CAGE: Complementing Arm CCA with GPU Extensions

Chenxu Wang*‡, Fengwei Zhang†*✉, Yunjie Deng†*, Kevin Leach§,
Jiannong Cao‡, Zhenyu Ning¶, Shoumeng Yan‖ and Zhengyu He‖
*Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
†Department of Computer Science and Engineering, Southern University of Science and Technology, China
‡Department of Computing, The Hong Kong Polytechnic University, China
§Institute for Software Integrated Systems, Vanderbilt University, USA
¶Hunan University, China, ‖Ant Group, China
{12150073, 12032869}@mail.sustech.edu.cn, zhangfw@sustech.edu.cn, kevin.leach@vanderbilt.edu
csjcao@comp.polyu.edu.hk, zning@hnu.edu.cn, {shoumeng.ysm, zhengyu.he}@antgroup.com

***Abstract*—Confidential computing is an emerging technique that provides users and third-party developers with an isolated and transparent execution environment. To support this technique, Arm introduced the Confidential Computing Architecture (CCA), which creates multiple isolated address spaces, known as realms, to ensure data confidentiality and integrity in security-sensitive tasks. Arm recently proposed the concept of confidential computing on GPU hardware, which is widely used in general-purpose, high-performance, and artificial intelligence computing scenarios. However, hardware and firmware supporting confidential GPU workloads remain unavailable. Existing studies leverage Trusted Execution Environments (TEEs) to secure GPU computing on Arm- or Intel-based platforms, but they are not suitable for CCA's realm-style architecture, such as using incompatible hardware or introducing a large trusted computing base (TCB). Therefore, there is a need to complement existing Arm CCA capabilities with GPU acceleration.**

To address this challenge, we present CAGE to support confidential GPU computing for Arm CCA. By leveraging the existing security features in Arm CCA, CAGE ensures data security during confidential computing on unified-memory GPUs, the mainstream accelerators in Arm devices. To adapt the GPU workflow to CCA's realm-style architecture, CAGE proposes a novel shadow task mechanism to manage confidential GPU applications flexibly. Additionally, CAGE leverages the memory isolation mechanism in Arm CCA to protect data confidentiality and integrity from the strong adversary. Based on this, CAGE also optimizes security operations in memory isolation to mitigate performance overhead. Without hardware changes, our approach uses the generic hardware security primitives in Arm CCA to defend against a privileged adversary. We present two prototypes to verify CAGE's functionality and evaluate performance, respectively. Results show that CAGE effectively provides GPU support for Arm CCA with an average of 2.45% performance overhead.

✉Fengwei Zhang is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24763
www.ndss-symposium.org

I. Introduction

Confidential computing is an emerging technique that provides users and third-party developers with an isolated and invisible execution environment. Both cloud platforms and endpoints, which support confidential computing, secure sensitive data from all unauthorized access, including illegal applications, untrusted clients, and even cloud providers [5], [14], [43], [65]. To support confidential computing, the major processor companies proposed the corresponding concepts and hardware primitives, such as Intel's Trust Domain Extensions (TDX) [51], AMD's Secure Encrypted Virtualization (SEV) [3], and IBM's Protected Execution Facility (PEF) [49].

Arm also proposed the design of Arm Confidential Compute Architecture (CCA) [20] and its hardware security primitive—*Realm Management Extensions (RME)* [22] —to support confidential computing in next-generation [12] Arm devices. CCA introduces realms, the basic unit of the confidential computing environment, and the Realm Management Monitor (RMM), which behaves as a thin hypervisor for realm isolation. To mitigate the latency in realm execution, an untrusted hypervisor is delegated to schedule the realms and manage memory resources, but cannot access realms.

While CCA provides strong data security and enables confidential computing on next-generation Arm devices, the support for GPUs [4], [21], [69], which are widely used to accelerate the general-, high-performance, and artificial intelligence computing scenarios [15], [18], [34], [45], [55], is only recently proposed. However, such support, called RME Device Assignment (RME-DA) [25], is currently a high-level concept without completed hardware implementation. Currently, CCA design characterizes the GPU as an untrusted peripheral on which data security is not guaranteed. As a result, if realms use the GPU for computation, their sensitive data can be accessed by the adversary who controls the GPU via compromised software (e.g., a compromised GPU driver or programming model).

To address the problem of data security in confidential GPU computing, recent study [73] proposes a similar design to RME-DA, while its protection mechanism requires non-trivial modification on hypervisor software. Moreover, it introduces heavyweight GPU software to realm's TCB. Previous works

have developed Trusted Execution Environments (TEEs) for GPUs. However, most solutions [31], [39], [52], [54], [61], [67], [70], [76], [81] rely on hardware security primitives that are not suitable for next-generation Arm devices (e.g., customized hardware, Intel-based security primitive, or traditional Arm security hardware), incurring tremendous challenges in mitigation. For example, HIX [52] leverages the Intel SGX, which is not supported in Arm devices, and StrongBox [39] trusts *secure world* software, which are not trusted in Arm CCA. Overall, it leads to a critical question: **How can we extend the support of GPU acceleration to Arm CCA?**

We present CAGE, which supports confidential GPU computing on next-generation Arm devices. By leveraging the hardware security primitives (i.e., the RME) in Arm CCA, CAGE secures the confidential computing on the widely deployed unified-memory GPUs [7], [21], [68], [71], which are expected to become the mainstream accelerators in next-generation Arm devices. We design CAGE based on two critical observations (detailed in §II): (1) CCA provides novel hardware-assisted security features, such as the Granule Protection Check (GPC), to flexibly isolate and protect computing on both the CPU and peripherals, and a hardware-isolated *root world* to configure these features. Therefore, CAGE deploys its security modules in the *root world* Monitor, protecting GPU computation from untrusted software (e.g., the OS, hypervisor, and *secure world* components) and peripherals. (2) Most of the components in the GPU software stack, which perform essential functions (e.g., memory allocation and task scheduling), do not require access to sensitive data and code. Thus, CAGE delegates the untrusted GPU software to schedule confidential GPU applications without direct access to the sensitive data. We use these key insights to design the workflow of CAGE, supporting confidential GPU computing on Arm CCA.

The design of CAGE faces three key challenges. **C1:** The GPU software in the host is not intended to manage applications in realms. Since GPU software requires frequent interaction with the GPU application, it unavoidably introduces non-negligible performance overhead due to communication with the hypervisor and world switching. To handle this problem, CAGE employs a novel shadow task mechanism. Specifically, CAGE permits the GPU software stack to create and manage stub GPU applications corresponding to real GPU applications in realms, then verifies and synchronizes these operations to the real GPU applications in the Monitor (§IV-B). **C2:** Without the latest RME-DA support, generic Arm CCA design regards the GPU device as *normal world* peripheral and disallows it to access the realm. Thus, it is not intended to achieve the same CPU-side isolation on GPU nor protects the GPU execution environment from various attacks. Worse, the unified-memory GPU shares the memory with the untrusted software and peripherals, exposing a large attack surface during the computing. To address this challenge, CAGE proposes a two-way isolation mechanism for GPU environment protection. By leveraging the existing GPC for the CPU and peripherals, CAGE provides different GPU memory views for each realm during confidential GPU computation while securing the GPU environment from the untrusted software and peripherals (§IV-C). **C3:** Based on the above GPU protection mechanism, CAGE must create multiple Granule Protection Tables (GPTs) for the corresponding GPCs and synchronize these GPTs during the GPU environment protection, generating additional performance overhead. For this challenge, we propose optimization techniques in GPT maintenance, especially on synchronizing GPTs for untrusted components and initializing GPU GPTs for realms (§IV-D).

We prototype CAGE on two official platforms: The Arm Fixed Virtual Platform (FVP) [24] with software-simulated Arm CCA feature and the Arm Juno R2 development board [10] with a Mali-T624 GPU. Our prototype introduces 1,301 lines of code (LoC), reaching a thin TCB. We analyze the system design of CAGE by fairly comparing it with other CCA extensions (including the RME-DA) and the state-of-the-art GPU TEEs. Moreover, we compare the performance by running the Rodinia GPU benchmark suite [37], a popular benchmark for varied GPU TEEs [39], [52]. Next, we examine the effectiveness of our optimization techniques and verify the robustness with three neural network models (LeNet-5 [56], SqueezeNet [50], and MobileNet-v1 [46]). We also discuss the security of CAGE against the assumed adversary. Results indicate that CAGE securely provides GPU support for Arm CCA with an average of 2.45% performance overhead.

We claim the following contributions to this work:

- We present CAGE, which provides GPU acceleration support for Arm CCA. In confidential GPU computing, CAGE secures the sensitive data from the strong adversary assumed in Arm CCA.
- We prototype CAGE on the Arm official emulator and a real-world development board without hardware changes. We share the source code of CAGE[1] and will maintain it to benefit the Arm community.
- We comprehensively evaluate CAGE with respect to its performance and security. The results indicate that CAGE effectively secures the sensitive data and code with an average of 2.45% performance overhead.

II. Background

A. Arm TrustZone and Arm CCA

Arm introduced the hardware-based TrustZone [8] that provides an isolated and confidential environment for sensitive data and security-critical code. TrustZone creates two separate *worlds* for execution states: (1) *normal world*, which runs the traditional OS, applications, and hypervisor, and (2) *secure world*, which is a mirrored but isolated environment to run secure components. Resources such as memory, interrupts, and peripherals are also separated into the two worlds based on the TrustZone hardware (e.g., TZASC [9] and TZPC [27]). As part of the firmware, TrustZone leverages special Secure Monitor code, which is of the highest privilege, to handle the world switching (e.g., through a special instruction, *smc*) and resource partitioning to protect the security-critical data and code in *secure world*. However, studies [35], [36] have shown that the *secure world* components, such as the secure OS and secure hypervisor, are still vulnerable: an attacker can gain control of these components to fully compromise the system, including the *normal world* software and the Secure Monitor.

[1] https://github.com/Compass-All/NDSS24-CAGE

Table I: Access control of physical address space in GPC.

Security State	Normal PAS	Secure PAS	Realm PAS	Root PAS
Normal	✓	✗	✗	✗
Secure	✓	✓	✗	✗
Realm	✓	✗	✓	✗
Root	✓	✓	✓	✓

In the latest Armv9 architecture [12], Arm has advanced the state of confidential computing through two enabling technologies: (1) the Confidential Compute Architecture (CCA) [20], and (2) an associated hardware primitive, called Realm Management Extensions (RME) [22]. Figure 1 shows the software architecture of Arm CCA. CCA retains the *normal world* and *secure world* dichotomy. In addition, CCA introduces a *realm world* in which to run multiple confidential realms. These realms are isolated from the other worlds, but are managed by untrusted software components (e.g., a *normal* hypervisor). As for the memory isolation between realms, CCA provides a lightweight Realm Management Monitor (RMM) running in the hypervisor-layer of *realm world*. Besides the *realm world*, CCA also provides a *root world* in which to house the highest-privilege Monitor code stored in the firmware. This Monitor leverages a new memory isolation mechanism, *Granule Protection Check* (GPC), to achieve flexible and fine-grained access control for the entire main memory. Specifically, when a software component accesses a physical address space (PAS), the GPC checks the *Granule Protection Table* (GPT) to fetch the security attribute of the PAS and determine whether the access is valid (summarized in Table I). For security purposes, CCA advises storing the GPT in *root* PAS and only permits the Monitor to configure GPC registers.

Key observation. Compared to the traditional Arm TrustZone, Arm CCA secures the highest-privilege Monitor in a hardware-isolated *root world* and provides novel hardware-assisted mechanisms for memory isolation. Based on this, CAGE deploys its security modules in the Monitor to additionally secure GPU computation, such as controlling access to GPU memory and GPU registers.

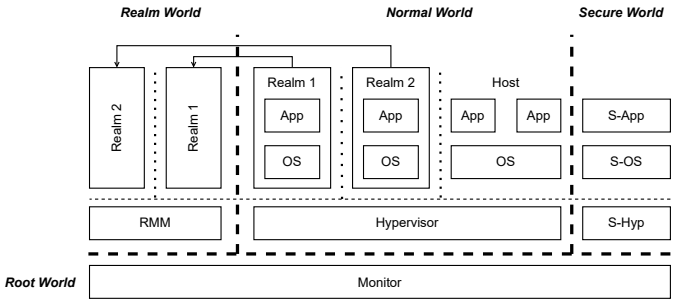

Figure 1: Architecture overview of Arm CCA.

B. Arm GPU and SMMU

Unlike discrete GPUs on Intel-based devices, mainstream Arm GPUs share a unified memory with the CPU and other peripherals. To perform data transfer and communication between the host and GPU, Arm employs a set of GPU software, including the kernel-layer driver (e.g., Midgard [16] driver) and the user-layer runtime (e.g., OpenCL [17] libraries). Specifically, the GPU software manages (1) the GPU computation environment and (2) the interaction with the GPU hardware.To prepare the execution environment, the GPU software allocates physical memory and creates GPU buffers based on the requirements of the GPU task. Next, it loads the critical components of the GPU task into the GPU memory, including GPU task code, data, and metadata (e.g., job descriptors that indicate the address of the GPU buffer). Note that the GPU software stack should create the GPU page table and configure the corresponding GPU registers to allow the GPU to access the critical components via Direct Memory Access (DMA). As for interacting with the GPU hardware, the GPU software schedules the execution order and submits the GPU tasks via Memory-Mapped Input/Output (MMIO). When GPU computation is terminated, the GPU software fetches the execution results and restores the environment.

Since the GPU and other peripherals share the main memory with the CPU, Arm introduced the System Memory Management Unit (SMMU) to manage DMA-capable peripherals. Currently, most Arm GPUs (e.g., Arm Mali GPUs [21], Qualcomm Adreno GPUs [71], NVIDIA Tegra X1 Maxwell GPUs [68]) and other peripherals are physically connected to an SMMU. Like the CPU MMU, SMMU supports address translations (i.e., Stage-1 and Stage-2 translation) to control the access from the peripheral to the PAS. To enable access control, privileged software configures SMMU registers (e.g., page table registers and translation configuration registers) via MMIO. Besides address translation, GPC is also supported on SMMU [23] in Arm CCA. To protect the SMMU GPC, CCA introduces additional SMMU MMIO registers that are only accessible to the *root world*. These registers provide basic configurations of SMMU GPC, such as GPT base, GPC controls, fault handling, and TLB invalidation.

Key observation. The workflow of Arm GPU can be adapted to Arm CCA's realm-style architecture. On the one hand, the GPU software performs most of its functions without accessing sensitive data. We use this observation to reserve these components in the untrusted host and perform data-dependent functions within a TCB. On the other hand, a peripheral's memory access is also subject to the corresponding GPC, which allows CAGE to ensure GPU environment isolation.

III. Threat Model and Assumptions

Following with Arm CCA, we assume a strong adversary who controls the entire software stack in both *normal world* and *secure world*, including the GPU software, untrusted OS and hypervisor, and the same layer software in *secure world*. The adversary aims to leak or tamper with the sensitive data (i.e., the input, intermediate data, and execution result) of confidential GPU tasks. Specifically, the adversary can directly access the unified memory that stores the sensitive data or control DMA-capable peripherals to perform the same attack. Moreover, the adversary seeks to break the isolated execution environment by compromising the GPU software components, such as compromising the memory management, changing the execution order of the confidential tasks, and modifying the GPU register states. We also assume that the adversary may submit malicious GPU tasks and use GPU to access or tamper with the sensitive data inside other realms. In addition to software attacks, we can defend against several physical attacks (e.g., cold-boot attacks [78]) on memory by using the memory encryption support built in Arm CCA. We further

discuss these in §VII. Lastly, we consider the side-channel attacks and Denial-of-Service are out of the scope of this paper.

On the hardware side, we assume next-generation Arm devices use hardware security primitives such as the RME, which provides hardware extensions for Arm CCA, and hardware root of trust, which assists the secure boot, remote attestation, and constructing secure communication channel with a realm user. We also trust the GPU and assume its memory access is subjected to GPC, such as an SMMU with RME support [23]. On the software side, we trust the Monitor since its firmware is securely verified and loaded during the secure boot.

IV. Design

CAGE provides the support of GPU acceleration for Arm-based confidential computing. Based on this, we design CAGE by achieving four critical goals:

G1: Compatibility with CCA. CAGE should follow Arm CCA's realm-style architecture (i.e., creating and managing realms by *normal world* software but hiding the sensitive data in realms from this software) to manage confidential GPU computation. More concretely, CAGE must delegate the complex but data-independent functions (e.g., memory management and task scheduling) to the untrusted GPU software stack and ensure data security with a marginal increase in TCB.

G2: Strong data security. As a supplement for Arm CCA, CAGE must protect the data security of realms from the strong adversary assumed in Arm CCA. During confidential GPU computation, CAGE must defend against attacks from privileged software (e.g., untrusted OS, hypervisor, and the *secure world* software) and untrusted peripherals.

G3: Optimized performance. Compared with the native GPU execution workflow, CAGE must not generate a high performance overhead during confidential GPU computation.

G4: No hardware modification. CAGE must preserve hardware compatibility with next-generation Arm devices. In the design and implementation of CAGE, we must leverage generic hardware features in Arm CCA and GPU without introducing hardware changes.

A. CAGE *Overview*

We envision scenarios where users or third-party developers request a realm to persistently store and execute the confidential GPU applications. The realm user transfers sensitive data through a secure and encrypted channel [32], [58], [79] to the requested realm. To follow Arm CCA's realm-style architecture, the realm user provides (1) GPU task code and (2) descriptions of data buffers to the untrusted GPU software to help construct a stub execution environment, including metadata, GPU buffers, and GPU page table. Since the untrusted GPU software may tamper with code and descriptions, the user should also transfer the signatures of code and descriptions to the realm for integrity verification. Once the environment is created, CAGE protects the GPU, creates a real GPU environment to replace the stub one, and submits the real GPU task to the GPU after security introspection. Lastly, the GPU computes sensitive data and stores results in the realm, from which user retrieves the data via a secure channel.

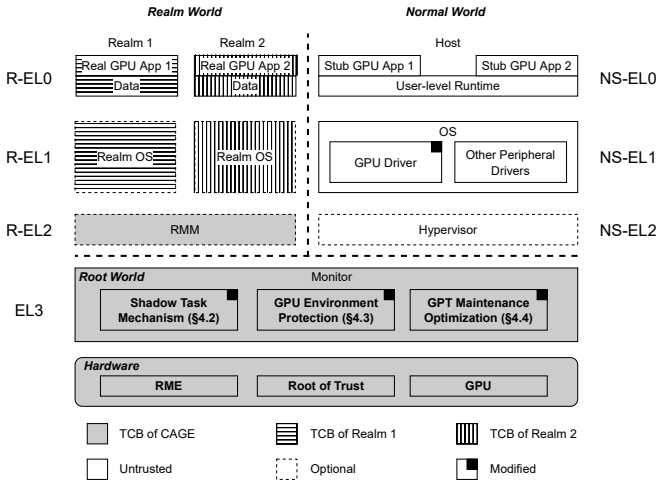

Figure 2: CAGE architecture overview. Note that the *secure world* is omitted since CAGE does not modify the software components in this world.

Figure 2 shows an overview of CAGE. Based on the existing architecture and hardware primitives in Arm CCA, CAGE supports generic Arm-based confidential computing with GPU acceleration. CAGE deploys three security components in the highest-privilege Monitor to guarantee confidential GPU computation. The Monitor is housed in the *root world* that is natively isolated from the untrusted components in other worlds. In addition, it provides several APIs to configure these hardware primitives that enable reaching CAGE's security goals. (**G1:**) To achieve confidential GPU computation on next-generation Arm devices, CAGE uses a shadow task mechanism (§IV-B) based on CCA's realm-style architecture. To avoid exposing sensitive data, the shadow task mechanism requires the untrusted GPU software stack to create and manage stub GPU tasks, which are securely replaced by real GPU tasks with authentic data before submission. (**G2:**) Next, CAGE ensures strong data security during the process of the above GPU workflow. The RMM and previous work [80] isolate realms on CPU side but have yet to isolate GPU computation. Thus, we propose a two-way realm isolation on GPU by leveraging the GPC on CPU, GPU and untrusted peripherals. We restrict access from untrusted components by configuring GPCs for CPU and untrusted peripherals. Further, we isolate realms in GPU computing by providing each realm with different GPU memory view in GPU GPC (§IV-C). (**G3:**) Moreover, since our memory protection mechanism manages various GPT views for realms and peripherals, we also mitigate the additional performance overhead in GPT management. Our optimization mechanism focuses on two aspects: (1) synchronizing CPU and untrusted peripheral GPTs, and (2) initializing GPU GPTs. To achieve this, we design a specific structure of GPTs and the corresponding maintenance process (§IV-D) without undermining the security of our protection. We deploy our security modules in the *root world* Monitor and introduce minimal modification on the untrusted GPU driver to collaborate with the Monitor. (**G4:**) Note that our design neither requires additional hardware nor customizes the existing CPU and GPU, ensuring high hardware compatibility.

Deployment scenario. As shown in Figure 2, CAGE only requires realms to store the sensitive data provided by the realm

user and real GPU applications, while Realm OS and RMM are optional. It allows CAGE to handle application scenarios on both servers and endpoint devices. When running on servers, CAGE trusts a RMM, which collaborates with the hypervisor to create and isolate realm VMs. Based on this, CAGE extends confidential GPU computing support for these realm VMs. As for endpoint devices that currently do not run a hypervisor (and possibly do not run a RMM in next-generation devices), CAGE satisfies confidential GPU computation for user-level realms, which can be created and isolated by GPU GPC in recent work [80]. Following CCA's realm-style architecture, our confidential GPU computation starts with being created through *normal world* GPU software, then proceeds to the Monitor to execute the GPU workload confidentially.

B. Shadow Task Management

The primary goal of CAGE is to provide GPU acceleration support for Arm CCA. To achieve this goal, a naive solution [52], [54], [73], [81] is to encapsulate the heavyweight GPU software stack (e.g., the GPU driver and the user-level runtime) into each realm. However, this introduces a very large TCB within the realm and exposes a correspondingly large attack surface. To address this problem, we adapt CCA's realm-style architecture to the GPU workflow. Specifically, we delegate the heavyweight, complex but data-independent functions to the *normal world*, and meanwhile preserve sensitive data within realms. Based on this, realms benefit from GPU acceleration without needing to expose the whole TCB entailed by the entire GPU software stack.

C1. However, adapting this workflow to CCA's realm-style architecture is challenging, primarily because the GPU software on the host is not intended to create and manage GPU applications belonging to other virtual machines or the realms. This workflow requires frequent interaction with the hypervisor and world switching, generating a large performance overhead and even conflicting with the functionality of the hypervisor.

Solution to C1. To address this challenge, we propose a novel shadow task mechanism. Figure 3 shows the design of the shadow task mechanism. We build a pair of GPU tasks: (1) the stub GPU task, which has the same structure as a normal GPU task (e.g., the GPU buffers, metadata, and GPU page table) but does not contain any sensitive data, and (2) the real GPU task, which includes sensitive data to be processed on the GPU hardware. Thus, we require the GPU software on the host to create and manage the stub GPU task during the confidential GPU computing without frequent interactions with the hypervisor, the monitor, and the realms. Next, we replace the stub task with the real task data in the Monitor and synchronize data-independent operations to the real task. To guarantee data security, we secure the data path between the real task and the GPU hardware. After GPU computation, the execution results are finally stored in the realm and fetched by the realm user. We elaborate the workflow of the shadow task mechanism as follows.

Initialization and stub task creation. A key aspect of the shadow task mechanism is to create an empty stub GPU application which is later populated with real task data. By doing so, an adversary that compromises the system will only see the empty stub task and will not have access to the sensitive task data that is protected by realms. To achieve this, the user provides the host with two essential components: The GPU task code and the descriptions of GPU data buffers. The descriptions show the expected attributes of the data buffers, such as the buffer size and input data. Considering that the untrusted host may compromise execution integrity by modifying the code and descriptions, or changing the execution order of GPU task, we require the user to additionally provide a GPU task signature (e.g., Hash-based Message Authentication Code) for integrity verification. Specifically, the user generates the signature by combining the memory content of code, descriptions, and task index. Next, the user transfers the sensitive data and signatures to the realm via a secure channel. As a result, CAGE uses signature to verify the integrity of GPU tasks before submission.

Based on the code and descriptions, the GPU software allocates memory, prepares the stub task, and stages sensitive code and data descriptions into the corresponding data buffers. One additional issue here is to record the creation and update of the GPU page table for the stub task (i.e., `Stub GPU PTE` in Figure 3), which enables synchronizing the page table of the real task with the stub task (i.e., `Real GPU PTE` in Figure 3). However, using simple creation/synchronization approaches for the two page tables, such as (1) copying the entire stub table to the real one, or (2) synchronously replaying the operations on the stub table to the real one, can incur substantial performance overhead. To mitigate this overhead, one key observation is that the GPU software generally does not update the GPU page table during execution, which allows CAGE to asynchronously replay the previous operations on the stub table to the real one. To achieve this, the GPU software records the new or updated entries of the stub page table in batch, then submits them to the Monitor for further update or synchronization after security introspection.

When the stub task is prepared, the GPU software inserts it into the GPU task queue to schedule the task execution order. The GPU software follows the general workflow to process the non-confidential GPU tasks (i.e., directly submitting the normal tasks to the GPU hardware), while the workflow is different for handling the stub tasks. Since the running tasks inside the GPU may map to the GPU memory of other tasks (and thus leak sensitive data), the GPU software must help to create an exclusive computing environment for the stub tasks. Therefore, when processing the stub tasks, the GPU software stack cleans up the GPU hardware by (1) temporarily blocking the submission of other tasks and (2) waiting until the current GPU computation (if any) is finished. When the GPU computation completes, it unblocks the submission of new tasks. Next, the GPU software routes the stub tasks to the Monitor to proceed with execution. However, since the adversary can compromise the GPU software, we must double-check the GPU status in the Monitor.

Real task creation and execution. When receiving the stub task, the Monitor creates a corresponding real task inside the realm, then replaces the stub task with the real one and submits the real task to the GPU device. We provide the memory layout of this process in Figure 3 and detail the operations as follows.

The structure of the real task is similar to that of the stub task. Specifically, it reuses the metadata and code buffers inside the stub task but prepares different data buffers and

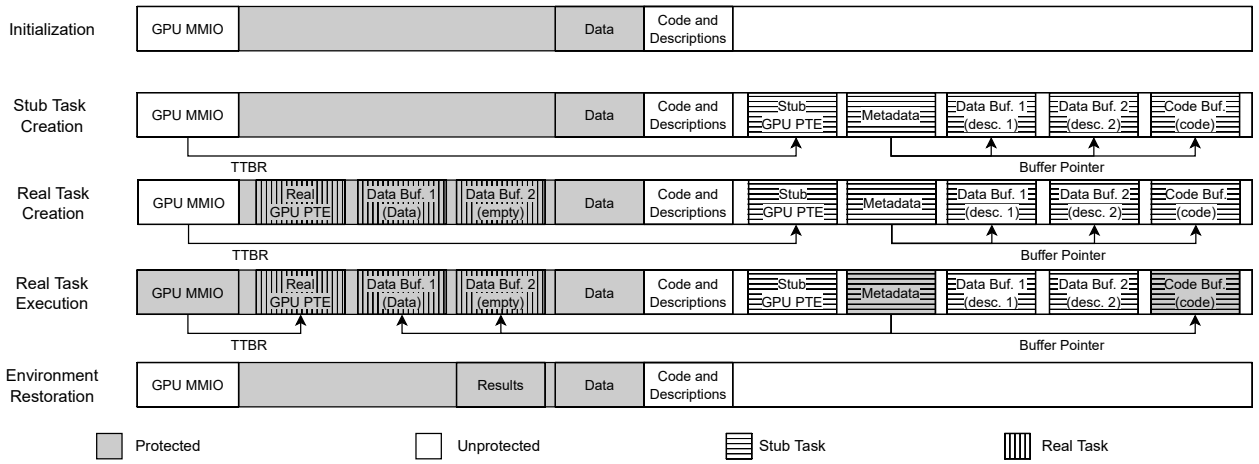

Figure 3: The design of shadow task mechanism. Note that Data Buf. 1 is the input buffer and Data Buf. 2 is the output buffer.

GPU page tables. To create the data buffers, we verify the integrity of the descriptions by the provided signature, then faithfully construct buffers based on the provided attributes (e.g., the buffer size). If the description requires, we fill sensitive data into the target real data buffers. Next, to construct the page table, we check the recorded page table entries (e.g., whether they have duplicated mapping or illegal mapping) and replay the mappings in the realm. Since the stub data buffers do not participate in the GPU computing, we change the corresponding mappings to map the real buffers.

To execute the real task, the Monitor first protects the GPU environment (detailed in §IV-C) by locking three regions: (1) the GPU MMIO, which includes the base address of the GPU page table (i.e., `TTBR` in Figure 3), (2) the metadata, which contains the pointer to the GPU buffers, and (3) the code buffers. If these regions are unprotected, the adversary can leak or modify sensitive data (e.g., by exporting the execution results to an unprotected buffer or by executing malicious code). After protection, we replace the stub task with the real task by (1) changing the stub GPU page table to the real one in `TTBR` and (2) modifying the buffer pointer to point to the new buffers in the realm. Considering that the compromised GPU software may provide incorrect task code and data buffer descriptions, we must verify the signature to ensure integrity. The adversary may also tamper with the metadata or provide incorrect address of metadata, while it incurs a Denial-of-Service without leaking the sensitive data. Next, the Monitor checks the current GPU status, ensuring that no malicious GPU tasks are hidden in the GPU. Once the verification passes, the Monitor submits the real task by writing the start command to the corresponding GPU registers. Since the MMIO to the GPU registers is already protected before GPU status checking, the adversary can neither hide malicious tasks in the GPU nor modify the verification results.

Environment restoration. When the GPU finishes computing, the Monitor restores and cleans the execution environment in tandem with the GPU software stack. We first restore the values of GPU MMIO registers and metadata (e.g., buffer pointers). Next, we clean the previous GPU environment, such as flushing GPU caches and TLB entries of the GPU page table. Once the environment is cleaned, we restore access to GPU MMIO, metadata, and code buffers, while the execution results are stored inside the realm.

C. GPU Environment Protection

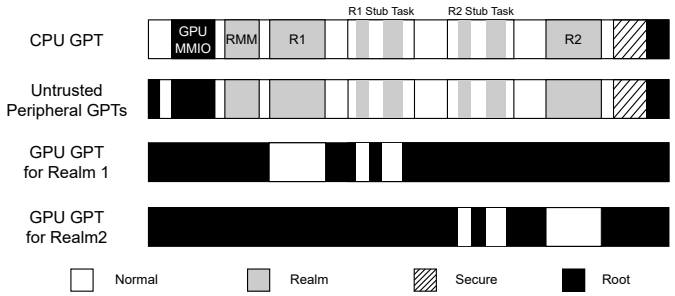

Figure 4: Access control and isolation of GPU environment when CAGE performs confidential GPU computation. The untrusted peripherals indicate the *normal* or *secure* peripherals with DMA capability and are accessible to memory with corresponding security states.

CAGE ensures data confidentiality, integrity, and isolated execution environment to support the confidential GPU acceleration. Based on the memory layout in the shadow task mechanism, we secure both the GPU runtime and the GPU memory, preventing illegal access to sensitive data, code, and GPU environment configuration.

C2. However, the latest support for GPU computation in Arm CCA (i.e., RME-DA [25]) is only an abstract concept without completed hardware or firmware implementations. Thus, GPU hardware in the existing implementation of Arm CCA is considered as a *normal* peripheral, whose security status cannot be re-configured as *realm* due to the lack of hardware support. This problem makes the existing memory isolation mechanisms for realms (i.e., the RMM) challenging to directly protect the execution environment of unified-memory GPU: First, since the GPU is considered *normal* hardware, it can be manipulated by privileged software from any world, including the *normal world* and *secure world* components that are untrusted in Arm CCA. Second, the RMM does not have excessive privilege compared to the same-layer software in other worlds (i.e., secure and normal hypervisor). Therefore, the RMM cannot introspect and prevent malicious access to the GPU environment from adversaries.

Solution to C2. We leverage the existing memory isolation mechanism in Arm CCA, GPC, to achieve two-way isolation between the GPU execution environment and the other components, including the untrusted software, peripherals, and other realms. Deployed on the CPU MMU and peripheral SMMUs, the GPC is initially used to show the security view of the main memory. We further extend this feature to achieve memory access control and GPU environment isolation for the *normal* GPU device.

Figure 4 shows various GPT configurations in our GPU environment protection mechanism. To achieve this, we design two types of GPT: (1) the CPU and untrusted peripheral GPTs for access control of untrusted components, and (2) the GPU GPT for realm isolation on GPU. The first type of GPT has a similar security view of the main memory but can be customized for different access control requirements (e.g., allowing a peripheral to access its MMIO but restricting the other peripherals to access the same region). When it is used for CPU GPC, it disallows the *normal* and *secure* software to access the protected regions (i.e., *realm* and *root* regions). Such restriction is also valid for the *normal* and *secure* peripherals that are fully controlled by the adversary. In regard to the GPU GPT for confidential computing, we provide each realm with a unique GPU GPT, in which the corresponding GPU environment is strictly isolated from the other regions. Our protection mechanism is based on the existing GPC design in Arm CCA without introducing any hardware modification of the CPU or GPU. We elaborate upon the design and configuration of GPU environment protection below.

Preventing access from untrusted components. CAGE restricts illegal access to the GPU environment from untrusted software and peripherals. Recall from §IV-B that we secure the sensitive data and the real data buffers in realms. Thus, the adversary cannot access the authentic data during GPU computing. However, we still need to temporarily protect (1) the reused metadata and code buffers, and (2) the GPU MMIO since the shadow task mechanism allows the untrusted software stack to manage these components. To secure the reused metadata and the code buffers, we configure these regions as *realm* in CPU GPT and untrusted peripheral GPTs, while the illegal access from other realms is prevented by CPU-side isolation (e.g., RMM). Considering that the compromised GPU driver may provide the incorrect address of the reused metadata and the code buffers, we must check whether these regions overlap with other *realm* or *root* regions in CPU GPT. If overlapped, we terminate the GPU computing and restore the GPU environment. As for the GPU MMIO protection, we configure this region as *root* in CPU GPT and untrusted peripheral GPTs. Note that the GPU MMIO is a fixed and unmodifiable region in most Arm-based devices [6], [10], [42], [74]. Moreover, we flush the TLB entries for CPU GPC and untrusted peripheral GPCs to defend against the TLB attacks.

Realm isolation on GPU. Besides access control on the CPU and untrusted peripherals, we must isolate the GPU's access to the unauthorized regions to complement our two-way isolation. Therefore, we design GPU GPTs for different realms. When the GPU executes a confidential task, CAGE fetches the corresponding GPU GPT to confine GPU's memory access. Since GPU GPTs and GPC configurations are only accessible to the *root world* Monitor, the executed task cannot bypass the GPU GPC to access the unauthorized physical memory. As for the configuration of GPU GPTs, we only permit access to the realm region, the reused metadata, and the code buffers, primarily because they can access the sensitive data of other realms via GPU. To allow GPU to access these regions, we configure the corresponding PAS as *normal* in GPU GPT. Finally, to enable GPU GPC for this realm, we configure GPU GPC registers (e.g., set the base address of GPU GPT to that of the realm) and flush the TLB entries. Since both CAGE and other CCA studies [73], [80] leverage GPC to achieve environment isolation, we provide a detailed comparison between CAGE and these studies in §IV-F.

The GPC configuration in CAGE does not conflict with the native GPC usability on CPU, GPU and other peripherals. For GPC on CPU-side access, CAGE does not interfere with the functionality of software components, such as the realm manager and the GPU software stack in the Host and the memory protection module in the Monitor. Moreover, the GPC on GPU-side access does not interfere with the memory access and computation within the GPU. Specifically, CAGE allows the GPU to access the page table, metadata, and buffers to compute using the data. Lastly, CAGE allows the untrusted peripherals to access their own device memory to fulfill their functional purpose. Thus, by configuring GPCs for CPU, GPU, and untrusted peripherals, CAGE achieves a two-way isolation between the GPU execution environment and other components. We also analyze our security guarantees in §VI-B.

D. GPT Maintenance Optimization

As discussed in §IV-C, CAGE ensures memory isolation between the confidential GPU environment and untrusted components with different GPTs. The CPU GPT and peripheral GPTs are initialized in the monitor during the boot phase. As for the GPU GPTs, they are initialized during the creation of the corresponding realms. Further, CAGE dynamically maintains these GPTs during confidential GPU computation.

C3: However, managing multiple GPTs can introduce non-trivial performance overhead for two reasons. First, during the confidential GPU computation, we must synchronize the access control of CPU and untrusted peripherals on their GPTs. Second, in the realm initialization process, we must create a unique GPU GPT that describes the fine-grained layout of the entire main memory.

Solution to C3. Based on the structure feature of GPTs, we present our optimization mechanism for these two issues.

First, we propose a new solution to mitigate the redundant synchronization process on CPU and peripheral GPTs. Recall from §IV-C that CAGE configures CPU GPT and peripheral GPTs to protect the reused metadata and code buffers. To simplify the synchronization process, a straightforward solution is to use a unified GPT to replace these GPTs. However, this may conflict with the customized access control requirements of these components. Nevertheless, we propose an optimized technique based on two features of GPT: (1) GPT supports a hierarchical architecture which is composed of a top-level table and a sub-level table. (2) Unlike the address translation entry, the GPT descriptors in the sub-level table only describes the security attribute of the target memory without the output address, the read/write permissions, and other attributes. These

features allow us to configure a sub-level GPT, which is shared with the CPU GPT and untrusted peripheral GPTs, to protect the reused region (i.e., metadata and code) on GPU memory. Figure 5 shows our optimization mechanism on GPT synchronization. We require the GPU software to prepare the stub GPU tasks on a specific memory region, whose access control is managed by a unified sub-level table. Next, we configure the table descriptors of various GPTs to point to the same sub-level GPT. Since this region is reserved for usage by the GPU, we flexibly modify the sub-level table to protect or unprotect the metadata and code without interfering with the functionality of other peripherals.

Second, we inspire from optimization [80] on CPU GPT construction to reduce the latency when creating GPU GPTs. The GPU GPT in CAGE only indicates two types of memory regions: The *normal* (i.e., accessible) regions and the *root* (i.e., inaccessible) regions. Thus, the GPU GPTs for different realms are derived from one GPT template with minimal adjustment. Specifically, we fork the GPU GPT from a template configuring the entire main memory as *root*, then set the realm region as *normal*. During confidential GPU computation, we additionally configure the reused GPU memory (i.e., code and metadata) as *normal* to permit the GPU to access them.

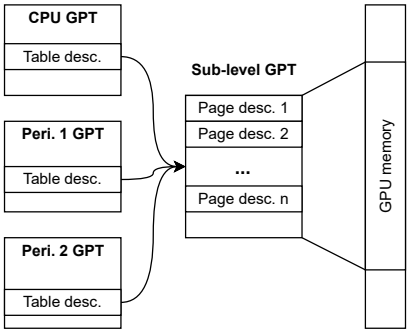

Figure 5: Optimization mechanism in GPT synchronization.

Based on these mechanisms, CAGE successfully mitigates the performance overhead in GPT maintenance. We also demonstrate the efficiency of our optimization in §VI-E.

E. Trust Establishment in CAGE

In this section, we discuss our assumptions of how the realm user establishes trust with the software and hardware components in confidential GPU computation.

Secure boot. CAGE leverages CCA's secure boot process [19] to safely initialize its security modules. During the secure boot process, it verifies the Monitor firmware image, the image metadata, and the payload to ensure the integrity and authenticity of CAGE's security modules. Based on this, we securely initialize CAGE to protect GPU computing.

Remote attestation and key management. Following Arm CCA's design [25], CAGE can assist the realm in attesting its execution environment by leveraging hardware Root of Trust (e.g., a hard-coded private key in ROM). Such attestation includes (1) the initial state of the realm, and (2) the Arm device including GPU hardware. CAGE can combine them to generate an attestation report for each realm to validate the GPU computing environment.

CAGE can benefit from other works [32], [39], [63] to manage keys for each realm. Specifically, the user can exchange keys with its realm via Diffie-Hellman [40] or Elliptic-curve Diffie-Hellman [64] protocol. To defend against the man-in-the-middle attacks, we can install a public key infrastructure (with certificates and public-private key pairs) in the Monitor. This infrastructure allows the user to authenticate its realm and encrypt the exchanged keys, preventing the attacker from impersonating the realm and leaking the secret keys. The exchanged keys are safely stored in the realm to defend against the unauthorized access. Based on this, a secure communication channel is built between the realm user and the realm through which to transfer the sensitive data and the signature. Thus, the transferred data are decrypted in realms before GPU computing, without requiring additional key exchange with the GPU.

F. Comparison to CCA Extensions and GPU TEEs

Comparison to RME-DA and ACAI. We discuss two CCA extensions for GPUs: The official RME Device Assignment (RME-DA) [25] and ACAI [73]. CAGE outperforms these extensions due to three reasons. First, CAGE maintains a smaller realm TCB compared with these designs. Realms running on RME-DA and ACAI must employ large GPU software stack to manage GPU applications. Instead, CAGE removes the software from the realm via the shadow task mechanism. Second, both RME-DA and ACAI rely on Stage-2 translation on SMMU, which suffers from the untrusted hypervisor, to isolate GPU execution environment. Thus, they must address the threat from hypervisor with non-trivial modification. Specifically, RME-DA introduces hardware changes on SMMU (e.g., adding a device permission table) to safely allocate GPU resources to realms, and ACAI replaces the SMMU configuration code with special interfaces in hypervisor software. Instead, CAGE isolates and protects GPU environment with GPC, which natively defends against the attack from the hypervisor. Third, CAGE is compatible with more Arm devices. The RME-DA and ACAI extensions rely on the RMM to build a GPU TEE, which is typically absent on endpoint devices. Nonetheless, RME-DA proposes a standardized mechanism (called TEE Device Interface Security Protocol) to attest and manage the GPU device interface, which potentially benefits CAGE for authenticating GPU hardware.

Comparison to Shelter. Next, we discuss a user-space isolation extension for Arm CCA, Shelter [80]. Although both CAGE and Shelter leverage GPC to isolate execution environments and optimize GPT maintenance, they differ in three aspects. First, Shelter only provides the CPU GPT configuration because it targets CPU-side isolation. However, since unified-memory GPUs share memory with untrusted components, CAGE must provide complete GPT configurations for GPC on CPU, GPU, and untrusted peripherals. Second, Shelter provides each isolated user space with a unique CPU GPT for environment isolation. In CAGE, we adapt this approach so that GPU GPT design not only isolates realms on the GPU, but also allows the GPU to access realms. This is because the GPU is regarded as a *normal* device and cannot be reconfigured as a *realm* device. Third, Shelter does not optimize redundant GPT synchronization processes on CPU and peripheral GPTs. In CAGE, we manage CPU and peripheral access to GPU memory with a sub-level GPT (described in §IV-D).

Table II: Comparison between CAGE and the state-of-the-art GPU TEEs.

	GPU Memory	Protection Mechanism	Minimal TEE TCB	Sharing of GPU	HW Compatibility	Native Size of Trusted Components (LoC)	Runtime TCB Increment (LoC)
Non-Arm Designs							
Graviton [76]	Dedicated	Customized GPU HW	✓	✓	✗	Firmware (N/A)	N/A
HIX [52]	Dedicated	Intel SGX	✗	✓	✗	Firmware (N/A)	N/A
HETEE [81]	Dedicated	Customized HW	✗	✗	✗	Firmware (N/A)	N/A
CURE [31]	Unified	Customized HW	✗	✗	✗	Firmware (N/A)	3.1K on Firmware
HoneyComb [61]	Dedicated	AMD SEV-SNP	✓	✓	✓	Security Monitors (N/A)	83K on Security Monitors
Arm Designs							
Cronus [54]	Dedicated	TZASC + S-Hyp	✗	✓	✓	S-Hyp (34K) + Monitor (0.4M)	6.4K on S-Hyp
StrongBox [39]	Unified	TZASC + S2 trans.	✓	✗	✓	OP-TEE (0.3M) + Monitor (0.4M)	1.2K on Monitor
GR-T [70]	Unified	TZASC + Remote VM	✗	✗	✓	VM (17M) + OP-TEE (0.3M) + Monitor (0.4M)	1K on VM + 3.5K on OP-TEE
ACAI [73]	Unified / Dedicated	GPC + RMM	✗	✗	✓	RMM (26K) + Monitor (0.4M)	0.4K on RMM + 1.6K on Monitor
CAGE	Unified	GPC	✓	✓	✓	CPU-side Isolation (2–26K) + Monitor (0.4M)	1.3K on Monitor

Thus, compared to Shelter, CAGE effectively and efficiently achieves confidential GPU computing.

Comparison to StrongBox. We further compare CAGE with the unified-memory GPU TEE, StrongBox [39]. Compared with StrongBox, CAGE is more suitable for confidential GPU computing on next-generation Arm devices for four reasons. First, StrongBox trusts the entire of the *secure world* components (e.g., a secure OS and secure hypervisor), while these components are excluded in CAGE's TCB to follow the threat model in Arm CCA. Second, CAGE supports GPU computation on both edge endpoints and servers. CAGE ensures confidential GPU environment by configuring GPCs on CPU and peripherals. Thus, hypervisors do not conflict with our protection and cannot bypass the GPC. However, StrongBox is not compatible with hypervisors since a compromised hypervisor can bypass the Stage-2 translation, which is the key protection mechanism in CAGE. Third, CAGE outperforms StrongBox in performance. CAGE allows a realm to receive encrypted data from the realm user and decrypt data ahead of computation. In addition, our shadow task mechanism enables creating data buffers inside the realm, so that we securely transfer plaintext data to these buffers and store the results. Since sensitive data are not operated upon by the GPU software directly, CAGE does not require cryptographic operations, which is essential to StrongBox in GPU computation. Fourth, CAGE is more flexible than StrongBox. To protect the GPU environment, StrongBox achieves application-level exclusivity for the GPU. Specifically, it blocks the submission of other confidential tasks until all tasks of the current application are finished. Instead, CAGE achieves task-level exclusivity by introducing the shadow task mechanism. Realms exclusively occupy the GPU hardware in a task-level time slot rather than needing to completely finish long-running applications, enabling flexible task scheduling.

Comparison to other GPU TEEs. Lastly, we compare CAGE with other GPU TEEs in Table II. Without hardware modification, CAGE leverages the latest memory protection mechanism in Arm CCA (i.e., GPC) to secure GPU TEE from a strong adversary, while hardware primitives in other GPU TEEs (e.g., Intel SGX in HIX [52] and Arm TZASC in Arm GPU TEEs [39], [54], [70]) are not suitable for next-generation Arm devices. Moreover, although CAGE does not optimize the native runtime TCB (i.e., the Monitor and the RMM), CAGE maintains a minimal TCB for realms by the shadow task mechanism instead of loading the heavyweight GPU software into the TEE [31], [52], [54], [70], [81]. Note that the shadow task mechanism also allows the GPU applications from both untrusted software and multiple TEEs to share the same GPU. In addition, we ensure task-level exclusivity for realms to secure GPU computing. In addition, we also compare the software TCB between CAGE and state-of-the-art in two aspects: the native size of trusted components and the size of incremented TCB. CAGE ensures data security with a thin TCB increment (1.3K LoC) on the Monitor, similar to several state-of-the-art (e.g., 1.6K LoC in ACAI [73] and 1.2K LoC in StrongBox [39]). We also discuss the performance comparison between CAGE and state-of-the-art in §VI-C.

V. Implementation

To implement CAGE, we craft two prototypes: (1) a functionality prototype, which verifies CAGE's functions and examines the system security, and (2) a performance prototype, which emulates the latency of the functionality prototype. We elaborate our implementation as follows.

A. Functionality Prototype

We prototype CAGE on Arm FVP Base RevC-2xAEMvA [24] simulator with RME enabled.[2] It simulates the latest Armv9 hardware features and is widely used in previous studies [41], [58], [60], [73], [80]. The FVP does not support an authentic unified-memory GPU model [26]. Instead, it provides a connected test engine that faithfully performs memory accesses as though it is a DMA-capable peripheral, and an SMMU with RME support, which controls memory access from peripherals. We leverage these simulated hardware devices to verify CAGE's functionality, including the shadow task mechanism and GPC protection. Specifically, we reserve a 1GB memory region (`0x880000000 – 0x8BFFFFFFF`) for the test engine. In this region, we control the test engine to perform memory access as a unified-memory GPU. We run Linux v5.3.0 kernel as the *normal world* Host and Trusted Firmware-A (TF-A) v2.8 [29] as the Monitor.

To verify the memory isolation mechanism in CAGE, we initialize GPTs on the FVP prototype and test with the CPU and the test engine peripheral. Note that the existing TF-A v2.8 firmware assigns the same GPT for these components, so we need to additionally prepare peripheral GPTs in a 32MB reserved memory (`0xA0000000 – 0xA1FFFFFF`). Moreover, to use these GPTs to control the memory access from CPU and peripherals, we configure the system registers of MMU and SMMUs. Specifically, we set `GPTBR_EL3` as the base of CPU GPT and set `GPCCR_EL3` to enable CPU GPC. Similarly, we configure `SMMU_ROOT_GPT_BASE` and `SMMU_ROOT_GPT_BASE_CFG` to enable peripheral GPC. Since we update both CPU GPT and peripheral GPTs in our GPU environment protection (described in §IV-C), we must invalidate the stale GPC TLBs in both MMU and

[2] CAGE uses the software-based simulator since currently no RME-enabled hardware is available.

SMMUs. To achieve this, we execute `TLBI PAALLOS` instruction to flush the TLB in MMU. As for TLB invalidation on SMMUs, we write the range of the region to be invalidated to `SMMU_ROOT_TLBI` and invalidation command to `SMMU_ROOT_TLBI_CTRL`. Overall, our prototype provides a faithful basis on which to demonstrate confidential GPU computation, which is shown in §VI.

B. Performance Prototype

Since the platform of our functionality prototype is not cycle-accurate [24], [80], we implement an ecologically valid physical prototype for measuring performance by adapting the Armv9 CCA features to Armv8 hardware. To measure the performance of CAGE, we further transplant our FVP prototype to an Arm Juno R2 board [10] with a Mali-T624 GPU and 8GB DRAM. To interact with the GPU, we deploy the Arm official Midgard GPU driver [16] and OpenCL [17] libraries on the board. To accurately estimate the performance overhead, we measure the CPU cycles via `CNTPCT_EL0` register. However, considering the architecture variance between the FVP and the development board, we must implement two additional measures on the performance prototype.

Emulation of Arm CCA. Since the processors of the development board are based on the traditional Armv8 architecture [11] and do not support Arm CCA, we develop an approach to faithfully model the expected real-world performance overhead associated with CCA-related operations.

First, we replace CCA-related instructions with Armv8 instructions that cost similar latency. We consider two types of instructions that must be replaced: (1) Read or write instructions from or to GPT system registers (e.g., `mrs` or `msr` instructions applied to `GPCCR_EL3` or `GPTBR_EL3` registers). In our prototype, we rewrite these instructions to operate on other Monitor system registers available on Armv8 (e.g., `ACTLR_EL3`). (2) TLB invalidation instructions related to the CPU GPC (e.g., `TLBI PAALLOS`). In our prototype, we emulate these instructions by executing TLB invalidation instructions for address translation.

Second, we replace the MMIO operations on peripheral GPCs since the development board does not implement the associated SMMU registers (e.g., `SMMU_ROOT_GPT_BASE_CFG`). We perform the memory access on other memory regions to emulate these operations. Moreover, to emulate TLB invalidation on SMMU (i.e., operating `SMMU_ROOT_TLBI_CTRL` and `SMMU_ROOT_TLBI` registers), we both perform memory access and TLB invalidation on the target memory region.

Third, we emulate the configurations on several GPT instances. We implement this step by directly operating on the table entries of these GPTs since it mainly requires read and write operations to the corresponding GPT memory. If maintenance operations must access the values of GPT base registers (i.e., `GPTBR_EL3` and `SMMU_ROOT_GPT_BASE_CFG`), we fetch them from the replaced registers or memory regions.

Interaction with GPU hardware. Although the test engine performs memory access as the GPU, the performance prototype still need three additional steps to interact with the Arm Mali GPU on development board. First, we verify the status of Mali GPU by accessing the `JS_STATUS` register for each GPU job slot via GPU MMIO. If the returned value is not zero, it indicates that the adversary hides malicious GPU tasks in the GPU. Second, when submit the GPU task, we write the start command into `JS_COMMAND_NEXT` register. Lastly, since the native GPU generates a *normal* interrupt (i.e., not handled by the Monitor) after execution, we must temporarily set this interrupt as the Monitor interrupt for further operations (e.g., catch the interrupt and restore environment) in the Monitor.

By following these steps, we devise a real Armv8 hardware prototype that can accurately model the latencies associated with implementing Armv9 CCA operations.

VI. Evaluation

In this section, we evaluate our CAGE prototypes (§V) with respect to five research questions:

RQ1: How large is the TCB of CAGE?

RQ2: Can CAGE defend against privileged adversaries?

RQ3: How much performance overhead does CAGE incur?

RQ4: How much overhead does CAGE incur on neural network models?

RQ5: How effective is our optimization on GPT maintenance?

For these questions, we evaluate the functionality prototype to answer **RQ1** and **RQ2**. Next, we measure the performance prototype for **RQ3**, **RQ4**, and **RQ5**.

A. RQ1: TCB Size of CAGE

Table III: Introduced TCB size of CAGE.

Function	Lines of Code (LoC)
Shadow Task Management	667
GPU Environment Protection	350
GPT Maintenance Optimization	137
Other Configuration	147
All	1,301

We use a generic code statistic tool, *cloc* [2], to measure the TCB size of CAGE in terms of standard lines of source code. CAGE modifies an Arm Trusted Firmware-A v2.8 [29] Monitor by introducing 1,301 LoC additions, which is detailed in Table III. In addition, CAGE trusts a thin CPU-side isolation software (e.g., TF-RMM v0.2 [28] with 26K LoC, or Shelter [80] with 2K LoC). Overall, the introduced TCB is smaller than the heavy-weight GPU software stack (e.g., a 30K LoC Midgard GPU driver [16] and 32MB OpenCL libraries [17]).

B. RQ2: Security Analysis of CAGE

To show that CAGE ensures data security in confidential GPU computing, we analyze a wide range of attacks based on our threat model (§III) and use them to evaluate our functionality prototype. Table IV shows a list of attack scenarios and adversary capabilities along with corresponding solutions.

Unauthorized memory access and modification. To subvert data security, the adversary may directly leak or tamper with the sensitive data inside the GPU memory. To defend against this attack, we leverage GPC to secure sensitive regions from untrusted software during confidential GPU computing. We

Table IV: Three types of adversary with the corresponding attack scenarios and the defense mechanism in CAGE. ① indicates the GPC on CPU and peripheral access. ② indicates the integrity verification. ③ indicates the Monitor checks. ④ indicates the fixed MMIO address. ⑤ indicates the hardware-assisted isolation of *root world*. ⑥ indicates the TLB invalidation. ⑦ indicates the CPU-side memory isolation.

Adversary Type	Attack Scenarios	Defense
Untrusted software	Unauthorized memory access and modification	①②
	Illegal GPU memory management	①③
	Illegal GPU task scheduling	②③
	Malicious GPU tasks	①③
	Fake GPU and SMMU	④
	CPU GPC circumvention	①⑤⑥
Peripherals	Malicious DMA	①
	Peripheral GPC circumvention	①⑤⑥
Realms	Realm abuse	①⑦

also require the realm user to transfer sensitive data via a secure channel. Besides having direct access to the sensitive data, the adversary could modify the metadata to export the execution result to the unprotected region. However, we prevent modification of metadata by GPC protection. Moreover, the adversary may modify the task code or the descriptions of data buffers to mislead the GPU computation or cause malicious code to execute. We address this by verifying the integrity of code and descriptions (e.g., calculating the signature) before submitting the GPU task. Since the authentic signature is already transferred to the realm via a secure channel, our integrity verification process cannot be subverted.

Illegal GPU memory management. Since CAGE delegates GPU software stack to manage GPU memory, the adversary may subvert these functions to leak sensitive data. For instance, an Iago-style [38] attack could return incorrect addresses of the allocated metadata and buffers. To defend against this attack, the Monitor checks whether the metadata and GPU buffers overlap with TCB of other realms and CAGE. Specifically, the Monitor verifies whether the PAS of the metadata and GPU buffers overlaps with the *normal world* PAS (i.e., the PAS of metadata and GPU buffers) in other realms' GPU GPT. The adversary may provide an incorrect metadata value to mislead the protection, while it incurs a Denial-of-Service without leaking the sensitive data. Another attack is to provide incorrect GPU mappings (e.g., double mapping or mapping to the unauthorized region). In this case, we have the Monitor validate mappings before updating the real GPU page table to mitigate this attack. In our functionality prototype, since the test engine may lack a private MMU to perform address translation, we measure this attack by configuring the SMMU Stage-2 translation. Note that the SMMU Stage-2 translation has similar page table structure as that in GPU and is also restricted by the SMMU GPC.

Illegal GPU task scheduling. The adversary may compromise the task scheduling to subvert confidential GPU computing. The adversary may submit the confidential task to the GPU instead of the Monitor. However, the submitted stub task does not contain any sensitive data. Another attack is to provide the incorrect owner of the stub task or the incorrect task execution order, while the task fails the integrity check in the Monitor. Note that the adversary cannot modify the signature which is already stored in the realm.

Fake GPU and SMMU. The adversary may impersonate a GPU device and route confidential tasks into the fake device. Besides the GPU, the adversary may emulate the SMMUs to spoof the GPC configuration. However, we ensure the Monitor interacts with the authentic hardware instead of the software-emulated device. Specifically, the Arm device manual [10] indicates that the physical addresses of both the GPU and SMMU MMIO registers are fixed and unmodifiable. Other malicious or fake devices would be mapped to other (unmodifiable) addresses. This allows us to ensure that any communication with the GPU will go to the real physical device.

Malicious GPU tasks. The adversary may directly leak or tamper with the sensitive data in realms. To achieve this, she may map the address space of the realms in the GPU page table of a malicious GPU task so that the malicious task can access the sensitive data via the GPU. However, this attack fails to bypass the GPU GPC restriction. Moreover, she may compromise the isolated execution environment of confidential tasks. During the confidential GPU computation, she may submit a malicious task to monitor the execution of victim tasks. Therefore, we control the GPU interrupt and the illegal access to GPU MMIO registers. In addition, she may launch an Iago-style [38] attack in the GPU driver. To hide a malicious GPU task, she may provide incorrect GPU status when submitting the confidential task to the Monitor. To defend against this attack, the Monitor must protect the GPU MMIO and verify the GPU registers again before submission. Overall, CAGE defends against malicious GPU tasks with GPC protection and additional security checks in the Monitor.

CPU GPC circumvention. We consider three approaches by which an adversary may bypass the CPU GPC. First, the adversary may disable the GPC or replace the authentic CPU GPT with a malicious one. However, the adversary cannot access the GPC-related registers since she lacks *root world* privilege. Second, the adversary may modify the CPU GPT to remove memory isolation. To defend against this attack, we place GPTs in *root world* memory. Thus, any illegal access to the GPT generates a GPC fault and fails to proceed. Third, the adversary may exploit GPC TLB entries to access the newly protected regions (e.g., the metadata and GPU MMIO). To defend against this attack, we must invalidate the TLB entries when modifying the CPU GPT.

Malicious DMA. The adversary may control other *normal* and *secure* peripherals (e.g., sensors, USB, and display devices) to perform malicious DMA to the GPU execution environment. To defend against this attack, CAGE configures the corresponding peripheral GPC to restrict DMA to these regions. Specifically, we follow the configurations in CPU GPT to set the protected regions with the *root* or *realm* attributes, which disallow access from the untrusted peripherals. In addition, the adversary may execute a malicious confidential application to replay the same attack. As a mitigation, we also configure the GPU GPT for confidential GPU applications, so that the application is only permitted to access the corresponding realm in confidential GPU computing.

Peripheral GPCs circumvention. Similar to the attack scenario with untrusted software, the adversary may attempt to bypass the peripheral GPCs to perform malicious DMA on GPU memory, realms, and the TCB of CAGE. However, they fail to access the GPC-related registers due to the lack of *root*

Table V: Problem size of the selected Rodinia benchmark.

Application	Size	Data Buffers	Tasks	Memory
KNN	42764 nodes	2	1	0.49 MB
PF	100000 × 100 points	4	5	38.59 MB
LUD	2048 × 2048 nodes	1	382	16.00 MB
H3D	512 × 512 × 8 nodes	3	500	24.00 MB
LMD	25 × 25 × 25 boxes	4	1	63.42 MB
GS	2048 × 2048 nodes	3	4094	32.01 MB

privilege. In addition, we protect peripheral GPTs so that the adversary cannot modify them to revoke the access control. Considering that the adversary may leverage the TLB to bypass our GPC, we invalidate the TLB entries in SMMU when the GPT is modified. Note that the adversary cannot modify these TLB entries since it is not supported on Arm SMMU.

Abuse of realms. The adversary may request a realm to direct access or tamper with the sensitive data and code in other realms. However, it fails to bypass the memory isolation on CPU side (e.g., Stage-2 translation in RMM or CPU GPC in Shelter [80]). Moreover, the adversary may launch confidential GPU applications to achieve the same attack from the GPU, but this is restricted by GPU SMMU GPC.

C. RQ3: Evaluations on GPU Benchmarks

Experimental setup. To measure the performance of our prototype, we follow best practices from previous GPU TEE studies [39] and select six applications from the well-recognized Rodinia benchmark suites [37]. We select these benchmarks because they cover a wide range of use cases for Arm GPUs, including a lightweight KNN, three medium-weight applications (LUD, PF, and H3D), and two heavy-weight applications (GS and LMD). We report the problem size and memory consumption in Table V. We introduce two minimal modifications to the benchmark applications to perform confidential computing. First, we replace the input sensitive data and the corresponding data buffer descriptions with hash values. This is because we disallow the untrusted software stack to directly operate the sensitive data, while we perform these operations in the Monitor based on the descriptions. Note that the realm user has transferred the sensitive data to the realms before confidential GPU computing. Second, we prevent a GPU buffer from sharing the same physical page, which is the minimal granule in GPT, with other buffers or metadata. To achieve this goal, we introduce several wrapped OpenCL APIs to reside the GPU buffers in isolated physical pages. Compared to the problem size of the benchmark, our modification only introduces minimal memory consumption during computation.

Based on these configurations, we perform confidential GPU computing on the six applications and compare the performance with that of normal GPU computation. To better understand the performance, we also provide a breakdown of the performance overhead of our prototype into four components: **GPU**, which is the execution time on the GPU hardware; **GStack**, which shows the performance overhead for the untrusted GPU software stack; **STask**, which reports the overhead for the shadow task mechanism; and **GProtect**, the latency induced by protecting the GPU environment.

Performance analysis. Figure 6 shows the performance comparison between our CAGE prototype and the native system. It indicates that CAGE introduces 0.58% – 5.31% overhead in the six Rodinia benchmark applications. Furthermore, we provide a detailed breakdown of the performance overhead in Table VI. GPU computation (**GPU**) contributes most to the overhead in most benchmarks except the lightweight KNN application. Moreover, we observe that the overhead in the shadow task mechanism (**STask**) is determined by the number of tasks and the memory size. It is also reflected in our results: The shadow task mechanism incurs more overhead in GS (which consists of 4096 tasks) than PF (only 5 tasks), though their memory consumption is similar. Next, our mechanism introduces orders of magnitude of overhead on the large-sized application (e.g., LMD) compared to the lightweight one (e.g., KNN). Lastly, Table VI reports that the protection on the GPU environment (**GProtect**) introduces the least latency in the entire application. The major reason is that our protection is mainly achieved by varied GPCs with optimized GPT maintenance and integrity verification. At the same time, we do not introduce additional security operations on GPU memory (e.g., cryptographic operations). Note that the execution time on both CAGE prototype and the native system may slightly increase in real CCA-supported devices since we only configure GPTs instead of performing GPC on MMU/SMMU.

Table VI: Breakdown (ms) of overhead of CAGE prototype.

	GPU	GStack	STask	GProtect
KNN	0.31	58.36	0.62	0.02
PF	2788.71	271.83	82.02	0.65
LUD	3362.08	431.86	48.15	6.03
H3D	4902.95	464.44	86.82	9.70
LMD	13474.49	111.62	153.64	1.02
GS	47237.44	2991.07	351.17	71.02

Comparison to state-of-the art. We further discuss the performance comparison between CAGE and state-of-the-art. However, most state-of-the-art GPU TEEs lack source code [31], [52], [61], [73], [76], [81] and are implemented on different platforms (e.g., Cronus [54] on QEMU, GR-T [70] on Hikey960, and StrongBox [39] on Juno R2), incurring costly re-implementation efforts. Thus, we compare CAGE and StrongBox on the same Rodinia benchmarks. Figure 6 shows that CAGE introduces slightly lower overhead on these benchmarks. The major reason is that StrongBox leverages the untrusted GPU driver to directly operate the memory with sensitive data, thus it introduces additional cryptographic operations to protect these data. However, CAGE does not require these operations due to the shadow task mechanism.

D. RQ4: Evaluation on Neural Network Models

To verify whether CAGE has the capability to secure the sensitive data in complex GPU computation scenarios, we next perform machine learning inference on three neural network models (LeNet-5 [56], SqueezeNet [50], and MobileNet-v1 [46]), which cover a wide range of use case (detailed in Table VII). Compared with the protection on general computing, we additionally guarantee data confidentiality for the weights and bias parameters since they directly influence the execution results during the inference process. We export the model code to the untrusted GPU software, but protect it during GPU computation. Table VII reports the performance comparison between CAGE and the native system with a detailed breakdown. It shows that CAGE introduces 1.24% –

Table VII: Problem size and execution time of the selected neural network models.

	Tasks	Memory	Data Buffers	Vanilla	CAGE				
					Total	GPU	GStack	STask	GProtect
LeNet-5	6	242.20 KB	13	332.58ms	336.70ms	1.30ms	333.91ms	1.38ms	0.11ms
SqueezeNet	30	13.22 MB	67	788.73ms	812.62ms	346.49ms	453.42ms	12.27ms	0.44ms
MobileNet-v1	47	37.27 MB	167	898.62ms	967.30ms	532.27ms	359.50ms	73.85ms	1.68ms

7.64% performance overhead on the selected models. Although SqueezeNet and MobileNet-v1 require to protect more physical memory and data buffers than LeNet-5, their additional performance overhead is not sharply increased. The major reason is that our shadow task mechanism (**STask**) only replaces and secures the processed GPU buffers in the current GPU task instead of all the GPU buffers in the application.

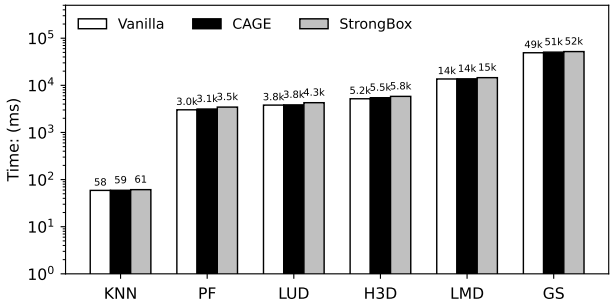

Figure 6: Performance overhead on six Rodinia benchmarks.

E. RQ5: Evaluation on GPT Optimization

Synchronizing CPU and peripheral GPTs. CAGE optimizes the GPT synchronization in GPU memory protection. Rather than modifying the access control on CPU GPT and every untrusted peripheral GPT, we only maintain a unified sub-level GPT that represents the security view of the GPU memory. To prove the effectiveness of our optimization mechanism, we evaluate it with (1) different sizes of protected memory regions and (2) varied number of GPTs that share the sub-level GPT. As shown in Figure 7, CAGE mitigates 87.40% – 87.50% of performance overhead in GPT synchronization when we scale up the protected memory size, and 50.01% – 93.65% of performance overhead in the varied number of GPTs. It shows that CAGE effectively optimizes the overhead of updating GPT entries to protect both small-size and large-size physical memory. Moreover, the increasing number of GPTs only introduces a small additional overhead to our optimization mechanism. This is because the new GPTs still share the same sub-level table with previous GPTs, and meanwhile flushing the TLB of the new peripheral GPCs incurs minimal overhead.

Initializing GPU GPT for each realm. We further compare our GPU GPT initialization mechanism with the native solution. Since the existing TF-A has yet to provide the specific GPT configuration for GPU, we emulate it by invoking the CPU GPT initialization APIs in TF-A v2.8 [29]. As for the GPT configuration, we describe half of the main memory with 1GB-level granule descriptors and the remaining memory with 4KB-level granule descriptors. We measure the performance overhead of the native solution and CAGE with four different size of the main memory (2GB, 4GB, 8GB, and 16GB). As shown in Figure 8, we mitigate 84.63% – 96.55% performance overhead of GPU GPT initialization in these configurations. During initialization, increasing the size of main memory unavoidably generates more latency, while we still reduce a large portion of the overhead compared to the native solution.

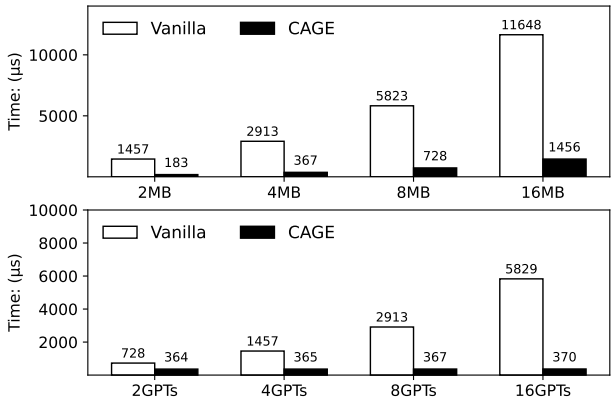

Figure 7: Performance comparison between the non-optimized mechanism and CAGE in GPT synchronization. Note that the upper benchmark configures the GPT number as 8 and the lower benchmark sets the size of the protected region as 4MB.

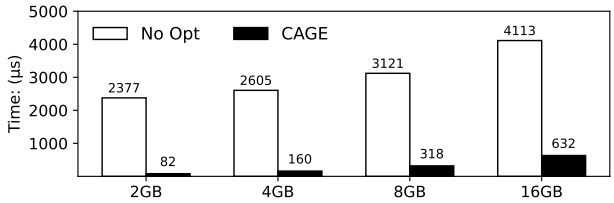

Figure 8: Performance improvement in GPU GPT initialization with four different size of main memory.

VII. Discussion

Physical adversaries. Currently, CAGE mainly focuses on software adversaries since our prototype is based on the latest RME-supported FVP model which lacks hardware-assisted memory encryption support. Nevertheless, we discuss potential solutions against several physical threats.

By leveraging the Memory Protection Engine (MPE) built in future CCA devices, CAGE can protect realms' sensitive data from cold-boot attacks [78]. The MPE automatically encrypts/decrypts the data that are transferred between the system bus and DRAM, preventing the adversary from directly accessing the plaintext DRAM contents. Moreover, the adversary may launch rollback attacks on GPU memory, while we can use the monotonic counters [31], [62] to record data states and guarantee the freshness of GPU memory. In addition, the adversary may attempt to launch bus probing or hardware man-in-the-middle attacks when CAGE accesses the GPU and SMMU MMIO, though it is unclear how these attacks can be deployed on the MPE-supported devices. In the future, we can apply other techniques (e.g., encrypting the access patterns on memory bus [1], [30]) to secure the bus traffic from these attacks.

Arm xPU TEEs. Besides the GPU hardware, we consider CAGE can potentially support other widely-used accelerators such as NPUs and DPUs. As with GPUs, these xPU devices delegate untrusted software components (e.g., Arm Ethos-N driver [13]) to manage applications such as memory allocation and xPU task scheduling. In addition, Arm xPUs also share the unified memory with CPU and peripherals so that they are threatened by the same scope of adversaries as GPUs. To secure xPU computing, previous studies [47], [57], [66] leverage TrustZone or customized hardware to construct xPU TEEs, which can benefit CAGE to extend CCA on Arm xPUs.

Mitigating performance overhead CAGE protects GPU MMIO by configuring GPCs for CPU, GPU, and untrusted peripherals. Nevertheless, it can be optimized by configuring a *Completer-side PAS filter* [25]. As a private module for a peripheral, this filter decides which type of request (e.g., *normal world* or *root world* request) can access the peripheral MMIO registers. Thus, we can configure the GPU filter only once to confine accesses from the CPU and untrusted peripherals. However, since it is uncertain how Arm FVP and real-world GPUs support this filter, we instead configure every GPC (i.e., GPC on CPU, GPU, and other peripherals) to restrict the illegal accesses to GPU MMIO. We suggest Arm provides detailed support for this filter in the future.

Suggestions for future Arm CCA CAGE configures GPCs on both MMU and SMMU to secure GPU computing. During the configuration of SMMU, we find that several real-world Arm devices [33], [68], [72] implement an integrated SMMU to manage the memory access from multiple peripherals. To distinguish these accesses, SMMU provides each peripheral with a unique `StreamID`, from which SMMU fetches the corresponding table (e.g., address translation table) to check access. The `StreamID`-based access control is supported in traditional mechanisms such as the Stage-2 translation, while it has yet to support the latest GPC. SMMU only provides a single MMIO region to set the GPT, so that all the connected peripherals share the same view of main memory in GPC. In the future, we suggest that Arm introduce `StreamID` to distinguish different SMMU GPCs, or allows SMMUs to provide each peripheral with a unique `Root control page` to configure its own peripheral GPC.

VIII. Related Work

Arm confidential computing. Studies [32], [44], [48], [53], [75] have leveraged Arm TrustZone to simulate confidential computing environments in *normal world* and *secure world*. However, these security features are vulnerable to the *secure world* attacker [35] and new yet-unrealized adversaries in Arm CCA. Furthermore, these studies consider GPUs as untrusted peripherals that do not provide any support for confidential computing. On the other hand, recent studies focus on exploring Arm CCA's security feature to achieve confidential computing. Shelter [80] secures application enclaves from privileged software by constructing multiple GPTs. Li *et al.* [59] build realms for confidential computing and verifies the security of these regions. Twinvisor [58] provides confidential virtual machines with simulated Arm CCA features. ReZone [35] addresses the excessive privilege of secure OS to create an isolated execution environment. However, these studies have yet to ensure confidential GPU computing.

GPU TEE. Researchers have explored GPU TEEs to guarantee data security on GPUs. To achieve secure communication between the user and the GPU, typical GPU TEEs leverage a CPU-side TEE to transfer sensitive data to the GPU and control access to GPU MMIO. For instance, HIX [52] and HoneyComb [61] create GPU enclaves and secure GPU MMIO with Intel SGX and AMD SEV-SNP, respectively. CURE [31] secures the GPU enclaves by introducing an access filter in the system bus. HETEE [81] introduces a security controller on FPGA hardware to distribute workloads into isolated execution environment with GPU resources. However, migrating non-Arm GPU TEEs to Arm devices may not be possible due to variations in architecture and organization. Recent works have also proposed several Arm-based GPU TEEs [39], [54], [70], [77]. These GPU TEEs leverage traditional Arm security features (e.g., TrustZone and Arm secure/non-secure virtualization) to achieve the same protection on GPUs, while they have yet to provide sufficient protection against new adversaries in Arm CCA. Lastly, few GPU TEEs directly construct TEE inside GPU without the need for CPU-side TEE, such as Graviton [76] and NVIDIA H100 GPU [67]. These GPU TEEs construct a channel between the untrusted Host and GPU, with which they monitor the command submission and data transfer. However, data protection in these GPU TEEs mainly depends on the physical memory isolation between GPU and CPU-side adversary. Thus, they are infeasible for the unified-memory GPU that shares the memory with CPU.

IX. Conclusion

In this paper, we present CAGE to extend GPU support on Arm CCA. Our design follows the Arm CCA's realm-style architecture in GPU computing, achieved through the novel shadow task mechanism. We ensure data security by leveraging the GPC on the CPU, GPU and untrusted peripherals, achieving two-way isolation between realm's GPU execution environment and the other components. To maintain multiple GPTs, we also present two optimization techniques on GPT synchronization and initialization. CAGE's design and implementation require no hardware changes, ensuring high compatibility with next-generation Arm devices. To demonstrate the functionality of our approach, we prototype CAGE on an official software-simulated platform. We then port this prototype to an off-the-shelf development board, and evaluate it with rigorous benchmarks. Results show that CAGE provides effective support for confidential GPU computing with an average of 2.45% performance overhead.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work is partly supported by the National Natural Science Foundation of China under Grant No. 62372218, No. 62002151 and No. 62102175, and Shenzhen Science and Technology Program under Grant No. SGDX20201103095408029, and HK RGC General Research Fund No. PolyU 15220020, and HK RGC Collaborative Research Fund No. C2004-21GF, and the Research Institute for Artificial Intelligence of Things, The Hong Kong Polytechnic University. This work is also in part supported by Ant Group Research Fund.

REFERENCES

[1] S. Aga and S. Narayanasamy, "Invisimem: Smart memory defenses for memory bus side channel," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 94–106.

[2] AlDanial, "cloc," https://github.com/AlDanial/cloc, 2021.

[3] AMD, "AMD Secure Encrypted Virtualization (SEV)," https://developer.amd.com/sev/, 2022.

[4] ——, "AMD Radeon™ RX Graphics Cards," https://www.amd.com/en/graphics/radeon-rx-graphics, 2023.

[5] ——, "Confidential Computing Solution Brief," https://www.amd.com/en/processors/epyc-confidential-computing-cloud, 2023.

[6] Amlogic, Inc., "S905 Datasheet," https://dn.odroid.com/S905/DataSheet/S905_Public_Datasheet_V1.1.4.pdf, 2016.

[7] Apple, "Discover Metal enhancements for A14 Bionic," https://developer.apple.com/videos/play/tech-talks/10858/, 2022.

[8] ARM, "ARM Security Technology Building a Secure System using TrustZone Technology," https://developer.arm.com/documentation/PRD29-GENC-009492/latest/, 2009.

[9] ——, "ARM CoreLink TZC-400 TrustZone Address Space Controller Technical Reference Manual," https://developer.arm.com/documentation/ddi0504/latest/, 2014.

[10] ——, "Juno r2 ARM Development Platform SoC," https://developer.arm.com/documentation/ddi0515/latest, 2016.

[11] ——, "Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile," https://developer.arm.com/documentation/ddi0487/latest/, 2022.

[12] ——, "Arm Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile," https://developer.arm.com/documentation/ddi0608/latest, 2022.

[13] ——, "Arm Ethos-N Driver Stack," https://github.com/ARM-software/ethos-n-driver-stack, 2022.

[14] ——, "Azure confidential computing," https://developer.arm.com/documentation/den0125/0200/, 2022.

[15] ——, "Mali Texture Compression Tool," https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-texture-compression-tool, 2022.

[16] ——, "Open Source Mali Midgard GPU Kernel Drivers," https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-drivers/midgard-kernel, 2022.

[17] ——, "OpenCL," https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-drivers/user-space, 2022.

[18] ——, "VR best practice," https://developer.arm.com/solutions/graphics-and-gaming/developer-guides/vr-best-practice, 2022.

[19] ——, "Arm CCA Security Model 1.0," https://developer.arm.com/documentation/DEN0096/latest/, 2023.

[20] ——, "Arm Confidential Compute Architecture," arm.com/architecture/security-features/arm-confidential-compute-architecture, 2023.

[21] ——, "Arm Mali Graphics Processing Units (GPUs)," https://developer.arm.com/ip-products/graphics-and-multimedia/mali-gpus, 2023.

[22] ——, "Arm Realm Management Extension (RME) System Architecture," https://developer.arm.com/documentation/den0129/latest/, 2023.

[23] ——, "Arm System Memory Management Unit Architecture Specification," https://developer.arm.com/documentation/ihi0070/latest/, 2023.

[24] ——, "Fixed Virtual Platforms," https://www.arm.com/products/development-tools/simulation/fixed-virtual-platforms, 2023.

[25] ——, "Introducing Arm Confidential Compute Architecture guide," https://developer.arm.com/documentation/den0125/latest/, 2023.

[26] ——, "Mali-G76," https://developer.arm.com/documentation/100964/1121/Fast-Models-components/Media-components/Mali-G76, 2023.

[27] ——, "PrimeCell Infrastructure AMBA 3 TrustZone Protection Controller (BP147)," https://developer.arm.com/documentation/dto0015/latest/, 2023.

[28] ——, "TF-RMM," https://git.trustedfirmware.org/TF-RMM/tf-rmm.git/tag/?h=tf-rmm-v0.2.0, 2023.

[29] ——, "Trusted Firmware-A release v2.8," https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git/tag/?h=v2.8, 2023.

[30] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 107–119.

[31] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "CURE: A Security Architecture with CUstomizable and Resilient Enclaves," in *Proceedings of the 30th USENIX Security Symposium*, 2021, pp. 1073–1090.

[32] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing Trustzone with User-space Enclaves," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, 2019.

[33] Broadcom, "Product Brief: Stingray PS225," https://docs.broadcom.com/doc/PS225-PB, 2022.

[34] Q. Cao, N. Balasubramanian, and A. Balasubramanian, "MobiRNN: Efficient Recurrent Neural Network Execution on Mobile GPU," in *Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications*, 2017, pp. 1–6.

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

[36] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems," in *Proceedings of the 41st IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 1416–1432.

[37] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the 12nd IEEE International Symposium on Workload Characterization*. Ieee, 2009, pp. 44–54.

[38] S. Checkoway and H. Shacham, "Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 253–264, 2013.

[39] Y. Deng, C. Wang, S. Yu, S. Liu, Z. Ning, K. Leach, J. Li, S. Yan, Z. He, J. Cao *et al.*, "Strongbox: A GPU TEE on Arm Endpoints," in *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 769–783.

[40] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[41] R. M. Farkhani, M. Ahmadi, and L. Lu, "PTAuth: Temporal Memory Safety via Robust Points-to Authentication," in *Proceedings of the 30th USENIX Security Symposium*, 2021, pp. 1037–1054.

[42] FuZhou Rockchip Electronics Co., Ltd., "Rockchip RK3288 Technical Reference Manual Part1," http://opensource.rock-chips.com/images/8/8f/Rockchip_RK3288_TRM_V1.2_Part1-20170321.pdf, 2017.

[43] Google, "Confidential Computing," https://cloud.google.com/confidential-computing, 2023.

[44] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017, pp. 488–501.

[45] C. Holmes, D. Mawhirter, Y. He, F. Yan, and B. Wu, "GRNN: Low-Latency and Scalable RNN Inference on GPUs," in *Proceedings of the 14th European Conference on Computer Systems*, 2019, pp. 1–16.

[46] 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.

[47] W. Hua, M. Umar, Z. Zhang, and G. E. Suh, "GuardNN: Secure Accelerator Architecture for Privacy-Preserving Deep Learning," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 349–354.

[48] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM TrustZone," in *Proceedings of the 26th USENIX Security Symposium*, 2017, pp. 541–556.

[49] G. D. Hunt, R. Pai, M. V. Le, H. Jamjoom, S. Bhattiprolu, R. Boivie, L. Dufour, B. Frey, M. Kapur, K. A. Goldman *et al.*, "Confidential computing for OpenPOWER," in *Proceedings of the 16th European Conference on Computer Systems*, 2021, pp. 294–310.

[50] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and $<$ 0.5 MB model size," *arXiv preprint arXiv:1602.07360*, 2016.

[51] Intel Corporation, "Intel Trust Domain Extensions," https://cdrdv2.intel.com/v1/dl/getContent/690419, 2022.

[52] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous Isolated Execution for Commodity GPUs," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 455–468.

[53] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. B. Kang, "PrivateZone: Providing a Private Execution Environment Using ARM TrustZone," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 797–810, 2016.

[54] J. Jiang, J. Qi, T. Shen, X. Chen, S. Zhao, S. Wang, L. Chen, G. Zhang, X. Luo, and H. Cui, "CRONUS: Fault-isolated, Secure and High-performance Heterogeneous Computing for Trusted Execution Environment," in *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2022, pp. 124–143.

[55] S. S. Latifi Oskouei, H. Golestani, M. Hashemi, and S. Ghiasi, "CNNdroid: GPU-Accelerated Execution of Trained Deep Convolutional Neural Networks on Android," in *Proceedings of the 24th ACM International Conference on Multimedia*, 2016, pp. 1201–1205.

[56] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[57] S. Lee, J. Kim, S. Na, J. Park, and J. Huh, "TNPU: Supporting Trusted Execution with Tree-less Integrity Protection for Neural Processing Unit," in *Proceedings of the 28th IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 2022, pp. 229–243.

[58] 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 SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 638–654.

[59] 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 16th USENIX Symposium on Operating Systems Design and Implementation*, 2022, pp. 465–484.

[60] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "PAC it up: Towards Pointer Integrity using ARM Pointer Authentication," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 177–194.

[61] H. Mai, J. Zhao, C. Kozyrakis, M. Gao, H. Zheng, Q. Li, Z. Liu, C. Wang, H. Cui, and X. Feng, "Honeycomb: An Secure, Efficient GPU Execution Environment with Minimal TCB," in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[62] A. Martin, C. Lian, F. Gregor, R. Krahn, V. Schiavoni, P. Felber, and C. Fetzer, "ADAM-CS: Advanced Asynchronous Monotonic Counter Service," in *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2021, pp. 426–437.

[63] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "WaTZ: A Trusted WebAssembly Runtime Environment with Remote Attestation for TrustZone," in *Proceedings of the 42nd IEEE International Conference on Distributed Computing Systems*. IEEE, 2022, pp. 1177–1189.

[64] A. Menezes and S. S. A. Vanstone, *Guide to elliptic curve cryptography*. New York: Springer, 2004.

[65] Microsoft, "Azure confidential computing," https://azure.microsoft.com/en-us/solutions/confidential-compute/, 2023.

[66] ——, "Confidential Computing within an AI Accelerator," https://www.microsoft.com/en-us/research/publication/confidential-computing-within-an-ai-accelerator/, 2023.

[67] NVIDIA, "NVIDIA CONFIDENTIAL COMPUTING," https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/, 2022.

[68] ——, "Tegra X1," https://developer.nvidia.com/content/tegra-x1/, 2022.

[69] ——, "Graphics Cards," https://www.nvidia.com/en-us/geforce/graphics-cards/, 2023.

[70] H. Park and F. X. Lin, "Safe and Practical GPU Computation in TrustZone," in *Proceedings of the 18th European Conference on Computer Systems*, 2023, pp. 505–520.

[71] Qualcomm, "Adreno Graphics Processing Units," https://developer.qualcomm.com/software/adreno-gpu-sdk/gpu/, 2022.

[72] ——, "The ARM SMMU and the Adreno GPU," https://static.linaro.org/connect/lvc20/presentations/LVC20-309-0.pdf, 2022.

[73] S. Sridhara, A. Bertschi, B. Schlüter, M. Kuhne, F. Aliberti, and S. Shinde, "ACAI: Extending Arm Confidential Computing Architecture Protection from CPUs to Accelerators," *arXiv preprint arXiv:2305.15986*, 2023.

[74] STMicroelectronics, "GPU device tree configuration," https://wiki.st.com/stm32mpu/wiki/GPU_device_tree_configuration, 2022.

[75] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 367–378.

[76] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted Execution Environments on GPUs," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, 2018, pp. 681–696.

[77] J. Wang, Y. Wang, and Z. Ning, "Secure and Timely GPU Execution in Cyber-physical Systems," in *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[78] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin, "Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors," in *Proceedings of the 23rd IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2017, pp. 313–324.

[79] J. Z. Yu, S. Shinde, T. E. Carlson, and P. Saxena, "Elasticlave: An Efficient Memory Model for Enclaves," in *Proceedings of the 31st USENIX Security Symposium*, 2022, pp. 4111–4128.

[80] 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 32nd USENIX Security Symposium*, 2023.

[81] J. Zhu, R. Hou, X. Wang, W. Wang, J. Cao, B. Zhao, Z. Wang, Y. Zhang, J. Ying, L. Zhang *et al.*, "Enabling Rack-scale Confidential Computing using Heterogeneous Trusted Execution Environment," in *Proceedings of the 41st IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 1450–1465.