Building Confidential Accelerator Computing Environment for Arm CCA

Chenxu Wang, Kun Lu, Fengwei Zhang†, Yunjie Deng, Kevin Leach, Jiannong Cao, Zhenyu Ning, Shoumeng Yan, Tao Wei, and Zhengyu He

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 regions, known as realms, to ensure data confidentiality and integrity in security-sensitive tasks. However, hardware and firmware supporting confidential accelerator workloads remain unavailable. Moreover, due to incompatible hardware or large trusted computing base (TCB) size, existing studies for protecting acceleration are unsuitable for CCA's realm-style architecture. Thus, there is a need to complement existing Arm CCA capabilities with accelerator support.

We present CAGE to support confidential accelerator computing for Arm CCA, ensuring data security with CCA's existing security features. To adapt the accelerator workflow to CCA's realm-style architecture, CAGE proposes a novel shadow task mechanism to manage confidential accelerator applications flexibly. Additionally, CAGE leverages the memory isolation mechanism in Arm CCA to protect data confidentiality and integrity from the strong adversary. CAGE also optimizes security operations in memory isolation to mitigate performance overhead. Without hardware changes, we design and implement CAGE on two types of accelerators: Unified-memory GPU and generic FPGA. Our evaluation shows that CAGE effectively provides confidential accelerator support for Arm CCA with moderate overhead.

Index Terms—Arm CCA, Confidential Accelerator Computing

1 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 that support confidential computing secure sensitive data from all unauthorized access, including illegal applications, untrusted clients, and even cloud providers [1], [2], [3], [4]. Major processor manufacturers have proposed corresponding hardware primitives to support confidential computing, such as Intel's Trust Domain Extensions (TDX) [5], AMD's Secure Encrypted Virtualization (SEV) [6], and IBM's Protected Execution Facility (PEF) [7].

Chenxu Wang is with Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China, and also with Department of Computing, The Hong Kong Polytechnic University, China. E-mail: 12150073@mail.sustech.edu.cn.
Kun Lu is with Department of Computer Science and Engineering, Southern University of Science and Technology, China. E-mail: 12232282@mail.sustech.edu.cn
Fengwei Zhang is with Department of Computer Science and Engineering, and Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China. E-mail: zhangfw@sustech.edu.cn
Yunjie Deng is with Research Institute of Trustworthy Autonomous Systems, and Department of Computer Science and Engineering, Southern University of Science and Technology, China. E-mail: 12032869@mail.sustech.edu.cn.
Kevin Leach is with Institute for Software Integrated Systems, Vanderbilt University, USA. E-mail: kevin.leach@vanderbilt.edu.
Jiannong Cao is with Department of Computing, The Hong Kong Polytechnic University, China E-mail: csjcao@comp.polyu.edu.hk
Zhenyu Ning is with Hunan University, China, and also with Department of Computer Science and Engineering, Southern University of Science and Technology, China. E-mail: zning@hnu.edu.cn.
Shoumeng Yan, Tao wei and Zhengyu He are with Ant Group, China. E-mail: shoumeng.ysm@antgroup.com, lenx.wei@antgroup.com, zhengyu.he@antgroup.com

Arm also proposed the Confidential Compute Architecture (CCA) [8] and its hardware security primitive—*Realm Management Extensions (RME)* [9]—to support confidential computing in next-generation [10] 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 during 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 accelerators [11], [12], [13], [14], which are widely used to accelerate the general-, high-performance, and artificial intelligence computing scenarios [15], [16], [17], [18], [19], have only recently been proposed. However, such support, called RME Device Assignment (RME-DA) [20], is currently a high-level concept without a complete hardware implementation. The CCA design characterizes generic accelerators (e.g., GPU and FPGA) as untrusted peripherals on which data security is not guaranteed. As a result, if realms use accelerators for computation, their sensitive data can be accessed by an adversary who controls the accelerators via compromised software (e.g., a compromised accelerator driver or programming model).

To address the data security problem in confidential accelerator computing, a recent study [21] proposes a similar design to RME-DA, but its protection mechanism requires non-trivial modification on hypervisor software. Moreover, it introduces heavyweight accelerator software to the realm's TCB. Previous works have also developed Trusted Execution Environments (TEEs) for accelerators to address these issues. However, most solutions [22], [23], [24], [25], [26], [27], [28], [29], [30] rely on hardware security primitives that are not suitable for next-generation Arm devices (e.g., customized hardware, Intel-based security primitives, or traditional Arm security hardware), incurring tremendous challenges in migration and mitigation. For example, HIX [26] leverages Intel Software Guard eXtensions (SGX), which is not supported in Arm devices, and StrongBox [28] trusts *secure world* software, which is untrusted in Arm CCA.

We present CAGE, a framework that supports confidential acceleration on next-generation Arm devices. By leveraging the hardware security primitives (i.e., the RME) in Arm CCA, CAGE secures confidential computing on the widely-deployed accelerators [12], [13], [14], [31], [32], [33]. We design CAGE based on two critical observations (detailed in §2): (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 acceleration from untrusted software (e.g., the OS, hypervisor, and *secure world* components) and peripherals. (2) Most of the components in the accelerator software, 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 accelerator software to schedule confidential acceleration applications without direct access to the sensitive data. We use these key insights to design the workflow of CAGE, supporting confidential acceleration on Arm CCA.

The design of CAGE faces three key challenges. **C1:** The accelerator software in the host is not intended to manage applications in realms. Since accelerator software requires frequent interaction with applications, 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 accelerator software stack to create and manage stub accelerator applications corresponding to real accelerator applications in realms, then verifies and synchronizes these operations to the real accelerator applications in the Monitor (§4.2). Note that our shadow task mechanism is a generic confidential acceleration mechanism, adapting the computing process on GPUs, FPGAs, and other accelerators. **C2:** Without the latest RME-DA support, generic Arm CCA design regards the accelerators as *normal world* peripherals and disallows them to access the realm. Thus, it is neither intended to achieve the same CPU-side isolation on accelerators nor protect the accelerator execution environment from various attacks. To address this challenge, CAGE proposes a two-way isolation mechanism to protect the accelerator environment. By leveraging the existing GPC for the CPU and peripherals, CAGE provides different accelerator memory views for each realm during confidential acceleration while securing the accelerator environment from the untrusted software and peripherals (§4.3). Our GPC-based protection ensures data security on accelerators with varied hardware architectures. Such hardware variation influences data transfer protection and access control in the accelerator computing environment. **C3:** Based on the above accelerator protection mechanism, CAGE must create multiple Granule Protection Tables (GPTs) for the corresponding GPCs and synchronize these GPTs during the accelerator environment protection, generating additional performance overhead. For this challenge, we propose optimization techniques for GPT maintenance, especially in synchronizing GPTs for untrusted components and initializing accelerator GPTs for realms (§4.4).

We prototype CAGE on two official platforms: (1) The Arm Fixed Virtual Platform (FVP) [34] with software-simulated Arm CCA feature and (2) the Arm Juno R2 development board [35] with an embedded Mali-T624 GPU and PCIe-connected Xilinx Virtex Ultrascale+ VCU118 FPGA [36]. Our implementation is verified on two types of accelerators: the Arm Mali GPU [12] and Xilinx FPGA [14]. Our prototype introduces 1,301 lines of code (LoC) for GPU and additional 140 LoC for FPGA, creating a thin TCB. Moreover, we evaluate performance by running the Rodinia GPU benchmark suite [37] and several FPGA benchmarks. For confidential GPU and FPGA computing, we also discuss the security of CAGE against an assumed adversary. Our results indicate that CAGE securely provides confidential acceleration for Arm CCA with a moderate (9.61% – 16.30%) performance overhead.

We claim the following contributions to this work:

- We present CAGE, which provides confidential acceleration support for Arm CCA. In confidential acceleration (e.g., GPU and FPGA computing), CAGE secures the sensitive data from the strong adversary assumed in Arm CCA.
- We prototype CAGE on the official Arm emulator and a real-world development board without hardware changes. We will share and maintain the source code[1] to benefit the Arm community.
- We comprehensively evaluate CAGE with respect to its performance and security, measuring the confidentia computing on Arm GPUs and Xilinx FPGA. The results indicate that CAGE effectively secures sensitive data and code with 9.61% – 16.30% performance overhead on several indicative benchmarks.

This paper is an extended version of our previous work [38] accepted in Network and Distributed System Security Symposium 2024. Based on that work, we further extend our confidential acceleration mechanism from the unified-memory GPU to support generic FPGAs, which are dedicated-memory devices that are widely used in generic and artificial intelligence acceleration on Arm-based cloud platforms [29], [39]. We implement an FPGA-oriented prototype on Arm Juno R2 development board with a PCIe-connected Xilinx Virtex Ultrascale+ VCU118 FPGA [36] (§7.1), on which we run a set of FPGA benchmarks (§7.2) and analyze its security (§7.3). We also discuss several solutions to improve the performance and security (§8). Our results show that CAGE effectively supports generic FPGA computing on Arm CCA with a moderate performance overhead.

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

2 Background

2.1 Arm CCA

In the latest Armv9 architecture [10], Arm supports confidential computing through two enabling technologies: (1) the Confidential Compute Architecture (CCA) [8], and (2) an associated hardware primitive, called Realm Management Extensions (RME) [9]. CCA introduces a new execution state, called *realm world*, to run multiple confidential realms. These realms are isolated from the other execution states (i.e., *normal world* for traditional software and *secure world* for secure software) 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* 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 main memory. Specifically, when software 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. For security, CCA advises storing the GPT in *root* PAS and only permits the Monitor to configure GPC registers.

Key observation. Compared to the older Arm TrustZone, Arm CCA secures the highest-privilege Monitor in a hardware-isolated *root world* and provides novel hardware-assisted mechanisms for memory isolation. As a result, CAGE deploys its security modules in the Monitor to additionally secure acceleration, such as controlling access to accelerator memory and accelerator registers.

2.2 Accelerators and SMMU

Accelerators (e.g., GPU, FPGA, and NPU) are widely used in Arm endpoints and clouds to process generic and high-performance computing tasks. To perform acceleration, the host controls a set of accelerator software packages, including the kernel-layer driver (e.g., Midgard [40] GPU driver and FPGA XDMA driver [41]) and the user-layer runtime (e.g., OpenCL [42] libraries), to transfer data and code and to execute commands. Specifically, the accelerator software manages (1) the accelerator computation environment and (2) the interaction with the accelerator hardware. To prepare the execution environment, the accelerator software allocates physical memory on the host, creates buffers, and loads critical components (e.g., code, data, metadata, and page table) into the memory. As for interacting with the accelerator hardware, the accelerator software schedules the execution order and submits commands via Memory-Mapped Input/Output (MMIO). Once the accelerator hardware receives the commands, it accesses the critical components via Direct Memory Access (DMA) and performs compututation. When the accelerator terminates computation, the accelerator software fetches the execution results and restores the environment.

Since accelerators and other peripherals can directly access the main memory. Arm introduced the System Memory Management Unit (SMMU) to manage DMA-capable peripherals. In Arm designs, accelerators (e.g., Arm Mali GPUs [12], NVIDIA Tegra X1 Maxwell GPUs [31], and Arm Ethos-N NPU [13]) and other peripherals are usually controlled by an SMMU. Like the CPU MMU, SMMU supports controlling access from peripherals to system memory. To enable access control, privileged software configures SMMU registers (e.g., page table registers and translation configuration registers) via MMIO. Besides address translation, SMMU supports GPC in the latest Arm CCA [43]. To protect the SMMU GPC, CCA introduces additional SMMU MMIO registers accessible only 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 accelerators can be adapted to Arm CCA's realm-style architecture. On the one hand, the accelerators' software performs most of its functions without accessing sensitive data. Therefore, we 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, allowing CAGE to isolate the accelerator computing environment.

3 Threat Model and Assumptions

Following Arm CCA, we assume a strong adversary who controls the entire software stack in both *normal world* and *secure world*, including the accelerator software, untrusted OS and hypervisor, and the same layer software in *secure world*. The adversary aims to leak or tamper with the sensitive data within or controller by accelerator tasks using various attack methods, such as accessing the realm memory from the CPU and untrusted peripherals or compromising the accelerator software stack. The adversary may perform several physical attacks (e.g., cold-boot attacks [44]) on memory, which can be addressed by CCA's memory encryption support [45] and orthogonal works [46], [47]. We also discuss this in §8. 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 supports secure boot, remote attestation, and constructing a secure communication channel with a realm user. Following with the state-of-the-art [24], [28], CAGE assumes the accelerators' hardware and firmware are not malicious/buggy. The SoC or third-party accelerators can leverage existing solutions [48], [49] to support CAGE verify their hardware and firmware, ensuring an authentic computing environment. However, memory access from the accelerator is restricted by GPC on RME-supported SMMU [43], preventing the accelerator from modifying CAGE's configurations (e.g., accelerator GPTs) in the Monitor. On the software side, we trust the Monitor since its firmware is securely verified and loaded during the secure boot.

4 Design

CAGE provides support for hardware accelerators for Arm-based confidential computing. In this section, we describe CAGE with respect to 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) to manage confidential acceleration. More concretely, CAGE must delegate the complex but data-independent functions (e.g., memory management and task scheduling) to the untrusted accelerator 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 acceleration, 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 accelerator execution workflow, CAGE must not generate a high performance overhead during confidential acceleration.

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 accelerators without introducing hardware changes.

4.1 CAGE Overview

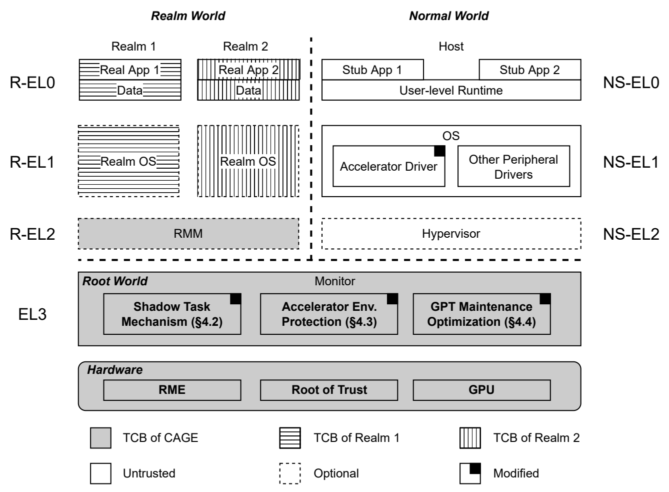


Figure 1: CAGE architecture overview. Note that the *secure world* is omitted since CAGE does not modify the software components in this world.

We envision scenarios where users or third-party developers persistently request a realm to store and execute the confidential accelerator applications. The realm user transfers sensitive data through a secure and encrypted channel [50], [51], [52] to the requested realm. To follow Arm CCA's realm-style architecture, the user provides data buffer descriptions for the untrusted accelerator software to construct a stub execution environment, including metadata and buffers. For several general-purpose accelerators (e.g., GPU), the user would also provide binary code to execute specific tasks. Since the untrusted accelerator 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 accelerator, creates a real acceleration environment to replace the stub one, and submits the real task to the accelerator after security introspection. If the software creates a stub page table for the accelerator's DMA, CAGE verifies its mapping, then reconstructs and maintains a real page table during the confidential acceleration. Lastly, the accelerator computes sensitive data and stores results in the realm, from which the user retrieves the data via a secure channel.

Figure 1 shows an overview of CAGE. Based on the existing architecture and hardware primitives in Arm CCA, CAGE supports generic Arm-based confidential computing with acceleration. CAGE deploys three security components in the highest-privilege Monitor to guarantee confidential acceleration. The Monitor is housed in the *root world* 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 acceleration on next-generation Arm devices, CAGE uses a shadow task mechanism (§4.2) based on CCA's realm-style architecture. To avoid exposing sensitive data, the shadow task mechanism requires the untrusted accelerator software to create and manage stub tasks, which are securely replaced by real tasks with authentic data before submission. (**G2:**) Next, CAGE ensures strong data security during the process of the above accelerator workflow. The RMM and previous work [53] isolate realms on the CPU side but have yet to isolate the acceleration environment. Thus, we propose a two-way realm isolation on the accelerator by leveraging the GPCs on the CPU, accelerators, and untrusted peripherals. On the one hand, we restrict access to untrusted components by configuring GPCs for CPU and untrusted peripherals. On the other hand, we isolate realms in acceleration by providing each realm with a different memory view in the accelerator's SMMU GPC (§4.3). (**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 GPTs for SMMU GPC on the accelerator. To achieve this, we design a specific structure of GPTs and the corresponding maintenance process (§4.4) without undermining the security of our protection mechanism. We deploy our security modules in the *root world* Monitor and introduce minimal modifications to the untrusted accelerator driver to collaborate with the Monitor. (**G4:**) Note that CAGE neither requires additional hardware nor customizes the existing CPU or the accelerator, ensuring high hardware compatibility.

To support new accelerators like FPGAs, we reuse a set of security modules in the existing CAGE design, including the GPC-based FPGA environment protection and GPT management process. During computation, since FPGAs must perform DMA operations to load/store data and are controlled by SMMU, our GPC-based memory access control is still feasible on FPGAs. As for the shadow task mechanism, although we show several workflow differences between FPGA and GPU computation, we reuse most of the code in real task creation and environment restoration (e.g., the integrity verification and real buffer creation process).

Deployment scenario. As shown in Figure 1, CAGE only requires realms to store the sensitive data the realm user provides, while Realm OS and RMM are optional. It allows CAGE to handle application scenarios on servers and endpoint devices. When running on servers, CAGE trusts an RMM, which collaborates with the hypervisor to create and isolate realm VMs. Based on this, CAGE extends confidential acceleration support for these realm VMs. As for endpoint devices that currently do not run a hypervisor (and possibly do not run an RMM in next-generation devices), CAGE satisfies confidential acceleration for user-level realms, which can be created and isolated by GPC in recent work [53]. Following CCA's realm-style architecture, our confidential acceleration starts with being created through *normal world* accelerator software, then proceeds to the Monitor to execute the accelerator workload confidentially.

4.2 Shadow Task Management

The primary goal of CAGE is to provide confidential hardware accelerator support for Arm CCA. To achieve this goal, a naïve solution [21], [24], [26], [29] is to encapsulate the heavyweight accelerator software stack (e.g., the accelerator driver and the user-level runtime) into each realm. However, this introduces an extensive TCB within the realm and exposes a large attack surface. To address this problem, we adapt CCA's realm-style architecture to the accelerator workflow. Specifically, we delegate the heavyweight, complex, but data-independent functions to the *normal world* while preserving sensitive data within realms. In doing so, realms benefit from the hardware acceleration without needing to expose the whole TCB entailed by the entire accelerator software stack.

C1. However, adapting this workflow to CCA's realm-style architecture is challenging, primarily because the host-side software is not intended to create and manage accelerator applications belonging to 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 2 shows the design of the shadow task mechanism. We build a pair of accelerator tasks: (1) the stub task, which has the same structure as a normal accelerator task (e.g., buffers, metadata, and other data structures) but does not contain any sensitive data, and (2) the real task, which includes sensitive data to be processed on the accelerator hardware. Thus, we require the accelerator software on the host to create and manage the stub task during the confidential acceleration 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. We secure the data path between the real task and the accelerator hardware. After computation, the execution results are finally stored in the realm and fetched by the realm user. We elaborate on the workflow of the shadow task mechanism as follows.

Initialization and stub task creation. A key aspect of the shadow task mechanism is creating an empty stub accelerator 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 the descriptions of data buffers. The descriptions show the essential 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 descriptions and code or changing the execution order of tasks, we require the user to provide a task signature (e.g., Hash-based Message Authentication Code) for integrity verification. Specifically, the user can generate the signature (e.g., hash-based MAC for code, descriptions, and expected task execution order) from the remote. These signatures can be transferred to the realm via a secure channel between the remote and its realm (built in trust establishment process). To prevent rollback attacks, CAGE can follow the state-of-the-art [54] to introduce version number (VN) in integrity checking. As a result, CAGE effectively verifies the integrity of tasks before submission.

Several accelerators, such as FPGA, do not contain any code buffers. Instead, the FPGA designer programs the computing logic into FPGA hardware. Thus, we prepare the data buffers for the FPGA task based on FPGA hardware and then connect these buffers to specific channels to load or store data. Considering the adversary may tamper with such connection to mislead our protection, we require the user to additionally provide correct channel-buffer relations in each data buffer description. Thus, we can verify the integrity of these descriptions and connect the data buffers in real tasks to the target channel.

Based on the code and descriptions, the accelerator software allocates memory, prepares the stub task, and stages sensitive code and data descriptions into the corresponding data buffers. Suppose the accelerator relies on the page table to perform DMA. In that case, one additional issue here is to record the creation and update of the accelerator page table for the stub task (i.e., `Stub PTE` in Figure 2), which enables synchronizing the page table of the real task with the stub task (i.e., `Real PTE` in Figure 2). 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 accelerator software generally does not update the accelerator 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 accelerator 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.

After preparing the stub task, the accelerator software inserts the task into the task queue to schedule the task execution order. The accelerator software follows the general workflow to process the non-confidential tasks (i.e., directly submitting the normal tasks to the accelerator hardware), while the workflow is different for handling the stub tasks. Since tasks running inside the accelerator hardware may map to the memory of other tasks (and thus leak sensitive data), the accelerator software must help to create an exclusive computing environment for the stub tasks. Therefore,

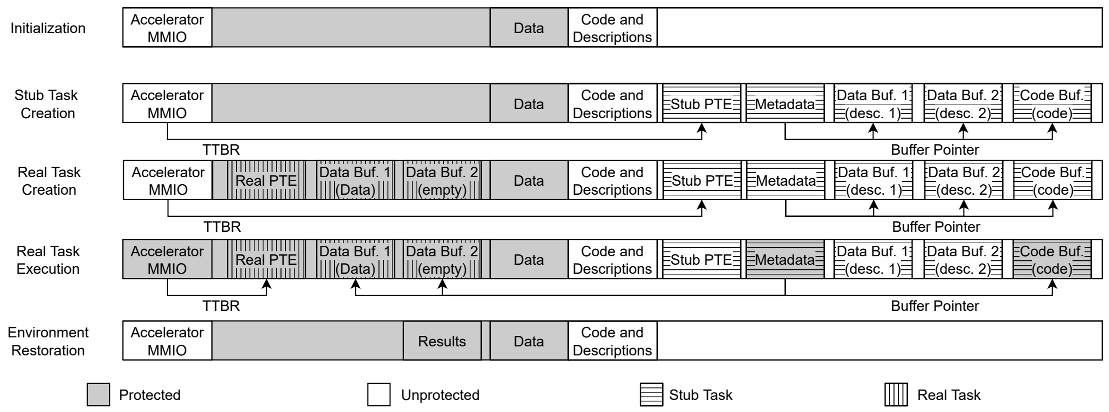


Figure 2: The design of shadow task mechanism. Data Buf. 1 is the input buffer and Data Buf. 2 is the output buffer. Note that several accelerators may not require the page table (PTE) and code buffers.

when processing the stub tasks, the accelerator software stack cleans up the accelerator hardware by (1) temporarily blocking the submission of other tasks and (2) waiting until the current accelerator task (if any) is completed. When the accelerator task is completed, it unblocks the submission of new tasks. Next, the accelerator software routes the stub tasks to the Monitor for execution. However, since the adversary can compromise the accelerator software, CAGE must double-check the accelerator 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 accelerator. We provide the memory layout of this process in Figure 2 and detail the operations as follows.

The structure of the real task is similar to that of the stub task. Specifically, it reuses the code and metadata inside the stub task but prepares different data buffers and page tables. To create the data buffers, the Monitor verifies the integrity of the descriptions by the provided signature, then faithfully constructs buffers based on the provided attributes (e.g., the buffer size). If the description requires, the Monitor fills sensitive data into the target real data buffers. Next, to construct the page table, the Monitor checks the recorded page table entries (e.g., whether they have duplicated or illegal mappings) and replays the mappings in the realm. Since the stub data buffers do not participate in the accelerator's execution, the Monitor changes the corresponding mappings to the real buffers.

To execute the real task, the Monitor first protects the acceleration environment (detailed in §4.3) by locking the metadata and accelerator MMIO as well as the code buffer regions. If these regions are unprotected, an adversary could leak or modify sensitive data (e.g., by exporting the execution results to an unprotected buffer or by executing malicious code). After protection, the shadow task module replaces the stub task with the real task by modifying the buffer pointer to point to the new buffers in the realm. If the page table is created, CAGE also changes the stub page table to the real one in `TTBR`. Considering that the compromised accelerator software may provide incorrect task code and data buffer descriptions, we must verify the signature to ensure integrity. The adversary could also tamper with the metadata or provide an incorrect address of metadata, inducing a Denial-of-Service without leaking the sensitive data. Next, the Monitor checks the current accelerator status, ensuring that no malicious tasks are hidden in the accelerator. Once verified, the Monitor submits the real task by writing the start command to the corresponding accelerator registers. Since the MMIO to the accelerator registers is already protected before accelerator status checking, the adversary can neither hide malicious tasks in the accelerator nor modify the verification results.

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

Protection on FPGAs. Compared with the unified-memory accelerators, an FPGA owns a dedicated physical memory to compute data. Since the GPC cannot control the isolation inside FPGA, the adversary may directly access this memory to leak or tamper with the sensitive data. For example, the adversary can immediately dump the FPGA memory after confidential computation. To defend against this attack, we follow the state-of-the-art [21], [26] to fully control FPGA MMIO during the lifecycle of a confidential FPGA application. After computation, the restoration module cleans the recently used memory in the accelerator before canceling accelerator protection.

Realm and Monitor TCB. CAGE does not introduce large TCB (e.g., a compiler or runtime) to the realm world. The realm is delegated to protect sensitive data and is not aware of the accelerator device. Instead, the user can prepare the code, descriptions, and task execution order in the remote, then transfer these binary files to CAGE. Based on this, CAGE reuses the runtime and compilers in the Host OS to prepare the accelerator application. Note that we do not consider the compilers and user-layer libraries in the normal world as CAGE's TCB.

Moreover, CAGE does not prepare real tasks in the realm, but in the Monitor. Nevertheless, these operations do not introduce a heavy runtime (e.g., OpenCL library [42]). In the Monitor, we achieve the real task preparation with three functions: (1) create new data buffers and fill data, (2) create new page tables, and (3) modify buffer pointers in the stub task metadata. Our evaluation (§6.1) also shows that the shadow task mechanism only introduces a small TCB.

4.3 Acceleration Environment Protection

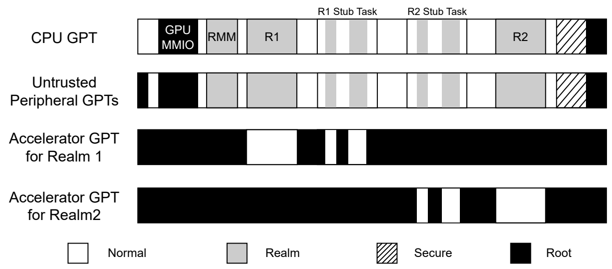

CPU GPT
GPU MMIO
RMM
R1
R1 Stub Task
R2 Stub Task
R2
Untrusted Peripheral GPTs
Accelerator GPT for Realm 1
Accelerator GPT for Realm2
Normal
Realm
Secure
Root

Figure 3: Access control and isolation of acceleration environment when CAGE performs confidential acceleration. 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 an isolated execution environment to support the confidential acceleration. Based on the memory layout in the shadow task mechanism, CAGE secures both the accelerator runtime and the task memory, preventing illegal access to sensitive data, code, and acceleration environment configuration.

C2. However, the latest support for confidential acceleration in Arm CCA (i.e., RME-DA [20]) is only an abstract concept without completed hardware or firmware implementations. Thus, accelerator hardware in the existing Arm CCA implementation is considered 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 protect the execution environment of the accelerator directly. First, since the accelerator 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 adversaries from accessing the accelerator execution environment.

Solution to C2. We leverage the existing memory isolation mechanism in Arm CCA, GPC, to achieve two-way isolation between the accelerator execution environment and the other components, including the untrusted software, peripherals, and other realms. Deployed on the CPU MMU and peripheral SMMUs with basic CCA support, 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 acceleration environment isolation for the *normal* accelerator hardware.

Figure 3 shows various GPT configurations in our accelerator execution 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 Accelerator GPT for realm isolation on acceleration. 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 Accelerator GPT for confidential computing, our accelerator protection module provides each realm with a unique Accelerator GPT, in which the corresponding acceleration environment is strictly isolated from the other regions. The Accelerator GPT only allows the accelerator to access the unique realm region. At runtime, CAGE switch the Accelerator GPT to prevent the realm from accessing other realms' memory via the accelerator. Our protection mechanism is based on the existing GPC design in Arm CCA, without needing the latest Arm RME-DA [43] support, or any hardware modification of the CPU or the accelerator. We elaborate upon the design and configuration of the acceleration environment protection below.

Preventing access from untrusted components. CAGE restricts illegal access to the acceleration environment from untrusted software and peripherals. Recall from §4.2 that CAGE secures the sensitive data and the real data buffers in realms. Thus, the adversary cannot access the authentic data during acceleration. However, we still need to temporarily protect (1) the reused metadata and code buffers and (2) the Accelerator MMIO since the shadow task mechanism allows the untrusted software stack to manage these components. To secure the reused metadata and the code buffers, CAGE configure these regions as *realm* in CPU GPT and untrusted peripheral GPTs, while illegal access from other realms is prevented by CPU-side isolation (e.g., RMM). Considering that the compromised accelerator driver may provide the incorrect address of the reused metadata and the code buffers, the environment protection module must check whether these regions overlap with other *realm* or *root* regions in CPU GPT. If it overlaps, this module terminates the acceleration and restores the acceleration environment. As for the Accelerator MMIO protection, CAGE configures this region as *root* in CPU GPT and untrusted peripheral GPTs. Note that the Accelerator MMIO is a fixed and unmodifiable region in most Arm-based devices [35], [55], [56], [57]. Moreover, CAGE flushes the TLB entries for CPU GPC and untrusted peripheral GPCs to defend against the TLB attacks.

Realm isolation on accelerator. Besides access control on the CPU and untrusted peripherals, we must isolate the accelerator's access to the unauthorized regions to complement our two-way isolation. Therefore, CAGE designs Accelerator GPTs for different realms. When the accelerator executes a confidential task, CAGE fetches the corresponding Accelerator GPT to confine the accelerator's memory access. Since Accelerator GPTs and GPC configurations are only accessible to the *root world* Monitor, the executed task cannot bypass the Accelerator GPC to access the unauthorized physical memory. As for the configuration of Accelerator GPTs, our protection mechanism only permits access to the realm region, the reused metadata, and the code buffers, primarily because they can access the sensitive data of other realms via the accelerator. To allow the accelerator to access these regions, our protection configures the corresponding PAS as *normal* in Accelerator GPT. Finally, to enable Accelerator GPC for this realm, our protection configures Accelerator GPC registers (e.g., set the base address of Accelerator GPT to that of the realm) and flush the TLB entries.

Currently, since Arm CCA regards generic accelerators as untrusted peripheral, CAGE must configure the authorized realm region as *normal* (instead of *realm*) in the accelerator GPT. This allows the accelerator to pass the SMMU GPC and access the authorized region (e.g., sensitive data in a specific realm). Nevertheless, in other accelerator GPTs, CAGE still configures the unauthorized realm region and Monitor as *root* to prevent illegal access.

4.4 GPT Maintenance Optimization

As discussed in §4.3, CAGE ensures memory isolation between the confidential acceleration and untrusted components with different GPTs. The CPU GPT and peripheral GPTs are initialized in the Monitor during the boot phase. As for the Accelerator GPTs, they are initialized during the creation of the corresponding realms. Further, CAGE dynamically maintains these GPTs during confidential acceleration.

C3: However, managing multiple GPTs can introduce non-trivial performance overhead for two reasons. First, during the confidential acceleration, we must synchronize the access control of the CPU and untrusted peripherals on their GPTs. Second, in the realm initialization process, we must create a unique Accelerator 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 for these two issues.

First, we propose a new solution to mitigate the redundant synchronization process on CPU and peripheral GPTs. Recall from §4.3 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 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 describe 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 shared with the CPU GPT and untrusted peripheral GPTs, to protect the reused region (i.e., metadata and code) on task memory. Figure 4 shows our optimization mechanism on GPT synchronization. We require the accelerator software to prepare the stub 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 accelerator, 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 [53] on GPT construction to reduce the latency when creating Accelerator GPTs. In the vanilla GPT construction method (e.g., GPT construction in TF-A [58]), the CCA Monitor prepares GPT that is mainly used for CPU-side access control. Thus, such GPT must contain address space information that is related to *secure* (e.g., secure hypervisor and OSes), *realm* (e.g., RMM and realms), and *root* (e.g., Monitor). Such address space can be scattered and require complex effort to prepare, while most of these regions (except the specific realm) are inaccessible in CAGE's Accelerator GPT. This motivates CAGE to design a specific GPT construction process instead of reusing the vanilla functions. In CAGE, the Accelerator 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 Accelerator GPTs for different realms are derived from one GPT template with minimal adjustment. Specifically, CAGE forks the Accelerator GPT from a template configuring the entire main memory as *root*, then sets the realm region as *normal*. During confidential acceleration, the GPC additionally configures the reused task memory (i.e., code and metadata) as *normal* to permit the accelerator to access them.

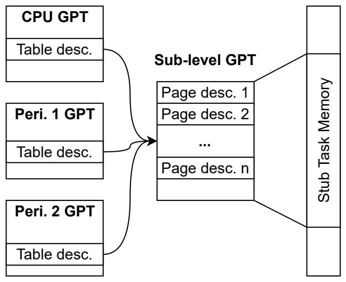


Figure 4: Optimization mechanism in GPT synchronization.

4.5 Trust Establishment in CAGE

Secure boot. CAGE leverages CCA's secure boot process [45] 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, CAGE securely initializes CAGE to protect acceleration.

Remote attestation and key management. Following Arm CCA's framework [20], 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 accelerator hardware. CAGE can combine them to generate an attestation report for each realm to validate the acceleration environment.

CAGE can benefit from other works [28], [50], [59] to manage keys for each realm. Specifically, the user can exchange keys with its realm via Diffie-Hellman [60] or Elliptic-curve Diffie-Hellman [61] protocol. To defend against man-in-the-middle attacks, our trust modules 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 unauthorized access. Based on this, a secure communication channel is built between the realm user and the realm to transfer sensitive data and signatures. Thus, the transferred data are decrypted in realms before acceleration.

5 Implementation

5.1 Functionality Prototype

We prototype CAGE on the Arm FVP Base RevC-2xAEMvA [34] simulator with RME enabled. It simulates the latest Armv9 hardware features and is widely used in previous studies [21], [51], [53], [62], [63]. The FVP provides a connected test engine that faithfully performs memory accesses as though it is a DMA-capable accelerator 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. We use Linux kernel v5.3.0 as the host and Trusted Firmware-A (TF-A) v2.8 [64] as the Monitor.

5.2 Performance Prototype

Since the platform of our functionality prototype is not cycle-accurate [34], [53], 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 [35] with a Mali-T624 GPU and 8GB DRAM. To interact with the GPU, we deploy the Arm official Midgard GPU driver [40] and OpenCL [42] 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. To properly measure the performance overhead of these functions, we faithfully model the expected real-world performance overhead associated with CCA-related operations in three aspects. First, we replace CCA-related instructions with Armv8 instructions that cost similar latency. 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`). Third, we emulate the configurations on several GPT instances, such as manipulating page tables and accessing GPT-related registers.

Interaction with GPU hardware. Although our functionality prototype uses the test engine to perform memory access as the DMA-capable peripheral, the performance prototype still needs three additional steps to interact with the authentic Arm GPU. First, we verify the status of the Mali GPU by checking the accelerator register status. We mainly check two parts of the registers: (1) the `JS_*` registers (e.g., `JS_STATUS` and `JS_HEAD`), which indicates the status of GPU hardware and GPU tasks; and (2) the `AS_*` registers (e.g., `AS_TRANSTAB` and `AS_TRANSCFG`), which shows the configurations of GPU address translation. If several `JS_*` registers show non-zero values (i.e., the number of running tasks is not zero), or `AS_*` registers show an incorrect configuration (e.g., incorrect page table base), we will terminate the task computing and restore the environment. Second, when submitting 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 a Monitor interrupt for further operations (e.g., catch the interrupt and restore environment) in the Monitor.

Preparing confidential applications. To adapt a normal accelerator application in CAGE's confidential computing, the developer does not require a large engineering effort. To achieve this, developers should introduce an additional wrapper to create page-aligned buffers. For each stub data buffer, the developer should prepare a buffer description (e.g., buffer size and required data) based on CAGE's requirement and fill the description to buffer. Overall, our design does not change the native programming libraries (e.g., OpenCL library [42]), GPU code kernel, and most of the GPU application code.

6 Evaluation

In this section, we evaluate our CAGE prototypes (§5) with respect to four research questions:
RQ1: How large is the TCB of CAGE?
RQ2: Can CAGE protect GPU from privileged adversaries?
RQ3: How much performance overhead does CAGE incur on GPU benchmarks and neural network models?
RQ4: How effective is our optimization on GPT maintenance?

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

6.1 RQ1: TCB Size of CAGE

We use a generic code statistic tool, *cloc* [65], to measure the TCB size of CAGE in terms of standard lines of source code. CAGE modifies an Arm Trusted Firmware-A v2.8 [64] Monitor by introducing 1,301 LoC additions, including 667 LoC on shadow task management, 350 LoC on accelerator environment protection, 137 LoC on GPT maintenance optimization, and 147 LoC on other configurations. In addition, CAGE must trust a thin CPU-side isolation software (e.g., TF-RMM v0.2 [66] with 26K LoC, or Shelter [53] with 2K LoC). Overall, the introduced TCB is smaller than the accelerator software stack (e.g., an approx. 30K LoC Midgard GPU driver [40] or a 32MB OpenCL libraries [42].

6.2 RQ2: Security Analysis for GPU Computing

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 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

Table 1: 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 memory management	①③
	Illegal task scheduling	②③
	Malicious tasks	①③
	Fake accelerator and SMMU	④
	CPU GPC circumvention	①⑤⑥
Peripherals	Malicious DMA	①
	Peripheral GPC circumvention	①⑤⑥
Realms	Realm abuse	①⑦

realm via a secure channel, our integrity verification process cannot be subverted.

Illegal 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 [67] 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. Moreover, CAGE does not trust the accelerator software and only uses it for preparing the computing environment, such as recording page table entries. However, to detect illegal operations, CAGE will additionally check the environment, such as checking the duplicated or unauthorized mapping in the page tables. Note that CAGE checks the page table records and builds real page table in protection regions, preventing the untrusted accelerator to perform illegal operations in this process.

Illegal 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 accelerator 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 [35] 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. For the third-party accelerators (e.g., a third-party FPGA [14]), we suggest the accelerator install a hardware root of trust to support the attestation [68], [69].

Malicious tasks. The adversary may directly leak or tamper with the sensitive data in realms. To achieve this, the adversary 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, the adversary may compromise the isolated execution environment of confidential tasks. During the confidential GPU computation, the adversary 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, the adversary may launch an Iago-style [67] attack in the GPU driver. To hide a malicious GPU task, the adversary 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 the adversary 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* 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

Table 2: Problem size of the selected Rodinia benchmarks.

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

cannot modify these TLB entries since it is not supported by the 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 [53]). Moreover, the adversary may launch confidential GPU applications to achieve the same attack from the GPU, but this is restricted by GPU SMMU GPC.

6.3 RQ3: Evaluations on GPU Benchmarks

Experimental setup. To measure the performance of our prototype, we follow best practices from previous GPU TEE studies [28] 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 2. 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 5 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 3. 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 3 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 3: 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

To further discuss the performance comparison between CAGE and state-of-the-art, we compare CAGE and StrongBox [28] on the same Rodinia benchmarks. Figure 5 shows that CAGE introduces slightly lower overhead on these benchmarks than StrongBox. 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.

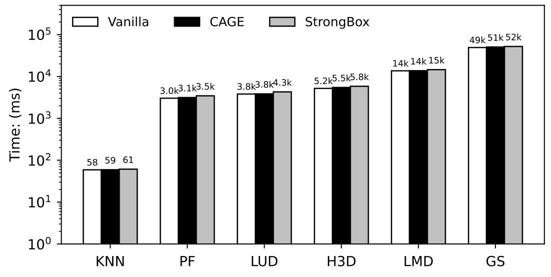


Figure 5: Performance overhead on six Rodinia benchmarks.

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 [70], SqueezeNet [71], and MobileNet-v1 [72]), which cover a wide range of use case (detailed in Table 4). 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 4 reports the performance comparison between CAGE and the native system with a detailed breakdown. It shows that CAGE introduces 1.24% – 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.

6.4 RQ4: Evaluation on GPT Optimization

Synchronizing CPU and peripheral GPTs. CAGE optimizes the GPT synchronization in accelerator 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 accelerator 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 6, 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.

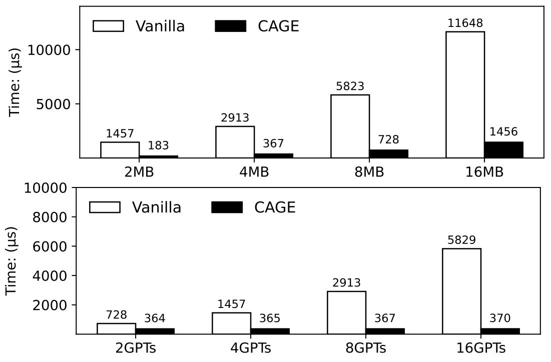


Figure 6: Performance comparison between the non-optimized mechanism and CAGE in GPT synchronization. Note that the upper benchmark configures the GPT number as 8 and the lower benchmark sets the size of the protected region as 4MB.

Initializing Accelerator GPT for each realm. We further compare our Accelerator GPT initialization mechanism with the native solution. Since the existing TF-A has yet to provide the specific GPT configuration for accelerator, we emulate it by invoking the CPU GPT initialization APIs in TF-A v2.8 [64]. 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). In our evaluation, we assume each confidential GPU application perform only initialize accelerator GPT one time. As shown in Figure 7, we mitigate 84.63% – 96.55% performance overhead of Accelerator GPT initialization in these configurations. Note that CAGE optimizes the GPT maintenance operations but does not introduce significant performance improvement in confidential GPU application. This is because the GPT creation time takes a relative small part in confidential GPU computing.

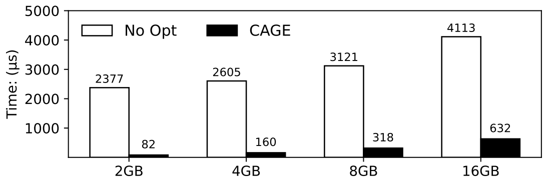


Figure 7: Performance improvement in Accelerator GPT initialization with four different size of main memory.

7 Extending CAGE on FPGA Computing

Besides GPUs, the generic FPGA accelerators are increasingly attractive to cloud providers [73], [74]. As reconfigurable computing fabrics, FPGAs serve an important role in custom accelerators. Studies have used FPGAs to support various high-performance workloads (e.g., neural network training and tensor processing). This reconfigurability also allows manufacturers re-program the FPGA to support the latest computing requirements. However, generic FPGA designs focus on high computing performance but do not carefully consider data security, so that a privileged adversary can easily access the sensitive data entailed during FPGA computation. In addition, the existing CCA design regards generic FPGA hardware as an untrusted peripheral and does not guarantee its security. Recent studies [69], [75] propose security extensions inside FPGAs, although this reduces hardware compatibility on FPGA design. Hence, there is a need to complement Arm CCA with generic FPGA computing extension.

To address these issues, we propose an FPGA extension to CAGE, which supports confidential FPGA computation on Arm CCA. We introduce several major changes to CAGE to adapt to an FPGA's workflow. We also analyze and discuss the security and performance of our extension prototype. Our extension secures generic FPGA computation and can potentially adapt to other accelerators that use similar workflows and hardware design to FPGA, such as NPU and TPU.

7.1 Prototype Implementation

Our extension reuses the same functionality prototype (§5.1) to verify the accelerator functionality and security guarantee. Moreover, due to the architecture variance between GPU and FPGA, we must implement an additional prototype to evaluate CAGE's performance on FPGA. This prototype is built on an Arm Juno R2 board, which connects to a Xilinx Virtex Ultrascale+ VCU118 FPGA [36] via a PCIe port. To implement the FPGA-based prototype, we implement the aforementioned changes to adapt FPGA's workflow with CAGE and follow the same techniques as

Table 4: 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

Table 5: Problem size of the selected FPGA benchmark.

Application	Type	Size	Memory
MM-256	Matrix Multiplication	128 × 128 nodes	0.75 MB
MM-384	Matrix Multiplication	256 × 256 nodes	1.69 MB
MM-512	Matrix Multiplication	512 × 512 nodes	3.00 MB
FFT-1024	Fast Fourier Transform	1024 nodes	16.01 KB
FFT-2048	Fast Fourier Transform	2048 nodes	32.01 KB
FFT-4096	Fast Fourier Transform	4096 nodes	64.01 KB

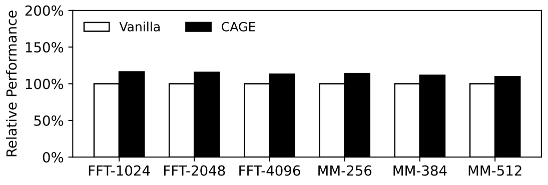


Figure 8: Relative overhead on the six FPGA benchmarks.

the GPU prototype (e.g., using the same approach to emulate CCA-related operations and CPU cycle measurement).

We prepare FPGA tasks as C programs and use the Xilinx PCIe DMA (XDMA) [41] driver. To collaborate with the XDMA driver, we deploy an XDMA IP core in FPGA hardware and export two types of DMA channels: `H2C` (Host to Card) channel, which loads data from the host to FPGA, and `C2H` (Card to Host) channel, which stores data from FPGA to the host. To properly use these channels, the data buffer descriptions of confidential task should contain additional bits to represent the buffer-channel relation (i.e., using 32 bits to represent buffer ID and 32 bits to represent `H2C/C2H` channel ID). To ensure relation integrity, we also generate task signatures with this relation.

During the shadow task mechanism, we first verify the `H2C/C2H Channel Status` registers to ensure a clean execution environment. Next, we verify each buffer description's integrity, which helps create real data buffers with authentic data. Note that the buffer description also provides critical information (e.g., index of a data transfer and its target `H2C/C2H` channel) to ensure the correctness of data transfer. Lastly, to help the FPGA hardware interact with the correct data buffers, we modify several attributes in XDMA's metadata (called `descriptor`). Specifically, we change the address in `Src_adr` when FPGA copies data from the host, and modify `Dst_adr` value when FPGA sends the execution results to the host. However, in this step, the `descriptors` host prepared can be inefficient for the real task. This is because the size of real data buffers can be larger than the stub data buffers, while each `descriptor` can only represent a limited size (e.g., `0x1000` in our FPGA board) of data transfer between data buffer and FPGA. To address this problem, we follow the official XDMA manual [76] to create a set of `descriptors` in the realm and fill the `Nxt_adr` field (i.e., the address of next `descriptor`) of each `descriptor`. Thus, the FPGA will read `descriptors` in order to perform data transfer. As for task submission, we set the last bit of the `H2C/C2H Channel Control` register to `0x1` to run the real task. After FPGA computing, we must clear this bit to idle the corresponding channel.

7.2 Performance Evaluation for FPGA Computing

Experimental setup. We measure our performance prototype on FPGAs. Since FPGA-based computation lacks standard benchmarks, we follow the previous work [21] and hand-coded two types of FPGA benchmark applications: (1) a small-sized fast Fourier transform and (2) a large-sized matrix multiplication as C programs. These two applications follows a highly similar workflow: creating data buffers and filling descriptions, opening `C2H` and `H2C` data transfer channels, providing input buffers, computing, and fetching results from output buffers. This is because the FPGA supports highly customized hardware programming, and therefore we directly compile the computing logic into FPGA hardware instead of providing software task code. For each application, we measure them with three different input problem sizes, which are reported in Table 5.

Performance analysis. Since the results of these FPGA benchmarks are widely varied, we normalize the execution time in Figure 8. Figure 8 shows that CAGE introduces 9.61% – 16.30% performance overhead on these benchmarks. We observe that CAGE shows a better performance on several large-sized benchmarks. For instance, CAGE introduces 11.59% and 9.61% performance overhead on `MM-384` and `MM-512` benchmarks, respectively. The major reason is that the data-independent security operations in CAGE (e.g., context switching between the host and the Monitor) have a relatively small influence on these large-sized benchmarks. Note that the performance overhead can be slightly increased in real CCA-supported devices because of performing GPC.

7.3 Security Analysis for FPGA Computing

Since we extend CAGE's accelerator protection on FPGA without any hardware or threat model changes, the aforementioned analysis on GPU computation (§6.2) covers most attack scenarios in FPGA computation as well. Nevertheless, due to the hardware architecture and workflow variance between GPU and FPGA, the adversary may exploit a new attack surface for FPGAs. In addition, such variance provides our new CAGE system with new defense mechanisms against several previous attacks. We elaborate upon these attacks and the defense mechanisms below.

TCB analysis. CAGE should modify the Monitor to serve the workflow of new accelerators. Nevertheless, such changes does not introduce large TCB to the Monitor – CAGE's FPGA extension introduces a thin TCB addition (140 LoC) in the Monitor. This is because we reuse most security operations in previous prototype, such as the integrity verification, the GPC-based protection and GPT management. Moreover, same as the previous prototype, our extension reuses the heavy FPGA software stacks (e.g., an approx. 6K Xilinx XDMA driver [41]) and does not introduces them to realms. Overall, our FPGA extension does not expose a large attack surface to the adversary.

Attack on task management. In FPGA computation, the adversary may provide an incorrect channel-buffer relation to mislead the computing. Specifically, the adversary may feed the data buffer into incorrect `H2C` or `C2H` channels, or provide incorrect buffers for each data transfer channel. To defend against this attack, we require the user to provide a detailed channel-buffer relation in each data buffer description. By verifying the integrity of description, we can provide the data transfer channel with correct data buffers. Note that the signature of the data buffer description effectively prohibits the adversary from tampering with such relations.

Attack on environment isolation. The adversary may insert a malicious task to access or tamper with sensitive data between the execution of two confidential tasks. Since the GPC cannot properly restrict CPU side from accessing FPGA's dedicated memory, in our new design, we control the FPGA MMIO during the execution of the entire FPGA application. Such defense prevents the adversary from submitting any malicious task to FPGA during the entire confidential computing. After computation, we also reset FPGA's dedicated memory to clean the data. Thus, the adversary cannot break the isolated FPGA environment to leak the sensitive data.

Attack on metadata. The adversary who controls the FPGA software stack may modify the metadata during the confidential acceleration or provide an incorrect metadata. The previous solution secures the provided metadata and considers the incorrect metadata only incurs a Denial-of-Service with without data leakage. Nevertheless, in FPGA computation, we defend against this attack by directly creating a new metadata for the real task. This is because the FPGA manual [76] provides the detailed format and description of the FPGA task metadata (i.e., XDMA `descriptor`). Thus, we create a set of new metadata inside realms and fill the authentic data buffer address, achieving a higher security guarantee.

8 Discussion

Task blocking. As aforementioned in §4.2, CAGE leverages a blocking method to prevent the accelerator driver from submitting non-confidential tasks. However, for unified-memory accelerators (e.g., Arm Mali GPUs [12]), our solution does not introduce large performance overhead, although the confidential application may have multiple tasks. This is because CAGE follows the task scheduling in the driver and does not block it in the entire lifecycle of the confidential application. CAGE only assigns the accelerator to confidential applications when executing its tasks. Meanwhile, CAGE releases the accelerator when it finishes the computing of a confidential task, allowing the accelerator driver to schedule the accelerator for non-confidential tasks. For accelerators with dedicated memory (e.g., NVIDIA GPUs [11] and Xilinx FPGAs [14]), CAGE must ensure the accelerator is assigned to the confidential application in its entire lifecycle. Nevertheless, CAGE can leverage the orthogonal studies (e.g., GPU latency hiding [77]) to reduce the overhead from GPU task blocking.

Physical attacks. CAGE mainly focuses on protecting accelerators from software-based adversaries, while orthogonal studies can help CAGE on physical attacks. For the attack via physical interfaces (e.g., JTAG or I2C interfaces), CAGE can leverage the detection solution [78] or secure physical interfaces [79] to prevent these attacks. In regard to attacks on interconnects (e.g., PCIe, AXI or NVLink bus) and memory, CAGE can leverage the hardware-assisted security features (e.g., PCIe Integrity and Data Encryption [80] and CCA Memory Protection Engine [81]), or memory protection solutions on bus [46], [47] to defend against these attacks. Moreover, the adversary may launch rollback attacks on GPU memory, while CAGE can use the monotonic counters [23], [82] to record data states and guarantee the freshness of GPU memory.

Code protection. Following with several state-of-the-art [28], [83], [84], CAGE ensures confidentiality on sensitive data but not code. We implement this strategy due to three major reasons. First, several accelerator applications leverage a generic code binary (e.g., generic deep learning model) in computing, while the input parameters are different and sensitive. Second, providing plaintext code binary helps the accelerator software to achieve its functionality and does not leak sensitive data. Third, adding code confidentiality protection introduces additional performance overhead due to de/encryption.

Monitor TCB optimization. CAGE modifies the Monitor to protect accelerators, while our changes do not introduce large TCB size to the Monitor. Nevertheless, inspired by the state-of-the-art [85], we can delegate partial accelerator protection operations (e.g., environment protection operations and accelerator GPT maintenance) to realms or RMM, which are trusted in CAGE but have lower privilege than the Monitor. Based on this, the Monitor can use a smaller TCB (e.g., interaction modules with the external hardware) to control and provide sensitive information with such hardware.

Moreover, for Arm platforms with a large number of accelerator devices, CAGE can design hierarchical accelerator management to minimize the added TCB in the Monitor. To achieve this, CAGE can decouple the device-specific functions, such as page table creation for specific GPU/FPGA, from the device-agnostic layer. This solution helps CAGE to reduce the redundant TCB for accelerator protection. However, if CAGE-equipped platform only supports a few types of accelerators, this solution can introduce more TCB for implementing device-agnostic functions.

Supporting new accelerators. CAGE's design is not essential to change every time for a new accelerator device. One major reason is that several accelerators from the same vendor can use similar hardware architecture and workflow (e.g., XDMA-based PCIe data transmission for Xilinx FPGAs). For accelerators with different workflows, CAGE should modify the Monitor to serve the workflow of new accelerators. Nevertheless, such changes do not introduce large software changes to the Monitor. This is because we reuse most security operations in the previous prototype, such as the integrity verification, the GPC-based protection, and GPT management.

Performance improvement. Besides the improvement of system security, CAGE can leverage FPGA's programmability to improve performance. Specifically, the FPGA can accelerate several memory transfer operations (e.g., copy data and verify data buffer description) with a well-designed engine. Moreover, during environment restoration, the FPGA can provide a hardware-assisted module to rapidly clean its dedicated memory. These hardware designs can effectively mitigate CAGE's performance overhead.

Suggestions on future Arm devices. CAGE configures GPCs on SMMU to manage access from accelerators and other peripherals. During the configuration of SMMU, we find that Arm devices typically use an integrated PCIe SMMU to control access from all PCIe peripherals including FPGA. 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. However, SMMU only provides a single MMIO region to set the GPT so that all the connected peripherals share the same view of the main memory in GPC. Although CAGE can combine PCIe Stage-2 translation with our GPC-based protection to address this problem (its feasibility is proved in [21]), such design still introduces more TCB to the Monitor and influence data security. In the future, we suggest that Arm introduces `StreamID` to distinguish different SMMU GPCs or allows SMMUs to provide each peripheral with a unique `Root control page` to configure its own GPC.

9 Related Work

Arm confidential acceleration. We discuss the existing Arm confidential acceleration solutions: The official RME-DA [20] and ACAI [21]. The RME-DA is an Arm-based implementation of TDISP [86], which aims at building a trusted framework among CPU TEEs, PCIe, and accelerator. Both solutions require non-trivial modification to protect acceleration from the threat of hypervisor: RME-DA modifies the SMMU hardware, and ACAI modifies SMMU-related code in hypervisor software. Moreover, they still require heavy accelerator software into realms to control accelerator applications, introducing large TCB size to the realm. Nonetheless, these solutions propose several new mechanisms to attest and manage the accelerator, potentially benefits CAGE for accelerator authentication.

GPU TEE. Researchers have explored GPU TEEs to guarantee data security on GPUs. To secure communication between the user and the GPUs, typical GPU TEEs leverage a CPU-side TEE to transfer sensitive data to the GPU and control access to GPU MMIO. Non-Arm GPU TEEs achieve this with Intel SGX (e.g., HIX [26] and GEVisor [87]), AMD SEV-SNP (e.g., HoneyComb [27]), bus controllers (e.g., CURE [23] and HETEE [24]), and other access control mechanisms. However, migrating non-Arm GPU TEEs to Arm devices can be challenging due to variations in architecture and organization. For Arm-based GPU TEEs [28], [29], [30], [84], they 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 (e.g., Graviton [22] and NVIDIA H100 GPU [25]) directly build TEE in GPU hardware. However, such an approach severely influences hardware compatibility.

Other accelerator TEE. Besides GPU protection, studies have also built a confidential computing environment that supports other accelerators. SGX-FPGA [69], ShEF [68] and GuardNN [75] build FPGA-based TEE. ITX [88] builds IPU TEE on IPU board. These designs introduce a set of secure modules inside the FPGA/IPU hardware board to control data and command transfer with the host, while such implementation reduces hardware compatibility. TNPU [54] and Securator [89] harden NPU TEEs with optimized integrity protection and memory access patterns, respectively. However, they trust the accelerator software stacks and thus increase the enclave's TCB size. Moreover, several GPU TEEs (e.g., HETEE [24], XpuTEE [90], and Cronus [29]) also protect other accelerators including the FPGA, while they have yet to adapt CCA's realm-style architecture to secure accelerator computing.

10 Conclusion

In this paper, we present CAGE to provide accelerator 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 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, we prototype CAGE on an official software-simulated platform. We then port this prototype to an off-the-shelf development board with a unified-memory GPU and evaluate it with rigorous benchmarks. Results show that CAGE provides effective support for confidential GPU computing with a moderate performance overhead.

In this new extension, we augment our approach by including accelerator support for FPGAs, which are widely-used accelerators and can be programmed like other accelerators including NPU and DPU. We provide a new design and implementation for generic FPGA computation due to the hardware and workflow difference compared to unified-memory GPUs. We also evaluate our prototype with a set of FPGA benchmarks and new security analysis. Our results show that CAGE secures generic FPGA computing with 9.61% – 16.30% performance overhead.

References

[1] AMD, "Confidential Computing Solution Brief," https://www.amd.com/en/processors/epyc-confidential-computing-cloud, 2023.
[2] ARM, "Azure confidential computing," https://developer.arm.com/documentation/den0125/0200/, 2022.
[3] Microsoft, "Azure confidential computing," https://azure.microsoft.com/en-us/solutions/confidential-compute/, 2023.

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

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

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

[7] 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.

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

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

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

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

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

[13] ——, "Ethos-N78," https://www.arm.com/products/silicon-ip-cpu/ethos/ethos-n78, 2023.

[14] AMD, "FPGA Leadership Across Multiple Process Nodes," https://www.xilinx.com/products/silicon-devices/fpga.html, 2023.

[15] 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.

[16] 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.

[17] 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.

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

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

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

[21] S. Sridhara, A. Bertschi, B. Schlüter, M. Kuhne, F. Aliberti, and S. Shinde, "ACAI: Extending Arm Confidential Computing Architecture Protection from CPUs to Accelerators," in *Usenix Security*, 2024.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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

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

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

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

[36] Xilinx, "AMD Virtex UltraScale+ FPGA VCU118 Evaluation Kit," https://www.xilinx.com/products/boards-and-kits/vcu118.html, 2023.

[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] C. Wang, F. Zhang, Y. Deng, K. Leach, J. Cao, Z. Ning, S. Yan, and Z. He, "CAGE: Complementing Arm CCA with GPU Extensions," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium*, 2024.

[39] J. Choi, J. Kim, C. Lim, S. Lee, J. Lee, D. Song, and Y. Kim, "Guardiann: Fast and secure on-device inference in trustzone using embedded sram and cryptographic hardware," 2022.

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

[41] Xilinx, "Xilinx DMA IP Reference drivers," https://github.com/Xilinx/dma_ip_drivers, 2023.

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

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

[44] 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.

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

[46] 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.

[47] 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.

[48] Michael Larabel, "NVIDIA Sends Out Signed Firmware Images For GP108 Pascal GPUs," https://www.phoronix.com/news/NVIDIA-GP108-Firmware, 2017.

[49] torpcoms, "AMD Vega Firmware Signing," https://forum.level1techs.com/t/amd-vega-firmware-signing/118459, 2017.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] Amlogic, Inc., “S905 Datasheet,” https://dn.odroid.com/S905/DataSheet/S905_Public_Datasheet_V1.1.4.pdf, 2016.

[56] 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.

[57] STMicroelectronics, “GPU device tree configuration,” https://wiki.st.com/stm32mpu/wiki/GPU_device_tree_configuration, 2022.

[58] ARM, “Arm-Trusted-Firmware,” https://github.com/ARM-software/arm-trusted-firmware.

[59] 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.

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

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

[62] 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.

[63] 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.

[64] ARM, “Trusted Firmware-A release v2.8,” https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git/tag/?h=v2.8, 2023.

[65] AlDanial, “cloc,” https://github.com/AlDanial/cloc, 2021.

[66] ARM, “TF-RMM,” https://git.trustedfirmware.org/TF-RMM/tf-rmm.git/tag/?h=tf-rmm-v0.2.0, 2023.

[67] 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.

[68] M. Zhao, M. Gao, and C. Kozyrakis, “ShEF: Shielded Enclaves for Cloud FPGAs,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 1070–1085.

[69] K. Xia, Y. Luo, X. Xu, and S. Wei, “Sgx-fpga: Trusted execution environment for cpu-fpga heterogeneous architecture,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021.

[70] 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.

[71] 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.

[72] 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.

[73] IBM, “cloudFPGA,” https://research.ibm.com/projects/cloudfpga, 2024.

[74] Amazon Web Services, “Amazon EC2 F1 Instances,” https://aws.amazon.com/ec2/instance-types/f1/, 2024.

[75] 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.

[76] Xilinx, “DMA/Bridge Subsystem for PCI Express Product Guide (PG195),” https://docs.xilinx.com/r/en-US/pg195-pcie-dma/Introduction, 2023.

[77] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, “Warped-Preexecution: A GPU Pre-execution Approach for Improving Latency Hiding,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 163–175.

[78] X. Ren, V. G. Tavares, and R. S. Blanton, “Detection of Illegitimate Access to JTAG via Statistical Learning in Chip,” in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 109–114.

[79] X. Ren, F. P. Torres, R. D. Blanton, and V. G. Tavares, “IC Protection against JTAG-based Attacks,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 1, pp. 149–162, 2018.

[80] PCI-SIG, “Integrity and Data Encryption (IDE) ECN Deep Dive,” https://pcisig.com/sites/default/files/files/PCIe%20Security%20Webinar_Aug%202020_PDF.pdf, 2020.

[81] ARM, “System Architecture,” https://developer.arm.com/documentation/den0126/0101/System-architecture, 2023.

[82] 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.

[83] R. Liu, L. Garcia, Z. Liu, B. Ou, and M. Srivastava, “SecDeep: Secure and Performant On-device Deep Learning Inference Framework for Mobile and IoT Devices,” in *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, 2021, pp. 67–79.

[84] 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.

[85] F. Sang, J. Lee, X. Zhang, and T. Kim, “PORTAL: Fast and Secure Device Access with Arm CCA for Modern Arm Mobile System-on-Chips (SoCs),” in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025.

[86] PCI-SIG, “IDE and TDISP: An Overview of PCIe Technology Security Features,” https://pcisig.com/blog/ide-and-tdisp-overview-pcie%C2%AE-technology-security-features, 2022.

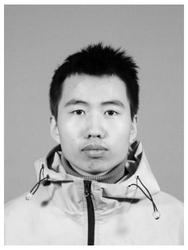
[87] X. Wu, D. J. Tian, and C. H. Kim, “Building gpu tees using cpu secure enclaves with gevisor,” in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, 2023, pp. 249–264.

[88] Microsoft, “Confidential Computing within an AI Accelerator,” https://www.microsoft.com/en-us/research/publication/confidential-computing-within-an-ai-accelerator/, 2023.

[89] N. Shrivastava and S. R. Sarangi, “Securator: A fast and secure neural processing unit,” in *2023 IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 2023, pp. 1127–1139.

[90] S. Fan, Z. Hua, Y. Xia, and H. Chen, “XpuTEE: A High-Performance and Practical Heterogeneous Trusted Execution Environment for GPUs,” *ACM Transactions on Computer Systems*, vol. 43, no. 1-2, pp. 1–27, 2025.

Chenxu Wang is working on the Joint Ph.D. degree from Southern University of Science and Technology (SUSTech) and The Hong Kong Polytechnic University. He received the Bachelor degree in Computer Science and Engineering from Southern University of Science and Technology (SUSTech). His research interests include virtualization and trusted execution environment on Arm architecture.

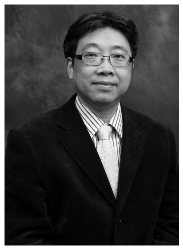


Kun Lu is working on the Master degree from Southern University of Science and Technology (SUSTech). He received the Bachelor degree in Computer Science from Jilin University. His research interests include FPGA trusted execution environment and RISC-V confidential computing.

Fengwei Zhang is an Associate Professor in Department of Computer Science and Engineering at Southern University of Science and Technology (SUSTech). His primary research interests are in the areas of systems security, with a focus on trustworthy execution, hardware-assisted security, debugging transparency, and plausible deniability encryption. Before joining SUSTech, he spent four years as an Assistant Professor at Department of Computer Science at Wayne State University.

Yunjie Deng is working on the Ph.D. degree from Georgia Institute of Technology. He received the Master and Bachelor's degree in Computer Science and Engineering from Southern University of Science and Technology (SUSTech). His research interests include trusted execution environment and GPU computing.

Kevin Leach is an Assistant Professor of Computer Science at Vanderbilt University. He received his PhD degree in computer engineering from the University of Virginia. His research interests include systems security, specifically the debugging transparency problem. He also works in the area of conversational artificial intelligence, program analysis, medical informatics, and big data applications.

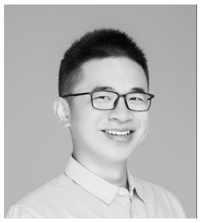


Jiannong Cao is the Otto Poon Charitable Foundation Professor in Data Science and the Chair Professor of Distributed and Mobile Computing in the Department of Computing at The Hong Kong Polytechnic University. His research interests include distributed systems and blockchain, wireless sensing and networking, big data and machine learning, and mobile cloud and edge computing.

Zhenyu Ning is an Associate Professor at Hunan University. He received his Ph.D. degree in Computer Science from Wayne State University in 2020. His research interests are in the areas of security and privacy, including system security, mobile security, IoT security, trusted execution environment, hardware-assisted security.

Shoumeng Yan is a principal engineer at Ant Group and the senior director of secure and trustworthy computing. He received his Ph.D. from Northwestern Polytechnic University. His research interests include OS, TEE, and domain specific accelerators. He publishes many papers in international conferences, including USENIX Security, USENIX ATC, ACM CCS, ACM ASPLOS, and etc.

Tao Wei is the Vice President at Ant Group. He received his Bachelor and Ph.D. degrees from Peking University. He has been committed to making complex systems more secure and reliable. His work has helped Windows, Android, iOS and other operating systems improve their security capabilities. He also led the development of many famous security open-sourced projects such as Mesatee/Teaclave, MesaLink TLS, OpenRASP, Advbox Adversarial Toolbox, etc. He publishes many papers in top-tier journals and conferences, including IEEE TDSC, IEEE TIFS, IEEE S&P, USENIX Security, and etc.



Zhengyu He is a senior principal engineer at Ant Group and the president of platform technology business group. He received his Ph.D. degree from the School of Electrical and Computer Engineering at the Georgia Institute of Technology. His research interests include the trusted execution environment, operating system, and virtualization.