# An Early Experience with Confidential Computing Architecture for On-Device Model Protection

Sina Abdollahi*, Mohammad Maheri†, Sandra Siby‡, Marios Kogias§ and Hamed Haddadi¶
*Imperial College London, s.abdollahi22@imperial.ac.uk
†Imperial College London, m.maheri23@imperial.ac.uk
‡New York University Abu Dhabi, sandra.siby@nyu.edu
§Imperial College London, m.kogias@imperial.ac.uk
¶Imperial College London, h.haddadi@imperial.ac.uk

***Abstract*—Deploying machine learning (ML) models on user devices can improve privacy (by keeping data local) and reduce inference latency. Trusted Execution Environments (TEEs) are a practical solution for protecting proprietary models, yet existing TEE solutions have architectural constraints that hinder on-device model deployment. Arm Confidential Computing Architecture (CCA), a new Arm extension, addresses several of these limitations and shows promise as a secure platform for on-device ML. In this paper, we evaluate the performance–privacy trade-offs of deploying models within CCA, highlighting its potential to enable confidential and efficient ML applications. Our evaluations show that CCA can achieve an overhead of, at most, 22% in running models of different sizes and applications, including image classification, voice recognition, and chat assistants. This performance overhead comes with privacy benefits, for example, our framework can successfully protect the model against membership inference attack by 8.3% reduction in the adversary's success rate. To support further research and early adoption, we make our code and methodology publicly available.**

## 1. Introduction

Machine-learning (ML) models are increasingly being deployed on edge devices for various purposes such as health monitoring, anomaly detection, face recognition, voice assistants *etc*. Running models locally can provide low-latency services to users without the need for sending data to an external entity. As end-users typically lack the resources required to train a model, they prefer to utilize a pre-trained, reliable model owned by a third party for inference and, potentially, personalization. The model owner, having invested significant resources in training the model, requires robust security assurances to safeguard the model's integrity and usage. Without these guarantees, the owner may not be willing to deploy the model on end devices.

Various solutions have been proposed for model protection on the edge. Cryptographic techniques such as homomorphic encryption (HE) [1]–[3] or secure multiparty communication (SMC) [4], [5] are hindered by computational and communication overheads, while the use of trusted execution environments (TEEs) is considered a more efficient approach. A TEE is an environment that uses hardware-enforced mechanisms to protect memory and execution from the operating system (OS) and its application layer (collectively known as the Rich Execution Environment, or REE [6]). Deploying models in a TEE mitigates privacy-stealing attacks from REE-based adversaries: even if the REE is compromised, the adversary is limited to black-box access to the model, whereas models deployed in the REE are exposed to white-box attacks.

On the other hand, using TEEs on end devices face security and functionality challenges. While Intel Software Guard Extensions (SGX) has been deprecated on end-user devices [7], Arm's TEE—commonly known as TrustZone—remains a widely adopted on-device solution, implemented in various mobile platforms (e.g., Qualcomm, Trustonic). However, as Cerdeira *et al.* [8] showed, TrustZone has been the target of high-impact attacks due to its security vulnerabilities. The high privilege level of TrustZone, has led vendors to impose functional restrictions in an effort to reduce the attack surface, restrictions such as lack of support for GPU accessing [9], and small memory size (32MB for OP-TEE) [10].

To overcome these limitations, several solutions have proposed partitioning models and executing only the more sensitive components within TEEs [11], [12]. These approaches aim to provide near black-box security without placing the entire model inside the TEE. However, Zhang *et al.* [13] demonstrate that such solutions remain vulnerable to privacy-stealing attacks and are not as secure as commonly assumed. Even partial model weights can leak private information about the training dataset, particularly when combined with publicly available resources (e.g., similar datasets or pre-trained models). Therefore, deploying the entire model within the TEE boundary remains the most effective strategy for protecting it against privacy attacks.

Arm Confidential Compute Architecture (CCA) [14] is a key component of the Armv9-A architecture that is expected to be available on Arm devices. Arm CCA allows the creation of special virtual machines called *realm*, orthogonal to the already existing TrustZone. Realm is de-privileged as it has virtualized access to the resources, and it is TEE because it has protection against REE actors. Realm creation and runtime are supported by hardware-backed attestation services which can provide enough evidence for a relying party (*e.g.*, model provider) about the trustworthiness of the realm. Compared to TrustZone, CCA benefits from a more flexible memory allocation

TABLE 1: Memory access rules applied by granule protection check (GPC)

| Security State | Normal PAS | Secure PAS | Realm PAS | Root PAS |
|---|---|---|---|---|
| Normal | Yes | No | No | No |
| Secure | Yes | Yes | No | No |
| Realm | Yes | No | Yes | No |
| Root | Yes | Yes | Yes | Yes |

scheme.[1] The CCA features seem promising for on-device model deployment. Given that Arm is the dominant architecture in mobile and edge devices, we anticipate CCA's widespread deployment in the near future.

**Motivation.** Inspired by (1) the limitations of existing TEE solutions, (2) vulnerabilities in current model partitioning strategies, and (3) the key features of Arm CCA, this work introduces and evaluates a framework for deploying on-device models within Arm CCA. We use the latest tools and plugins provided by Arm to simulate and trace Arm CCA behavior in running ML workloads. We do not employ partitioning strategies, ensuring that the entire model remains protected from REE actors. Our findings and evaluation results can be useful to support further research and early adoption, prior to the widespread adoption of CCA on end devices.

Our contributions are as follows:

- We define a basic framework for on-device model deployment within Arm CCA and use the latest tools, software, and firmware to simulate the framework.
- We evaluate the framework for models of different sizes and applications, all showing acceptable overhead (22% in the worst case).
- To showcase the security gain of the framework, we implement a membership inference attack on the models, showing that running models within a realm, on average, provides an 8.3% decrease in the success rate of membership inference attacks against the training dataset.
- We make all our code and framework openly available and will maintain it to benefit early-stage adoption of CCA software products[2].

## 2. Background

In this section, we first provide a brief overview of Arm CCA (Section 2.1) and why it comes with overhead (Section 2.2). We also discuss the possible choices of evaluating CCA (Section 2.3). Finally, in Section 2.4, we introduce a privacy-stealing attack commonly used in security evaluations of ML systems.

### 2.1. Arm Confidential Compute Architecture

Arm CCA [14] is a series of hardware and software architecture extensions that enhances Armv9-A support for confidential computing. As shown in Figure 1, Arm CCA, introduces four worlds (a.k.a, execution environment): root, realm, secure, and normal world. To enforce isolation between the worlds, CCA introduces a mechanism called granule protection check (GPC). Any memory access request succeeds only if the requester state (e.g., processor state) and the memory's state both comply with the rules defined in Table 1. Particularly, the root world state can access the physical address space (PAS) of all the other worlds while the realm and secure worlds state have access to the normal PAS, but they cannot access each other's PAS. The normal world (NW) state cannot access the PAS of the other worlds. Arm architecture allows different exception levels (EL) to exist, from EL3 (the highest privilege) to EL0 (the lowest privilege).

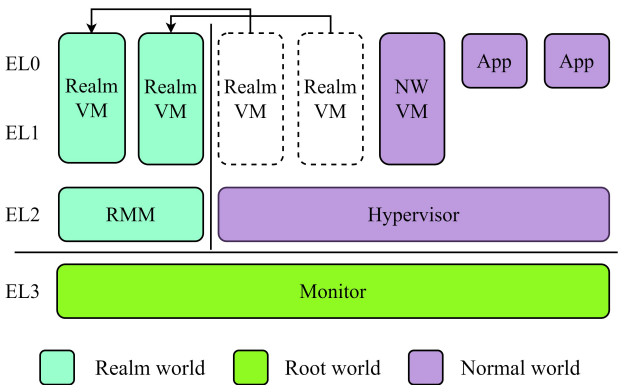


Figure 1: Arm CCA software architecture. The hypervisor allocates resources to realms but cannot access those resources, due to isolation boundaries between the realm and the normal world

**Software architecture.** Figure 1 shows the software architecture of Arm CCA. The Monitor is the highest privileged firmware in the system responsible for initially booting all EL2 firmware/software, managing the GPC, and context switching between different worlds. The normal world stack consists of a hypervisor operating at NW-EL2, virtual machines running at EL1 and EL0 and user-space apps running at EL0. The hypervisor is responsible for managing all resources (*e.g.*, CPU and memory) in the system. The realm world stack consists of a lightweight firmware known as Realm Management Monitor (RMM) which mediates resource allocation of realm VMs, and realm VMs (or simply realms) running at EL1 and EL0. RMM enforces isolation boundaries between the hypervisor and the realm VMs, making realm resources (*e.g.*, memory pages of the realm VM) inaccessible for the hypervisor. The RMM is also able to generate an *attestation report* for the realm VM. This report keeps necessary information about the initial content of the realm as well as the firmware (RMM and Monitor) in the system [15], [16].

### 2.2. Realm Overhead

Handling exceptions[3] is more complex for a realm VM, compared to a normal world VM. For a normal world

1. TrustZone enforces isolation using an Address Space Controller (TZASC) and bus-level protection, requiring coarse-grained changes to memory regions. In contrast, CCA uses standard page tables and the Memory Management Unit (MMU) to enforce isolation, allowing for fine-grained and dynamic memory management.

2. https://github.com/comet-cc/CCA-Evaluation

3. In Arm architecture, exceptions are conditions or system events that require action by privileged software [17]. Notably, interrupts triggered by virtual devices (*e.g.*, virtual timer) are a type of exception.

VM, every exception is directly received and handled by the hypervisor, while for a realm VM, the RMM is initially responsible for handling the exception and, if necessary, forwarding it to the hypervisor. This complexity increases the overhead of running a workload within realm. Two notable sources of exceptions are the hypervisor's timer interrupt (timer at NW-EL2) and the realm's timer interrupts (timer at Realm-EL1), both necessary for process scheduling within the two kernels. Each time these timers are acknowledged by the processor, an exit from the realm occurs, which requires handling by the hypervisor. In Section 4.2 and Appendix B we compare the runtime execution and I/O operation between a realm and a NW VM.

### 2.3. CCA Evaluation Platforms

At the time of writing, there is no hardware compatible with the CCA specification. However, there are software that emulates the behavior of a CCA-compatible device. Linaro's QEMU [18] can be used to boot and run the CCA software stack [19]. Fixed Virtual Platform (FVP) is the official software released by Arm, compatible with the CCA specification [20], [21]. FVP provides useful plugins and tools which, combined, provide detailed information about the behavior of CCA. Devlore [22] used QEMU, but other works have used FVP in their evaluation as functional prototype [23], [24] and also performance prototype [25]–[27]. We utilize FVP for the evaluation. In Appendix A, we describe how FVP can be set up with plugins and tracing tools to measure realm's behavior. Furthermore, we explain the accuracy of FVP and other possible options to evaluate CCA. It is important to note that neither FVP nor QEMU is designed to provide performance predictions, and any evaluation based on these tools should be regarded as preliminary and approximate.

### 2.4. Membership Inference Attack

Membership inference attacks (MIA) are a class of attacks in which an adversary tries to determine whether a particular data point was a part of the training set or not. These attacks have been widely used in the literature to assess how much a system "leaks" information about the training dataset [11]–[13]. In a typical attack setting [13], [28], an adversary has access to a *shadow dataset* which is statistically similar to the target model's dataset. This dataset is then used to train a *shadow model* and an *attack model* (a binary classifier). Finally, to determine whether a data sample is a member of the target model's training dataset, the sample is fed to the target model, and the posteriors and the predicted label (transformed to a binary indicator on whether the prediction is correct) are fed to the attack model. Moreover, an adversary with white-box access can enhance the attack's accuracy by leveraging additional model information, such as classification loss and sample gradients (see [13], [28] for details). We use this attack in Section 4.3 to show the privacy protection of our framework.

## 3. Framework Architecture

In this section, we describe a basic framework to deploy on-device models within CCA.

### 3.1. System Model

As illustrated in Figure 2, the system involves three parties: *model providers*, *clients*, and a *trusted verifier*. A model provider is an entity responsible for training and deploying ML models on end-devices for various tasks. These models, along with their training datasets, are considered intellectual property and must be protected from unauthorized access by malicious users and other model providers. The client is an end-device, such as a smartphone or an IoT gateway, which supports Arm CCA. Clients host a wide variety of applications within their REE that may require machine learning services, such as facial recognition, voice detection, or chat assistants. The trusted verifier is responsible for providing realm images. A realm image includes a complete stack for a virtual machine, encompassing an operating system, user-space libraries, and applications necessary for running the model within the realm.

### 3.2. Threat Model

We assume that model providers and clients are two mutually distrusting entities, but they both trust the images offered by the trusted verifier. Clients may attempt to maliciously extract information about the model's weights and training data. Both the Monitor and the RMM are considered trustworthy due to their small codebase, and formal verification in the case of the RMM [29], [30]. However, the NW stack is untrusted as it is large and complex, containing unverified user-space applications, third-party libraries, and drivers. An adversary could exploit these vulnerabilities to compromise the entire NW. Arm CCA, by default, does not provide availability guarantees regarding runtime execution and memory of realm. However, we assume that the hypervisor allocates sufficient CPU time and memory to the realm, allowing it to effectively load the model and perform inference[4]. Physical and side-channel attacks are also significant threats to the deployment of the device model [31], [32]. However, there are considered out of scope and the hardware is trusted.

### 3.3. Model Deployment Pipeline

Figure 2 shows an overview of our framework. In the following, we provide a description of the various steps involved in deploying the model within a realm.

**Realm setup.** A NW app starts the process by obtaining a publicly-available and verified realm image from the trusted verifier (Step 1 in Figure 2). The realm creation is done by a collaboration among a virtual machine manager (VMM) at NW-EL0, the hypervisor, and the RMM (Step 2). After populating the realm memory, the hypervisor sends the activation command to the RMM. Once the realm is activated, it can receive CPU time, and the hypervisor is no longer able populate new content into the realm's address space.

**Model initialization.** After realm's kernel is booted, the realm establishes a TLS connection with the model

4. Altering these assumptions does not impact the security of the model, it only affects the quality of the ML service experienced by the NW app.

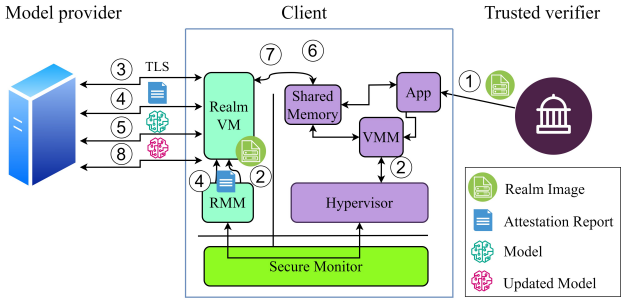


Figure 2: Overview of the steps required for running a ML model on the client edge device. We show a simplified view of the normal and realm worlds within the client. The client's steps are (1) obtaining realm image from verifier (2) creating and activating a realm VM (3) establishing connection with provider (4) realm attestation (5) obtaining model from provider (6) announcing model readiness to normal world (7) running inference (8) performing model updates.

provider (Step 3). Later, the realm sends an attestation request to the RMM and in return, the RMM sends the attestation report to the realm, which is forwarded to the model provider (step 4). The model provider can now use the attestation report to verify the content of the realm and decide whether it can trust the realm or not. On verification, the model provider sends the model to the realm via the TLS connection (Step 5).

**Inference.** The realm's kernel includes a virtio-9p driver, which is used to establish a file-system-based shared memory with the NW app. After receipt of the model, the realm announces its ability to respond to inference queries to the NW app (Step 6). Later, the NW app sends input data to the realm, the realm feeds it into the model, obtains the inference, and writes the output back to the shared file system so that the NW app can read it (Step 7).

**Service maintenance.** In addition to performing inference, the framework must also handle other maintenance operations. For example, a model provider might set usage limits—such as a validity period or maximum inferences—by embedding this functionality in the realm image. Once these limits are reached, the realm calls the hypervisor to terminate and release its memory. The realm can also periodically query the model provider for updates on the model (step 8).

**Integration with mobile devices.** Our framework can be adapted for deployment on mobile devices. A potential setup involves a hypervisor supporting CCA running at EL2, with Android at EL1 and user applications at EL0. In this configuration, while Android remains responsible for managing applications running at EL0, the hypervisor can create and manage Realm VMs, enabling secure execution environments for sensitive models.

# 4. Evaluation

In this section, we evaluate and compare our framework against a baseline scenario in which the model is deployed within a NW VM. We show the computational overhead and privacy benefit of our framework.

## 4.1. Experimental Setup

In this section, we describe the experimental setup used to evaluate our framework. We compare our framework with a baseline scenario, involving deploying model within a normal world VM. We use FVP to report the overhead of our framework in comparison to the baseline. FVP is instruction-accurate, that is, it accurately models the instruction-level behavior of a real processor that supports CCA [21], [33]. However, it does not effectively capture certain micro-architectural behaviors (e.g., caching and memory accesses), which makes cycle-accurate and timing-based measurements unreliable [33]. While we use FVP to report the number of instructions executed by the FVP's processor core, these measurements should be regarded as preliminary estimations. We do not claim that they represent the actual performance overhead on real CCA hardware. In Appendix A, we provide extensive information on how to set FVP in conjunction with tracing tools to accurately measure number of instructions executed by the FVP's core. We also use *Shrinkwrap* [34], a tool that simplifies the building and execution of firmware/software on FVP. Shrinkwrap automatically downloads and builds necessary firmware based on the given configuration files. More information on software and firmware version we used for the evaluation is provided in Appendix A.

**Privacy protection.** As discussed in the threat model (Section 3.2), all software in the normal world is considered untrusted. Consequently, in the baseline scenario—where the model runs within a normal world VM—the model is entirely exposed to potential adversaries. An adversary with this level of white-box access can launch privacy-stealing attacks to infer information about the model's training dataset. In contrast, our framework protects the model by executing it within realm, effectively concealing its weights from NW adversaries. Specifically, our framework restricts the adversary's access to the model to a black-box setting, where only query access is allowed. In Section 4.3, we demonstrate the resulting privacy advantages by evaluating both white-box and black-box membership inference attacks.

**Models.** Table 2 shows an overview of the models and settings used in the evaluation. We choose models of varied sizes and types for typical on-device tasks like image classification, speech recognition, and chat assistants. For each model, an appropriate VM size is chosen, which is enough for the run-time progress of inference. The size of the virtual machine depends mainly on the use of inference code, the size of the model, and the size of dynamic libraries required for each model.

## 4.2. Inference Overhead

In order to evaluate the overhead of our framework, we perform an evaluation with two scenarios. In the baseline scenario, the model and the code are stored in a NW VM. In the second scenario, the model and code are stored in a realm VM. In both scenarios, a file system-based data sharing is established between the VM and the NW app, allowing the NW app to send input queries and receive inference outputs. In order to get more insights about the

TABLE 2: Experimental settings used in the evaluation. The VM size depends on runtime memory use of inference code, size of model, and size of dynamic libraries required for each model.

| Experimental Setting | Model | Model Size (MB) | Library (API) | Input Format | VM size (MB) |
|---|---|---|---|---|---|
| ① | AlexNet | 9 | TensorFlow Lite (C++) | .bmp | 300 |
| ② | MobileNet_v1_1.0_224 [35] | 16 | TensorFlow Lite (C++) | .bmp | 400 |
| ③ | ResNet18 | 44 | TensorFlow Lite (C++) | .bmp | 450 |
| ④ | Inception_v3_2016_08_28 [36] | 95 | TensorFlow (C++) | .jpg | 1750 |
| ⑤ | VGG | 261 | TensorFlow (C++) | .wav | 3650 |
| ⑥ | GPT2 [37] | 177 | llama.cpp [38] (C++) | text | 900 |
| ⑦ | GPT2-large [39] | 898 | llama.cpp [38] (C++) | text | 1800 |
| ⑧ | TinyLlama-1.1B-Chat-v0.5 [40] | 1169 | llama.cpp [38] (C++) | text | 2000 |

TABLE 3: Mean (standard deviation) of instructions executed per inference service. Each experimental setting is described in Table 2.

| Setting | Model Initialization ($10^6$) | | | Read Input ($10^6$) | | | Inference Computation ($10^6$) | | | Write Output ($10^6$) | | | Total ($10^6$) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *R VM* | *NW VM* | *Ovh* | *R VM* | *NW VM* | *Ovh* | *R VM* | *NW VM* | *Ovh* | *R VM* | *NW VM* | *Ovh* | *R VM* | *NW VM* | *Ovh* |
| ① | 1.6 | 1.2 | 33% | 0.6 | 0.3 | 100% | 98.0 | 82.0 | 19% | 1.1 | 0.5 | 120% | 105.9 | 87.8 | 20% |
| ② | 1.7 | 1.2 | 41% | 4.7 | 1.1 | 100% | 335.4 | 278.9 | 20% | 0.7 | 0.3 | 133% | 351.8 | 289.3 | 21% |
| ③ | 2.1 | 1.6 | 31% | 0.6 | 0.3 | 100% | 418.2 | 344.0 | 21% | 0.9 | 0.4 | 125% | 442.8 | 363.2 | 20% |
| ④ | 397.9 | 333.4 | 19% | 2.8 | 1.8 | 55% | 7663.8 | 6382.8 | 20% | 4.6 | 3.5 | 31% | 8717.2 | 7201.1 | 21% |
| ⑤ | 345.1 | 295.8 | 16% | 1.8 | 1.1 | 63% | 6365.7 | 5420.7 | 17% | 0.15 | 0.09 | 66% | 6713.2 | 5717.9 | 17% |
| ⑥ | 1039.1 | 821.9 | 26% | 2.7 | 1.8 | 50% | 12036.6 | 9858.7 | 22% | 0.11 | 0.04 | 75% | 13144.9 | 10726.3 | 22% |
| ⑦ | 2653.6 | 2158.5 | 22% | 2.7 | 1.8 | 50% | 73603.1 | 59870.6 | 22% | 0.07 | 0.04 | 75% | 76412.3 | 62156.4 | 22% |
| ⑧ | 2784.9 | 2312.1 | 20% | 2.6 | 1.8 | 44% | 94480.0 | 79452.7 | 18% | 0.07 | 0.04 | 75% | 97433.3 | 81905.6 | 18% |

inference service, we divide the service into four stages and measure each one separately, (1) model initialization, which involves loading the model into memory allocated by the inference code, (2) getting input from the NW app and storing it in the inference code memory (3) inference computation, which refers to local computations within the VM to obtain the output, and (4) writing the output back to the NW. For each experiment in both scenarios, we instantiate five VMs and perform five inferences per VM, yielding a total of 25 repetitions per configuration. We report the mean values, however standard deviations are omitted as they are consistently below 10% in all experiments.

Table 3 shows the results of our evaluations. As illustrated, the total overhead of inference service within the realm is moderate, ranging from 17% to 22%. Model initialization overhead varies between 16% to 41% depending on the API used for inference. The highest numbers are within experiments ①, ②, and ③, all using the TFlite API. On the other hand, overhead of read input and write output are between 44% and 100%, and 31% and 133%, respectively, showing considerably bigger overhead in I/O-involved operations within realm. The variation in input read overhead is primarily due to differences in input size across models, while the variation in output write overhead is attributed to the number of output classes and the format in which outputs are returned to the NW app. As explained in Section 2.2, the main contributor to these overheads is the increased complexity of exception handling in the realm. Although I/O operations are relatively expensive in this setting, they represent only a small portion of the total computation and do not significantly affect the overall inference performance. We also perform another experiment to see how much each entity is responsible for the overhead of inference computation and report the results in Appendix B. Finally, it is important to note that these results represent only an initial approximation, based on the number of instructions executed by the simulator's core. We do not report these figures as overheads that would be replicated on actual CCA hardware.

### 4.3. Membership Inference Attack

To demonstrate the security benefits of our framework, we conduct both white-box and black-box membership inference attacks on two models (experimental setting ① and ③ in Table 2). We adopt the MIA proposed in [13], using the same settings and hyper-parameters (*e.g.*, learning rate, number of epochs, etc). In this attack, the adversary has access to a shadow dataset drawn from the same distribution as the training dataset. The adversary then uses the shadow dataset to train a binary classifier that infers membership in the target training dataset (see Section 2.4 for details). While the default assumption in [13] is that the shadow dataset size matches that of the training dataset, this assumption may not be realistic in all practical scenarios. To account for this, we experiment with three different ratios between the training dataset and shadow dataset sizes. Specifically, we fix the size of the training dataset across all scenarios and reduce the shadow dataset size to 1/4 and 1/8 of the training dataset size. We conduct these experiments using two models and three different datasets. The results, presented in Table 4, shows that the adversary's success rate decreases by an average of 12.4% and 4.2% for AlexNet and ResNet18, respectively (8.3% reduction on average). These findings are consistent with similar observations in [28]. Notably, the gap between the adversary's success rates in the two settings grows as the number of output classes increases. The gap is larger for CIFAR100 (100 output classes) than for CIFAR10 (10 output classes) and CelebA (configured for 32 output classes in our evaluation).

## 5. Discussion

**Realm device assignment.** Device assignment is one of the planned future enhancements for CCA [41], and it

TABLE 4: Adversary's success rate in the membership inference attack. NW: Model is deployed within NW, giving the adversary white-box access to the model, RW: Model is deployed within realm world, giving the adversary black-box (label-only) access to the model. R is the ratio between the size of shadow dataset and the size of training dataset

| Model | Deployment | R = 1 | | | R = 1/4 | | | R = 1/8 | | | Total (Average) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *CF10* | *CF100* | *CelebA* | *CF10* | *CF100* | *CelebA* | *CF10* | *CF100* | *CelebA* | |
| AlexNet | NW | 71.8 | 84 | 84.9 | 65.4 | 83.9 | 82.4 | 57.3 | 76.9 | 82.6 | |
| | RW | 68.9 | 76.0 | 69.0 | 66.3 | 50.0 | 68.6 | 67.9 | 50.0 | 60.6 | |
| | Diff | 2.9 | 8.0 | 15.9 | -0.9 | 33.9 | 13.8 | -10.6 | 26.9 | 22.0 | 111.9 (12.4) |
| ResNet18 | NW | 70.0 | 91.9 | 84.1 | 69.9 | 89.4 | 86.9 | 66.4 | 87.9 | 85.5 | |
| | RW | 68.9 | 85.8 | 81.4 | 68.5 | 73.3 | 81.0 | 68.9 | 80.9 | 80.0 | |
| | Diff | 1.1 | 6.1 | 2.7 | 1.4 | 16.1 | 5.9 | -2.5 | 7.0 | 5.5 | 37.8 (4.2) |

could enable the deployment of new capabilities across the ML pipeline. Securely assigning specialized hardware—such as GPUs and NPUs—to realm could significantly accelerate inference computation. More importantly, device assignment opens the possibility of protecting the entire inference pipeline within the TEE boundary. Although our current system protects the model itself from NW adversaries, it does not protect the source of input data. In safety-critical applications—such as health monitoring or autonomous driving—corrupted inputs can pose serious risks. Thus, achieving strong guarantees requires securing the entire inference workflow, including:(1) input generation, (2) delivery of inputs to the model, (3) generation of outputs, and (4) consumption of outputs by the requester.

**Membership Attack on LLMs.** The larger memory size of the realm, as compared to other on-device TEE solutions, allowed us to run LLMs within a realm. However, we did not show the privacy benefit of running the LLM within a realm. Future works could explore the trade-off between performance and privacy when deploying LLMs in realm compared to NW. Currently, several studies have examined MIA in black-box settings [42], [43] while others [44] have questioned the assumptions of previous attacks, investigating whether MIAs are feasible under more realistic conditions for LLMs. White-box MIAs for LLMs remain an emerging area, with no proposed white-box attacks demonstrating consistent superiority over black-box approaches.

**Limitation.** For the evaluations in this paper, we have emulated CCA using FVP, our results are only initial approximation not obtained from a real hardware. Accurate evaluation can be done in the future when a real device supporting Arm CCA will be available.

## 6. Related Works

**Model partitioning on end-devices.** To overcome limitation of current TEEs, several works have proposed to partition model in which more sensitive parts are running within a TEE – these include shielding deep layers [11], [12], shallow layers [45], intermediate layers [46], non-linear layers [47] within a TEE. Zhang *et al.* [13] showed that those partitioning solutions are vulnerable to privacy attacks when public information like datasets and pre-trained models engages in attacks.

**TEE extensions.** There are works aim to overcome the limitations of TEE by introducing system-level techniques. SANCTUARY [48] and LEAP [9], for instance, create isolated user-space enclaves in NW on top of TrustZone. However, in both works, the secure world (TrustZone) is trusted, making them vulnerable to malicious actors within the secure world. This is a significant concern, as [8] demonstrated, current implementations of TrustZone suffer from critical vulnerabilities. To address these issues, REZONE [49] proposes a system that deprivileges the TEE's operating system, offering enhanced protection against potentially malicious TEE components. Li *et al.* [50] Introduces a method to allocate large memory for TrustZone apps by modifying OP-TEE. However, the total amount of memory available to OP-TEE remains limited to the configured size at boot time.

**Systems based on CCA.** As CCA is still under development, there is limited prior work in this space. Formal methods is introduced in [29], [30] to verify security and functional correctness of RMM. SHELTER [24] provides user-space isolation in the normal world using CCA hardware primitives. ACAI [25] is a system that allows CCA realms to securely access PCIe-based accelerators with strong isolation guarantees. DEVLORE [22] is a system that allows realm VM to access legitimate integrated devices (*e.g.*, keyboard) with necessary memory protection and interrupt isolation from an untrusted hypervisor. GuaranTEE [27] took initial steps in using CCA for ML tasks. This framework provides attestable and private machine learning on the edge using CCA and evaluated it for running a small model within realm. In this work, we adopt their system model and utilize tracing tools to estimate the system's overhead.

## 7. Conclusion

In this paper, we presented an in-depth evaluation of Arm's Confidential Computing Architecture (CCA) as a solution to protect on-device models. We measure both the overhead and the privacy gains of running models of various sizes and functionalities within a realm VM. Our results indicate that, CCA can be a viable solution for model protection. While various challenges still remain before CCA's widespread deployment, we provide the first indication of its suitability as a mechanism to provide model protection.

## Acknowledgments

We wish to acknowledge the thorough and useful feedback from anonymous reviewers and our shepherd. The research in this paper was supported by the UKRI Open Plus Fellowship (EP/W005271/1 Securing the Next Billion Consumer Devices on the Edge) and an Amazon Research Awared "Auditable Model Privacy using TEEs".

# References


[1] C. Orlandi, A. Piva, and M. Barni, "Oblivious neural network computing via homomorphic encryption," *EURASIP Journal on Information Security*, vol. 2007, pp. 1–11, 2007.

[2] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *International conference on machine learning*. PMLR, 2016, pp. 201–210.

[3] T. van Elsloo, G. Patrini, and H. Ivey-Law, "SEALion: A framework for neural network inference on encrypted data," *arXiv preprint arXiv:1904.12840*, 2019.

[4] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," in *2017 IEEE symposium on security and privacy (SP)*. IEEE, 2017, pp. 19–38.

[5] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," in *Proceedings of the 2018 on Asia conference on computer and communications security*, 2018, pp. 707–721.

[6] A. Limited, "Learn the architecture - TrustZone for AArch64," Accessed Feb 2025. [Online]. Available: https://developer.arm.com/documentation/102418/latest/

[7] "Software Guard Extensions." [Online]. Available: https://en.wikipedia.org/wiki/Software_Guard_Extensions

[8] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1416–1432.

[9] 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*, 2022.

[10] OP-TEE, "Q: What's the maximum size for heap and stack? Can it be changed." [Online]. Available: https://optee.readthedocs.io/en/latest/faq/faq.html#q-whats-the-maximum-size-for-heap-and-stack-can-it-be-changed

[11] F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi, "Darknetz: towards model privacy at the edge using trusted execution environments," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, 2020, pp. 161–174.

[12] F. Mo, H. Haddadi, K. Katevas, E. Marin, D. Perino, and N. Kourtellis, "PPFL: privacy-preserving federated learning with trusted execution environments," in *Proceedings of the 19th annual international conference on mobile systems, applications, and services*, 2021, pp. 94–108.

[13] 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 *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 52–52.

[14] A. Limited, "Arm Confidential Compute Architecture," Accessed Feb 2025. [Online]. Available: https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture

[15] M. Sardar, T. Fossati, and S. Frost, "SoK: Attestation in confidential computing," *ResearchGate pre-print*, 2023.

[16] TrustedFirmware, "TF-RMM," Accessed Feb 2025. [Online]. Available: https://www.trustedfirmware.org/projects/tf-rmm

[17] A. Limited, "Learn the architecture - AArch64 Exception Model," Accessed Feb 2025. [Online]. Available: https://developer.arm.com/documentation/102412/latest/

[18] Linaro, "qemu," Accessed Feb 2025. [Online]. Available: https://git.codelinaro.org/linaro/dcap/qemu

[19] A. Bennée, "Building an RME stack for QEMU." [Online]. Available: https://linaro.atlassian.net/wiki/spaces/QEMU/pages/29051027459/Building+an+RME+stack+for+QEMU

[20] T. L. Foundation, "Arm Confidential Compute Architecture open-source enablement," Accessed Feb 2025. [Online]. Available: https://confidentialcomputing.io/webinars/arm-confidential-compute-architecture-open-source-enablement/

[21] A. Limited, "Fast Models Fixed Virtual Platforms (FVP) Reference Guide," Accessed Feb 2025. [Online]. Available: https://developer.arm.com/Tools%20and%20Software/Fixed%20Virtual%20Platforms

[22] 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 A Journey Beyond Memory to Interrupt Isolation," *arXiv preprint arXiv:2408.05835*, 2024.

[23] 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 *Network and Distributed System Security (NDSS) Symposium*, 2024.

[24] 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 *32nd USENIX Security Symposium (USENIX Security'23)*, 2023.

[25] S. Sridhara, A. Bertschi, B. Schlüter, M. Kuhne, F. Aliberti, and S. Shinde, "ACAI: Extending Arm Confidential Computing Architecture Protection from CPUs to Accelerators," in *33rd USENIX Security Symposium (USENIX Security'24)*, 2024.

[26] J. Chen, Q. Zhou, X. Yan, N. Jiang, X. Jia, and W. Zhang, "CubeVisor: A Multi-realm Architecture Design for Running VM with ARM CCA," in *2024 Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2024, pp. 1–13.

[27] S. Siby, S. Abdollahi, M. Maheri, M. Kogias, and H. Haddadi, "GuaranTEE: Towards Attestable and Private ML with CCA," in *Proceedings of the 4th Workshop on Machine Learning and Systems*, 2024, pp. 1–9.

[28] Y. Liu, R. Wen, X. He, A. Salem, Z. Zhang, M. Backes, E. De Cristofaro, M. Fritz, and Y. Zhang, "ML-Doctor: Holistic risk assessment of inference attacks against machine learning models," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4525–4542.

[29] 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 *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 465–484.

[30] A. C. Fox, G. Stockwell, S. Xiong, H. Becker, D. P. Mulligan, G. Petri, and N. Chong, "A Verification Methodology for the Arm® Confidential Computing Architecture: From a Secure Specification to Safe Implementations," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 376–405, 2023.

[31] Y. Yuan, Z. Liu, S. Deng, Y. Chen, S. Wang, Y. Zhang, and Z. Su, "CipherSteal: Stealing Input Data from TEE-Shielded Neural Networks with Ciphertext Side Channels," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 79–79.

[32] ——, "Hypertheft: Thieving model weights from tee-shielded neural networks via ciphertext side channels," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 4346–4360.

[33] A. Limited, "Fast Models Reference Guide," Accessed Feb 2025. [Online]. Available: https://developer.arm.com/Tools%20and%20Software/Fixed%20Virtual%20Platforms

[34] "Shrinkwrap," Accessed Feb 2025. [Online]. Available: https://shrinkwrap.docs.arm.com/en/latest/overview.html

[35] "TensorFlow Lite_Label Image," Accessed Feb 2025. [Online]. Available: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/examples/label_image

[36] "TensorFlow_Label Image," Accessed Feb 2025. [Online]. Available: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/label_image

[37] "GPT2," Accessed Feb 2025. [Online]. Available: https://huggingface.co/openai-community/gpt2

[38] ggerganov, "llama.cpp," Accessed Feb 2025. [Online]. Available: https://github.com/ggerganov/llama.cpp

[39] "GPT2-large," Accessed Feb 2025. [Online]. Available: https://huggingface.co/openai-community/gpt2-large

[40] "TinyLlama/TinyLlama-1.1B-Chat-v0.5," Accessed Feb 2025. [Online]. Available: https://huggingface.co/TinyLlama/TinyLlama-1.1B-Chat-v0.5

[41] "MAD24-410 Arm Confidential Compute Architecture open-source enablement update," May 17, 2024. [Online]. Available: https://resources.linaro.org/en/resource/rEjhEezEvnNMC3LALzUTrr

[42] F. Galli, L. Melis, and T. Cucinotta, "Noisy Neighbors: Efficient membership inference attacks against LLMs," *arXiv preprint arXiv:2406.16565*, 2024.

[43] R. Xie, J. Wang, R. Huang, M. Zhang, R. Ge, J. Pei, N. Z. Gong, and B. Dhingra, "ReCaLL: Membership Inference via Relative Conditional Log-Likelihoods," *arXiv preprint arXiv:2406.15968*, 2024.

[44] M. Duan, A. Suri, N. Mireshghallah, S. Min, W. Shi, L. Zettlemoyer, Y. Tsvetkov, Y. Choi, D. Evans, and H. Hajishirzi, "Do membership inference attacks work on large language models?" *arXiv preprint arXiv:2402.07841*, 2024.

[45] J. Hou, H. Liu, Y. Liu, Y. Wang, P.-J. Wan, and X.-Y. Li, "Model Protection: Real-time privacy-preserving inference service for model privacy at the edge," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 6, pp. 4270–4284, 2021.

[46] T. Shen, J. Qi, J. Jiang, X. Wang, S. Wen, X. Chen, S. Zhao, S. Wang, L. Chen, X. Luo *et al.*, "SOTER: Guarding Black-box Inference for General Neural Networks at the Edge," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 723–738.

[47] 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 *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1596–1612.

[48] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with User-space Enclaves." in *NDSS*, 2019.

[49] D. Cerdeira, J. Martins, N. Santos, and S. Pinto, "ReZone: Disarming TrustZone with TEE Privilege Reduction," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2261–2279.

[50] J. Li, X. Luo, H. Lei, and J. Cheng, "TEEm: Supporting Large Memory for Trusted Applications in ARM TrustZone," *IEEE Access*, 2024.

[51] TrustedFirmware, "TF-A," Accessed Feb 2025. [Online]. Available: https://www.trustedfirmware.org/projects/tf-a

[52] A. Limited, "linux-cca," Accessed Feb 2025. [Online]. Available: https://gitlab.arm.com/linux-arm/linux-cca

[53] Buildroot, "buildroot," Accessed Feb 2025. [Online]. Available: https://github.com/buildroot/buildroot

[54] "kvmtool-cca," Accessed Feb 2025. [Online]. Available: https://gitlab.arm.com/linux-arm/kvmtool-cca

## Appendix A. Experimental Setup

**Software stack.** We use the Trusted Firmware-A [51] (v2.11), and the Trusted Firmware implementation of RMM [16] (tf-rmm-v0.5.0) as the Monitor and the RMM of the software stack (Figure 1), respectively. We separately build linux-cca [52] and the file system for each experiment and pass them to Shrinkwrap. Shrinkwrap later boots FVP with the necessary firmware and the given kernel and file system. We also use Buildroot [53], to create customized file systems for each experimental setup. In order to create a virtual machine, we need to provision a virtual machine manager (VMM) to the hypervisor's file system. Both kvmtool-cca [54] (cca/rmm-v1.0-eac5) and Linaro's QEMU [18] (cca/v3) have support for realm VMs, but for each one, we need to use a compatible branch of linux-cca (which has a similar name to the branch of that VMM).

**FVP Accuracy.** FVP promises to accurately model the instruction behavior of a real processor [21], [33]. However, some micro architectural behaviors (*e.g.*, caching and memory accesses) are different between FVP and an actual device, making cyclic and timing measurements unreliable. [33]. Therefore, we do not report timing or cycle-accurate performance results from the simulation. While some studies [23]–[25] have prototyped CCA on existing Armv8-A hardware, these platforms lack essential features—such as GPC support in system registers and accurate cache behavior—which pose challenges to the accuracy of such prototypes. Although our framework is evaluated using FVP, these hardware-based prototypes may still be valuable for others, particularly for enabling cycle-level and timing evaluations.

**Instruction tracing in FVP.** FVP can be used in conjunction with tracing tools and plugins to provide detailed information about the behavior of CCA. Particularly, we use *GenericTrace* to choose a trace source (*e.g.*, instructions in our case) and *ToggleMTIPlugin* to enable/disable tracing during runtime. We configure GenericTrace to trace and print each instruction executed by an FVP's processor core, along with other metadata. The metadata includes the security state and the exception level of the core when running that instruction, and the total number of instructions executed until that point in time. Using ToggleMTIPlugin, FVP can be set to be sensitive to a particular assembly instruction[5]. Whenever this instruction is executed by the FVP's processor core, tracing is automatically started/stopped. We add this instruction at points in the code to enable and disable GenericTrace. This is necessary to reduce the size of the trace file and only get what it is necessary for each experiment. Lastly, similar to what has already been done by Sridhara *et al.* [25], we add a set of assembly instructions to the code to mark specific points (*e.g.*, beginning and end of inference) in the final trace file. Later, by analyzing this trace file, we can get the result of evaluations including number of instructions executed (for example between the beginning and end of inference).

**Runtime isolation.** Since FVP simulates a multi-core device, additional measures are necessary to ensure that the target workload is executed exclusively on the traced core. To achieve this, we utilize a kernel-command line parameter called *isolcpus* to isolate one core from the hypervisor's general load balancing and scheduling algorithms. This ensures that the hypervisor's scheduler does not assign any processes to the traced core by default. Subsequently, during runtime, we use the *taskset* tool to explicitly direct the hypervisor to use only the isolated core for the process that oversees the virtual machine.

**On-demand memory delegation.** During the VM's boot process, the hypervisor [52] delegates only the physical pages necessary to load the kernel and file system images. The remaining memory in the VM's address space is delegated on-demand, triggered by the first access to

5. We used HLT 0x1337

those addresses. To decouple this one-time overhead from the main experiment in each evaluation, we address it by running a user-space program within the VM. This program temporarily allocates all available memory in the virtual machine's user space and fills it with binary 1's. This ensures that the hypervisor delegates the entire memory beforehand, preventing any memory delegation during the main experiment.

**Experimental Hosts.** The membership inference attack in Section 4.3 is conducted on a system with dual Intel Xeon Gold 6136 CPUs (48 cores, 3.7 GHz max) and 251 GiB RAM, utilizing an NVIDIA Quadro GV100 GPU for acceleration. The environment run on Ubuntu 22.04.1 with kernel 6.5.0. Although FVP results are independent of the host platform, we report the system specifications for completeness. We conduct all FVP-related experiments on a Lenovo ThinkCentre M75t Gen 2 with 16GB RAM and an 8-core AMD Ryzen 7 PRO 3700 processor (OS: Ubuntu 22.04.4 LTS). We set FVP to have two clusters, each with four cores supporting Armv9.2-A and 4GB of RAM.

## Appendix B. Inference Overhead

In order to identify the source of overhead within the inference computation, we conduct an additional experiment to quantify the engagement of firmware and software components during inference computation. Using configuration ②, we deploy two VMs – one within the Realm world and the other in the NW – with both performing the same task (a single inference). We then measured the number of instructions executed by each software and firmware component in the system. The results are presented in Table 5. In both experiments, the number of executed instructions at EL0 and EL1 are relatively the same. However, significant differences emerge at EL2 and EL3, which are the main contributors to the overhead in the realm. Specifically, the virtualization support for the NW VM requires only 14.8 million instructions executed by the hypervisor. In contrast, the Realm VM required 16.84 million instructions executed by the hypervisor, with an additional 41.18 million instructions executed by the RMM and 5.13 million by the Monitor. These results suggest that the RMM is the main source of overhead, accounting for more than twice the number of executed instructions by the hypervisor. Worth noting that these measurements are done during the inference computation and there is no I/O involved.

## Appendix C. Realm Setup Overhead

In this section, we evaluate the overhead associated with booting and terminating a realm VM in comparison to a baseline scenario (a NW VM). As illustrated in Table 6, the overhead for booting and terminating a realm VM is substantial, with observed increases ranging from 867% to 21,902% for booting and from 644% to 3,521% for termination. These elevated overheads are primarily due to the additional RMM checks and processes required for page delegation (during boot) and reclaming those pages (during termination). Notably, the overhead for both booting and termination escalates with the size of the VM, as reflected in the experimental settings detailed in Table 2, with the exception of boot overhead between ④ and ⑤. This results suggests that, although realm booting and termination represent one-time costs, they become significantly burdensome when deploying larger models, which typically necessitate larger VM sizes.

TABLE 5: Number of instructions (in millions) executed by each software/firmware component for a single inference in both normal and realm VMs. These results correspond to experimental setting 2 in Table 2.

| **Exception Level** | **Realm VM Experiment** | | **NW VM Experiment** | |
|---|---|---|---|---|
| | ***Realm World*** | ***Normal World*** | ***Realm World*** | ***Normal World*** |
| EL0 | 240.14 | 0.04 | 0 | 240.18 |
| EL1 | 24.68 | 0 | 0 | 23.85 |
| EL2 | 41.18 | 16.84 | 0 | 14.80 |
| EL3 | 5.13 | | 0 | |

TABLE 6: Mean (standard deviation) of number of instructions executed for realm boot and termination. Each experimental setting is described in Table 2.

| **Experimental Setting** | **VM Boot** ($10^6$) | | | **VM Termination** ($10^6$) | | |
|---|---|---|---|---|---|---|
| | ***Realm VM*** | ***NW VM*** | ***Overhead*** | ***Realm VM*** | ***NW VM*** | ***Overhead*** |
| ② | 7630.1 (52.6) | 788.7 (0.7) | 867% | 619.9 (3.3) | 83.3 (0.1) | 644% |
| ④ | 24960.7 (132.9) | 1246.6 (0.9) | 1902% | 2332.4 (2.4) | 93.1 (0.2) | 2405% |
| ⑤ | 44499.3 (10.9) | 2329.4 (5.2) | 1832% | 5156.4 (6.9) | 142.4 (0.3) | 3521% |
| ⑥ | 21101.5 (71.4) | 1195.0 (0.2) | 1665% | 1325.3 (2.4) | 87.1 (0.1) | 1421% |