z. Yao, S. M. Seyed Talebi, M. Chen, A. Amiri Sani, and T. Anderson, "Minimizing a smartphone's TCB for security-critical programs with exclusively-used, physically-isolated, statically-partitioned hardware," in *Proceedings of the Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2023.

[91] H. Yu, A. M. Peters, A. Akshintala, and C. J. Rossbach, "AvA: Accelerated virtualization of accelerators," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[92] H. Yu, C. Chen, X. Du, Y. Li, A. Rashwan, L. Hou, P. Jin, F. Yang, F. Liu, J. Kim, and J. Li, "TensorFlow model garden," 2024. [Online]. Available: https://github.com/tensorflow/models

[93] Y. Yuan, Z. Liu, S. Deng, Y. Chen, S. Wang, Y. Zhang, and Z. Su, "HyperTheft: Thieving model weights from TEE-shielded neural networks via ciphertext side channels," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2024.

[94] Y. Zhang, Y. Hu, Z. Ning, F. Zhang, X. Luo, H. Huang, S. Yan, and Z. He, "SHELTER: Extending Arm CCA with isolation in user space," in *Proceedings of the USENIX Security Symposium (Security)*, 2023.

[95] Z. Zhang, C. Gong, Y. Cai, Y. Yuan, B. Liu, D. Li, Y. Guo, and X. Chen, "No privacy left outside: On the (in-)security of TEE-shielded DNN partition for on-device ML," in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2024.

APPENDIX A
ARTIFACT APPENDIX

ASGARD is a new on-device deep neural network (DNN) model protection solution based on a virtualization-based trusted execution environment. While the virtual machine abstraction used by ASGARD brings the benefit of strong compatibility with existing proprietary software including NPU drivers and secure monitors at EL3, it introduces both trust computing base (TCB) and run-time overheads. ASGARD aggressively minimizes both platform- and application-level TCB overheads, and reduces run-time overheads through our proposed DNN execution planning technique. Our evaluation includes (i) a qualitative yet comprehensive security analysis of ASGARD, and (ii) a quantitative analysis of ASGARD's TCB and run-time overheads on our prototype implementation of ASGARD on RK3588S. We outline below how to build our prototype, and how to reproduce our quantitative analysis results of TCB and run-time overheads.

A. Description & Requirements

1) How to access: We make the artifact available on GitHub[4], with all the main components and source code included as Git submodules. For intellectual property reasons, we were unable to upload the artifact to permanent storage by the artifact evaluation completion deadline.

2) Hardware dependencies: Building ASGARD requires compiling Android, which is a resource-intensive task. It requires a 64-bit x86 host machine with at least 1TB of free disk space, 64GB of RAM, and preferably a multi-core CPU. Additionally, a Khadas Edge2 development board with at least 16GB of RAM is required.

3) Software dependencies: The x86 host machine should have Ubuntu 20.04 LTS installed.

4) Benchmarks: Our experiments use a set of benchmarks that we specifically developed. We provide all the DNN models required to run the benchmarks.

[4]Artifact available at: https://github.com/yonsei-sslab/asgard

B. Artifact Installation & Configuration

1) Downloading Sources: The repository must first be cloned with the `--recurse-submodules` option to download all submodules along with the main repository. Due to the large size of the Android source, we include submodules named `asgard-manifest-*` that point to all necessary repositories for the host Android, host kernel, enclave kernel, and CROSVM. These can be downloaded using the `repo` command specified in the `README.md` file.

2) Building Sources: The host Android, host kernel, enclave kernel, CROSVM, and DNN applications must be compiled on the host machine by following the instructions in the `README.md` file. This process should take approximately 20 human-minutes and 85 compute-minutes on a machine with Intel i9-12900K CPU and 64GB of RAM.

3) Installation: First, we install the new host image on the development board by following the instructions in the `README.md` file. Next, we use Buildroot to create the enclave root file system image. We add DNN models, DNN applications, and the user-mode NPU driver to the image, and remove unnecessary binaries from the image. Finally, we transfer all the necessary files to the development board. This process should take approximately 27 human-minutes and 37 compute-minutes.

C. Major Claims

- (C1): ASGARD maintains high compatibility with existing accelerator drivers and secure monitors. This is proven by the experiments (E1) and (E2).
- (C2): ASGARD keeps the platform- and application-level TCB increase minimal (e.g., 2 kLoC introduced to TEEvisor, 17.439MB for the enclave image). This is proven by the experiments (E3) and (E4), whose results are reported in §VI-B and Table III.
- (C3): ASGARD achieves near-zero DNN inference latency, which is significantly lower than that of existing approaches that offload part of DNN inference to the rich execution environment (REE) (e.g., 4.333x faster than ShadowNet). This is proven by the experiments (E5) and (E6), whose results are reported in Figs. 5b, 5c and 8b.

D. Evaluation

The artifact contains a total of six experiments. The first four experiments verify the source code and proprietary binaries used in the artifact, all of which can be executed on the host machine. The last two experiments involve running DNN inference on the development board. Specific instructions for running the experiments are provided in the `README.md` file due to space constraints. The experiments should take approximately 105 human-minutes and 20 compute-minutes.

1) Experiment (E1): [Verify Secure Monitor and User-Mode NPU Driver] [10 human-minutes]: ASGARD does not require any modifications to the closed-source secure monitor or the user-mode NPU driver. This experiment verifies that these components will be used throughout the evaluation.

[Preparation] The `bin` directory contains the proprietary secure monitor `rk3588_bl31_v1.26.elf`[5] and user-mode RKNPU driver `librknnrt.so`[6] from the device vendor. On the host machine, go to the main repository.

[Execution] To verify that we are running the unmodified proprietary secure monitor: First, check that the bootloader is configured to load the proprietary secure monitor. Second, use the `cmp` command to perform a byte-to-byte comparison between the binary file running on the development board and the file located in the `bin` directory.

Furthermore, to verify that we are running the unmodified proprietary user-mode NPU driver, we will run protected DNN inference during the experiment (E5) using the proprietary driver, which was embedded into the root file system in Appendix §A-B.

[Results] For the secure monitor, the `cmp` command should not produce any output if no differences are found; for the user-mode NPU driver, the experiments (E5) and (E6) must be completed successfully.

2) Experiment (E2): [Verify Kernel-Mode NPU Driver] [10 human-minutes]: ASGARD uses an unmodified kernel-mode NPU driver in both the REE and the enclave. This experiment verifies that the unmodified driver will be used throughout the evaluation.

[Preparation] The `src/rknpu` directory contains the original kernel-mode NPU driver code obtained from the device vendor[7]. On the host machine, go to the main repository.

[Execution] Use the `diff` command to verify the drivers running in both the REE and the enclave.

[Results] The `diff` command should not produce any output if no differences are found. However, in this artifact, we have included our custom performance measurement framework that will be used in the experiments (E5) and (E6). This framework does not introduce any functional changes to the driver (i.e., affect the driver's original behavior). The framework comprises: (i) a component that measures the NPU task completion time using a hardware timer, and (ii) input/output control (IOCTL) call handlers and header definitions to acquire and clear the task completion time.

3) Experiment (E3): [Verify TEEvisor TCB Size] [5 human-minutes]: ASGARD adds 2 kLoC to the TEEvisor (see §VI-B). This experiment measures the LoC of the original, unmodified TEEvisor (i.e., the *p*KVM hypervisor[8]) and ASGARD's TEEvisor.

[Preparation] On the host machine, go to the main repository.

[Execution] Use the `cloc` command to measure the LoC changes.

[5] https://github.com/rockchip-linux/rkbin/blob/ae710c9/bin/rk35/rk3588_bl31_v1.26.elf

[6] https://github.com/rockchip-linux/rknn-toolkit2/blob/1f4415e/rknpu2/runtime/Linux/librknn_api/aarch64/librknnrt.so

[7] https://github.com/khadas/linux/tree/973dd55/drivers/rknpu

[8] https://android.googlesource.com/kernel/common/+/refs/heads/deprecated/android13-5.10-2022-11/arch/arm64/kvm/hyp

[Results] The value in the `code` column and the `SUM` row represents the total LoC for the TEEvisor. Subtract the original TEEvisor's value from ASGARD's value, which should be about 2 kLoC.

4) Experiment (E4): [Verify Enclave Image Size] [10 human-minutes]: ASGARD's enclave image is 17.439 MB (see Table III), which includes the kernel and the root file system. This experiment involves measuring the size of the kernel and file system images.

[Preparation] On the host machine, go to the main repository.

[Execution] Run the `size` command to measure the kernel image size, and run `make graph-size` in Buildroot to measure the root file system image size.

[Results] For the kernel image, the `size` command should output 7.936 MB in the `dec` column. For the root file system image, the output is produced at `output/graphs/file-size-stats.csv`. The files that we removed from the image in Appendix §A-B should not be counted. Run our Python script `get_rootfs_size.py`, which adds values in the `File size` column for the selected rows. The output should be 9.503 MB.

5) Experiment (E5): [Compare Inference Latency with REE] [60 human-minutes + 10 compute-minutes]: ASGARD achieves near-zero DNN inference latency overhead compared to that in the REE (see Fig. 8b). This experiment involves running unprotected DNN inference in the REE and protected inference within the ASGARD enclave, using all six DNN models.

[Preparation] Access the development board using `adb`. Running inference in the REE and ASGARD-protected environment requires loading different versions of the Rockchip IOMMU driver.

[Execution] Use the commands provided in the `README.md` file to run inference in the REE and in the ASGARD-protected environment. For SSD and Lite Transformer models, we apply exit-coalescing DNN execution planning for ASGARD.

[Results] Compared to the REE, the ASGARD-protected inference should exhibit the latency value shown in Fig. 8b.

6) Experiment (E6): [Compare Inference Latency with ShadowNet] [10 human-minutes + 10 compute-minutes]: ASGARD achieves DNN inference latency overhead that is significantly lower than that of existing REE-offloading approaches (see Figs. 5b and 5c). This experiment involves running DNN inference simulating ShadowNet and inference within the ASGARD enclave, using MobileNetV1.

[Preparation] Access the development board using `adb`. Running ShadowNet-simulated inference and ASGARD-protected inference requires loading different versions of the Rockchip IOMMU driver.

[Execution] Use the commands provided in the `README.md` file to run ShadowNet-simulated and ASGARD-protected inference.

[Results] The inference latency should match the values shown in Figs. 5b and 5c.