Building a Lightweight Trusted Execution Environment for Arm GPUs

Chenxu Wang, Yunjie Deng, Zhenyu Ning, Kevin Leach, Jin Li, *Senior Member, IEEE*, Shoumeng Yan, Zhengyu He, Jiannong Cao, *Fellow, IEEE*, and Fengwei Zhang, *Senior Member, IEEE*

***Abstract*—A wide range of Arm endpoints leverage integrated and discrete GPUs to accelerate computation. However, Arm GPU security has not been explored by the community. Existing work has used Trusted Execution Environments (TEEs) to address GPU security concerns on Intel-based platforms, but there are numerous architectural differences that lead to novel technical challenges in deploying TEEs for Arm GPUs. There is a need for generalizable and efficient Arm-based GPU security mechanisms. To address these problems, we present StrongBox, the first GPU TEE for secured general computation on Arm endpoints. StrongBox provides an isolated execution environment by ensuring exclusive access to GPU. Our approach is based in part on a dynamic, fine-grained memory protection policy as Arm-based GPUs typically share a unified memory with the CPU. Furthermore, StrongBox reduces runtime overhead from the redundant security introspection operations. We also design an effective defense mechanism within *secure world* to protect the confidential GPU computation. Our design leverages the widely-deployed Arm TrustZone and generic Arm features, without hardware modification or architectural changes. We prototype StrongBox using an off-the-shelf Arm Mali GPU and perform an extensive evaluation. Results show that StrongBox successfully ensures GPU computation security with a low (4.70%–15.26%) overhead.**

***Index Terms*—Arm endpoint GPU, secure virtualization, trusted execution environment.**

Manuscript received 6 April 2023; revised 10 September 2023; accepted 12 November 2023. Date of publication 28 November 2023; date of current version 11 July 2024. This work was supported in part by the National Natural Science Foundation of China under Grants 62372218, 62002151, and 62102175, in part by Shenzhen Science and Technology Program under Grant SGDX20201103095408029, in part by HK RGC General Research Fund under Grant PolyU 15220020, in part by HK RGC Collaborative Research Fund under Grant C2004-21GF, in part by the Research Institute for Artificial Intelligence of Things, The Hong Kong Polytechnic University, and in part by Ant Group Research Fund. *(Chenxu Wang and Yunjie Deng are co-first authors.) (Corresponding author: Fengwei Zhang.)*

Chenxu Wang is with the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen 518055, China, and also with the Department of Computing, Hong Kong Polytechnic University, Hong Kong, China (e-mail: 12150073@mail.sustech.edu.cn).

Yunjie Deng is with the Research Institute of Trustworthy Autonomous Systems, and Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: 12032869@mail.sustech.edu.cn).

Zhenyu Ning is with the Hunan University, Hunan 410012, China, and also with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: zning@hnu.edu.cn).

Kevin Leach is with the Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37235 USA (e-mail: kevin.leach@vanderbilt.edu).

Jin Li is with the School of Computer Science, Guangzhou University, Guangzhou 511370, China (e-mail: jinli71@gmail.com).

Shoumeng Yan and Zhengyu He are with the Ant Group, Hangzhou 310000, China (e-mail: shoumeng.ysm@antgroup.com; zhengyu.he@antgroup.com).

Jiannong Cao is with the Department of Computing, Hong Kong Polytechnic University, Hong Kong, China (e-mail: csjcao@comp.polyu.edu.hk).

Fengwei Zhang is with the Department of Computer Science and Engineering, and Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: zhangfw@sustech.edu.cn).

Digital Object Identifier 10.1109/TDSC.2023.3334277

I. Introduction

GPUS are now widely used in general- and high-performance applications such as 3D games [1], video processing and compression [2], mobile Virtual Reality [3], and neural network training and inference [4], [5], [6]. In addition, GPUs are used not only in server and cloud environments [7], [8], but also in small embedded systems [9], [10] such as smartphones and autonomous vehicles to satisfy the sharply-increasing performance demands.

As GPUs have enjoyed increased popularity and distribution, the associated security implications have not yet seen a corresponding level of scrutiny from the community. To access sensitive data processed by victim applications, an attacker can exploit numerous vulnerabilities at the OS level to gain control of the GPU Driver, which in turn enables access to the GPU's memory through Memory-mapped I/O (MMIO) interfaces. In addition, the attacker can break isolation between GPU applications by tampering with the GPU page table, leaking potentially sensitive data processed on victim GPU applications. Combined with the increase in the use of personally identifiable information [11], [12], [13] and sensitive secrets computed with GPUs [14], [15], there is an urgent need to address trusted computing requirements for ubiquitous GPUs.

Researchers and commercial vendors have proposed a number of approaches to defend against leaking sensitive data [16], [17], [18], [19]. Recently, one such technology is Trusted Execution Environments (TEEs) [19], [20], [21], [22]. By using specialized hardware and software, TEEs provide an isolated runtime environment for executing security-critical code. TEEs have recently been adapted to isolating secure GPU computation [23] using modified Intel Software Guard eXtensions (SGX) [21], Graviton [24] and HETEE [25] with customized TEEs. However, none of these techniques have been applied to Arm endpoint GPUs. One critical limitation lies in architectural differences between Intel and Arm GPU platforms. State-of-the-art GPUs on Intel-based devices are naturally isolated because discrete GPU devices have dedicated memory. In contrast, Arm-based devices often employ Systems on Chip (SoCs) in which a unified memory is shared between the GPU

© 2023 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

and CPU (and consequently, with an untrusted OS). This major change in architectural assumptions heavily influences the design of relevant protection mechanisms. In addition, several works [23], [24] involve highly-coupled software stacks (e.g., GPU Driver and runtime). This line of work requires porting heavy software to the enclave, which executes on behalf of the protected confidential GPU application. However, this may increase vulnerabilities within the system. On one hand, large software stacks increase the trusted code base of the enclave/TEE. On the other hand, the implementation of such ported software can be vulnerable [26], [27], [28], [29], [30], which severely threatens data security during computation. Finally, GPU TEE mechanisms [23], [24], [25] on Intel-based devices entail heavy hardware modification, which, if adopted to Arm devices, would result in poor compatibility with existing software systems. Existing secure computation on Arm endpoint GPUs requires porting the entire GPU driver into TrustZone and only focuses on specific applications (e.g., deep learning inference [31]). These defects have yet to be properly addressed. As for the defense of GPU chips, NVIDIA recently proposed the H100 GPU [32] to establish a trusted execution environment on GPUs, but this is not yet compatible with Arm endpoints.

We present StrongBox, the first GPU TEE for general computation on Arm endpoints. StrongBox aims to ensure secure and isolated computation on GPUs in Arm endpoints, which contains a unified memory with the untrusted OS and other peripherals. StrongBox achieves three key goals. *1) Security:* As a GPU TEE, StrongBox must isolate each secure GPU task from both the vulnerable system and malware. Thus, StrongBox prevents adversaries from leaking data or tampering with critical code during the life of confidential GPU applications. *2) Minimal TCB:* StrongBox must entail a minimal Trusted Computing Base (TCB) to reduce the potential attack surface. To achieve this goal, StrongBox delegates the heavy GPU Driver and GPU runtime code to perform complex operations, including memory allocation and deallocation, I/O, and task scheduling, while accessing sensitive data is strictly controlled by thin trusted components. *3) High Compatibility:* StrongBox maintains compatibility with ubiquitous Arm endpoint devices. In particular, StrongBox neither relies on the features of specific Arm endpoints, nor does it require hardware modification to GPU or CPU chips.

We discuss our prototype implementation of StrongBox using an Arm Juno R2 development board with clusters of Cortex-A53 and Cortex-A72 processors with a Mali-T624 GPU, both of which share a unified memory space. Our prototype introduces a TCB which is orders of magnitude smaller than the state-of-the-art approach [31] of porting a 30 K LoC Arm Midgard GPU Driver to TEE. We measure the performance of our prototype using a popular GPU benchmark suite, called Rodinia [33], which has been widely used to evaluate the performance on Arm devices [34], [35], [36]. We also analyze and discuss the security of StrongBox under an assumed adversary. Our evaluation results indicate that StrongBox successfully achieves its security guarantees while introducing a reasonably low (4.70%–15.26%) performance overhead. Moreover, we measure the slowdown caused by TCB reduction with OP-TEE xtest [37]. Results indicate that our prototype introduce a negligible overhead to *secure world*.

We claim the following contributions in this work:

- We present StrongBox, the first GPU TEE that runs on Arm endpoints. StrongBox provides an isolated execution environment for secure tasks and protects sensitive data and code from a compromised kernel.
- We further minimize the TCB of our design by considering the threat from *secure world*. We leverage a novel technique, called secure virtualization extension, to defend against compromised Secure OS's and applications while introducing negligible performance overhead.
- We implement a prototype of StrongBox on an Arm development board without any hardware or architecture modification. We further implement the TCB reduction mechanism based on this prototype.[1]
- We perform a comprehensive evaluation of StrongBox and present a detailed security analysis of our prototype. Results show that StrongBox effectively protects the sensitive data with a comparable performance overhead.

This article is an extended version of our previous work [38] accepted in ACM Conference on Computer and Communications Security 2022. Based on that work, we further reduce the TCB size by defending against vulnerable implementations of *secure world*. To handle potential attacks from this new threat model, we leverage a novel Arm feature, called secure virtualization, to restrict illegal access from the compromised secure OS and secure applications. To verify the feasibility of our defense mechanism, we prototype it on Arm Fixed Virtual Platforms (FVP) [39], which provides the official software-simulated Arm features and is widely used in other TEE works [40], [41], [42], [43]. We also measure the performance overhead of our prototype on Arm Juno R2 development board. Our results show that our approach introduces negligible performance overhead.

Comparison to Previous Work: We compare our new GPU TEE design with previous work. First, the major difference is that this improved GPU TEE design reduces the TCB by removing the vulnerable *secure world* software. The previous iteration of StrongBox trusts any component in *secure world*, thus exposing a large attack surface. Second, we design a lightweight approach to reduce TCB size by leveraging the new Secure Virtualization Extension proposed in Armv8.4, which is compatible with future Arm devices. However, since our native GPU TEE is designed for traditional Armv8 endpoints, it requires non-trivial modification on *secure world* software to reduce TCB.

We claim the new contributions in this article below:

- We reduce the TCB of our previous work by removing vulnerable *secure world* components (i.e., secure OS and secure applications). During the execution of confidential GPU applications, we prevent illegal accesses to sensitive data and to the StrongBox runtime from an attacker who controls *secure world*.
- We design a practical defense mechanism of vulnerable *secure world* OS/applications by leveraging the novel secure Stage-2 translation. Our design effectively reduces the

[1][Online]. Available: https://github.com/Compass-All/CCS22-StrongBox

TCB with no hardware modification or changes to *secure world* components.
- We implement a prototype of our defense mechanism using Arm FVP and measure the performance overhead on an Arm Juno R2 development board. Our evaluation shows that the new StrongBox implementation effectively defends against the threat of compromised *secure world* OS/applications while introducing negligible performance overhead.

II. Background

A. Arm TrustZone

Arm TrustZone [44] is a hardware-based security mechanism that provides a number of isolation guarantees for security-critical code on Arm devices. TrustZone isolates the execution into two states: (1) *secure world*, which provides a TEE for secure applications or secure OS, and (2) *normal world*, which is used for untrusted applications or traditional OS. To provide the strict resource and computing isolation between the secure and normal world, TrustZone deploys its firmware on a secure monitor. The secure monitor acts as the gatekeeper during the switching of these worlds. Moreover, it owns a higher privilege than the secure or non-secure components to protect the configurations. Thus, confidential computation within *secure world* is strictly protected by TrustZone via hardware isolation in memory, and can be requested in the normal world through several mechanisms, such as a privileged `smc` instruction.

The isolation of the normal and secure worlds is ensured by hardware components that are parts of the TrustZone architecture. One such component is the TrustZone Address Space Controller (TZASC). Embedded in the memory bus, the TZASC sits between DRAM and CPU/peripherals, monitoring access to secure and non-secure address spaces. Moreover, the TZASC assigns a Non-Secure Access Identity (NSAID) to each untrusted peripheral device. When a peripheral requires read/write access to an address, the TZASC looks up the configuration (usually stored in a register) of the corresponding memory region for the validity of the access. However, the TZASC only supports configuring 8 regions, limiting flexibility of such a memory protection mechanism. We present an assisted access control mechanism in Section II-B to address this limitation.

TrustZone also isolates interrupts to varied groups in response to device I/O by configuring a Generic Interrupt Controller (GIC) [45], [46]. For instance, GICv3 [46] creates two groups of interrupts: Group 0 is assigned to the secure monitor, while Group 1 owns both secure and non-secure interrupts, which are assigned to secure OS and non-secure components, respectively. TrustZone identifies the interrupt and its group when they occur, in turn dispatching the interrupt to the CPU with the related security state. In this article, we control the switching of GPU interrupt state to efficiently process sensitive data and restore the environment.

B. Arm Address Translation

Arm defines a two-stage (formally called Stage-1 and Stage-2) translation mechanism to map the memory space of OS and applications within physical memory. Stage-1 translates the virtual address (VA) of kernel or user space into an intermediate physical address (IPA), and Stage-2 maps the IPA to the real physical address (PA). Stage-2 translation is widely supported on Cortex-A series [47], [48], [49], [50] chips, which is the mainstream processor for GPU-equipped Arm endpoints. However, most Arm endpoints disable this translation since they do not typically fit multi-tenant hypervisors. In StrongBox, we enable this feature for page-level access control on the GPU MMIO registers and the GPU task memory.

C. Workflow of Arm Endpoint GPUs

To control endpoint GPUs at the software level, Arm provides two GPU software stacks: (1) the closed-source user runtime in the user layer (e.g., OpenCL [51]), and (2) the open-source GPU Driver in the kernel layer. The user-level runtime provides various high-level APIs, built-in functions, and specific data structures to support developing GPU applications. The kernel-level GPU Driver mainly controls memory allocation and task scheduling and submission via Memory-Mapped Interfaces (MMIO).

A GPU application is composed of one or more GPU tasks, which further contain several GPU threads. We present the typical execution of a GPU application on an Arm endpoint GPU as follows. First, the GPU software stacks allocate memory for the essential components in GPU tasks (i.e., GPU buffers, code segments, and non-confidential metadata) and build the corresponding GPU page table. Next, data are loaded into the allocated GPU buffers through a Direct Memory Access (DMA) controller. Then, the GPU software stack loads the binary code into GPU memory. After that, the GPU task start command is sent by configuring the GPU MMIO registers. After receiving the submission command, the GPU computes the task based on the code and data, and stores the execution result in specific memory. Once the GPU task is finished, the GPU sends a hardware interrupt to notify the interrupt handler in the GPU software stack. For multi-task GPU applications, the GPU software stack repeatedly loads the task code, submits the task, and waits for completed GPU computation. After processing all tasks, results are directly accessed or exported through DMA. Based on the general workflow of Arm endpoint GPUs, StrongBox secures the task execution and builds a secure data path for data transfer.

D. Arm Secure Virtualization Extension

Since Armv8.4 [52], Arm provides a novel *secure world* feature, called secure virtualization extension. This architecture extension introduces a secure hypervisor layer (i.e., S-EL2) to manage multiple secure OSes and applications, which is achieved by virtualizing the entire *secure world* (e.g., secure memory and secure peripherals). Although current Arm endpoints are not intended for deploying multiple secure OSes, it still provides several features to manage the secure OS. Similar to *normal world* virtualization, S-EL2 introduces secure Stage-2 translation to manage the memory used by secure OSes/applications. The secure Stage-2 translation is fully controlled by the secure hypervisor and the secure monitor, preventing low-privileged attackers from modifying critical registers to disable it. In contrast to non-secure Stage-2 translation, secure Stage-2 translation

owns several dedicated system registers to configure the base address of translation page table (VSTTBR_EL2) and translation control (VSTCR_EL2), though both secure and non-secure Stage-2 translation share the same register (HCR_EL2) to determine whether they are enabled.

In StrongBox, we correctly configure the secure Stage-2 translation to control the memory access from vulnerable secure OSes/applications. To prevent unauthorized access, we configure the page table of secure Stage-2 translation and invalidate the mapping to the protected region. Thus, memory access from secure OSes/applications to the protected region yields a translation fault instead of the sensitive data. Moreover, the secure OSes/applications cannot disable the Stage-2 translation due to the lack of privilege. We leverage the secure virtualization feature to reduce StrongBox's TCB size.

III. Threat Model and Assumptions

We assume a privileged attacker who seeks to leak or tamper with sensitive data and execution results of GPU applications. Specifically, the attacker can control the kernel as well as the entire GPU software stacks, including the GPU Driver, runtime, and other peripheral drivers. To tamper with sensitive data and code in GPU applications, the attacker can directly access a unified memory used for GPU tasks, or control peripherals to subvert detection. In addition to direct access, the attacker who controls the GPU driver can compromise the memory management of the GPU applications, mapping sensitive data to an unprotected region. We also consider an adversary aiming to break the isolated execution environment of the victim GPU applications, such as submitting an arbitrary number of malicious tasks. By modifying the corresponding GPU page table, the attacker can require malicious tasks to access the memory of the victim task. Furthermore, we consider threats from the *secure world* OS and applications. The attacker who exploits vulnerabilities [26], [27], [28], [29], [30] of secure OS and applications can control these components to directly access our TEE. In addition, the attacker can tamper with StrongBox runtime (e.g., modifying the configuration of TZASC) to bypass the access control, which is shown in recent study [53]. Following existing best practices for SGX-based GPU TEEs [23], we assume the GPU, TrustZone, and their firmware are trusted since they can be guaranteed by secure boot and attestation from a trusted remote host. Thus, StrongBox firmware is correctly loaded into Arm endpoints with verification. In addition, we consider cryptographic-based attacks, physical attacks, side-channel/spy attacks, and the Denial-of-Service attacks to be beyond the scope of this article.

IV. Design

A. Goals and Overview

The goal of StrongBox is to achieve an effective, lightweight, and compatible GPU TEE on Arm endpoint devices, in which the OS and applications are potentially compromised. As a result, our design must achieve three critical goals described below.

G1. Provide a Trusted Execution Environment for secure GPU tasks: The primary goal is to secure sensitive data for GPU applications. To achieve this goal, StrongBox must protect two modes of data access from Host OS to the execution environment: (1) from the OS to GPU and (2) from the OS to the memory of GPU tasks. In the former case, StrongBox diverts the control flow of the GPU from the untrusted GPU Driver to TrustZone's *secure world*, including the interaction with GPU registers and GPU interrupts (see Section IV-B). For the latter case, StrongBox manages the access to the unified memory to restrict untrusted access to the task execution environment (see Section IV-C).

G2. Reduce the size of trusted computing base: Next, we must maintain a lightweight TEE. Several GPU TEEs and secure computing systems [23], [24], [31] trust large software stacks (e.g., libraries and drivers) for pre-processing sensitive data, exposing a large attack surface within the TEE. However, we observe that the software stack can perform its critical functions (e.g., memory management of GPU tasks and scheduling GPU tasks) without direct access to the sensitive data. Thus, we instead preserve the GPU Driver in *normal world*, while introducing a lightweightStrongBox runtime that protects GPU memory even if the driver is compromised. This design achieves a thin TCB without undermining the security of the existing system. In addition, considering the secure OS and applications can be totally compromised, or curious about the sensitive data, we regard them as untrusted and design a defense mechanism against the *secure world* attacker (detailed in Section IV-D).

G3. Ensure compatibility with Arm endpoints: Third, we introduce a GPU TEE designed for Arm endpoints with minimal changes to the underlying platform. State-of-the-art GPU TEEs [23], [24], [25] adopt additional hardware components to ensure secure computation. These specialized hardware requirements increase challenges associated with migrating systems as well as the associated production costs. Thus, we design our approach to rely neither on specialized hardware components nor physical modification of devices.

Fig. 1 illustrates the design of StrongBox, which is divided into software and hardware components. StrongBox reuses the GPU runtime and driver software in the OS (EL1) to reduce the overall TCB size (**G2**), and additionally provides two principal components: GPU Guard and Task Protector. GPU Guard provides a protective layer that ensures the GPU can execute in isolation, and ensures that secure tasks are completed before final computed results are returned. Task Protector works in tandem with GPU Guard to ensure that sensitive data are protected to provide confidentiality. As a result, our approach ensures that GPU can execute secure tasks in isolation while executing within a potentially-compromised OS. Note that hypervisors in both *normal world* and *secure world* are not deployed on most Arm endpoints, and StrongBox requires no modification to secure applications and secure OS. For hardware components, StrongBox leverages existing and software configurable devices to ensure high compatibility (**G3**). We split the system's memory into four regions: Two untrusted regions, which we call (1) Normal RAM and (2) Non-secure Task RAM, which are respectively used for kernel and non-secure tasks; and two trusted regions that we call (3) Trusted RAM, which is reserved for the StrongBox runtime and non-secure Stage-2 translation table, and (4) Secure Task RAM, which is a fixed, non-secure memory region reserved

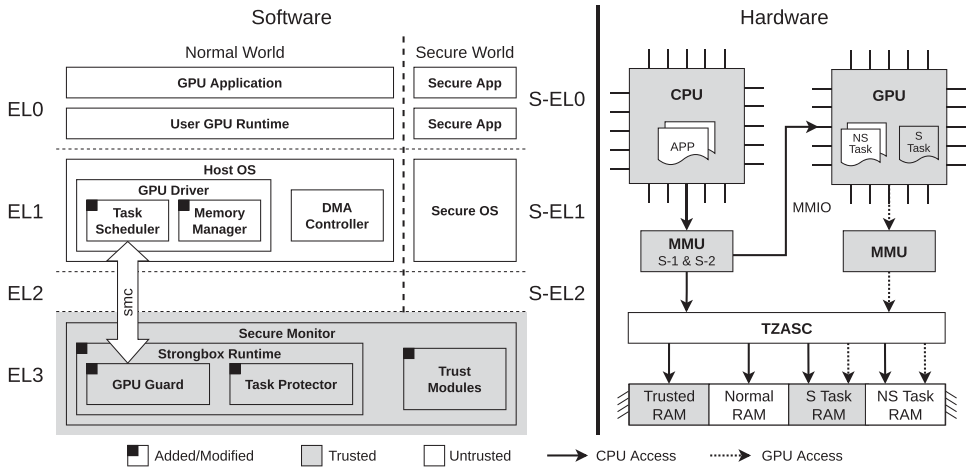

Fig. 1. StrongBox architecture overview.

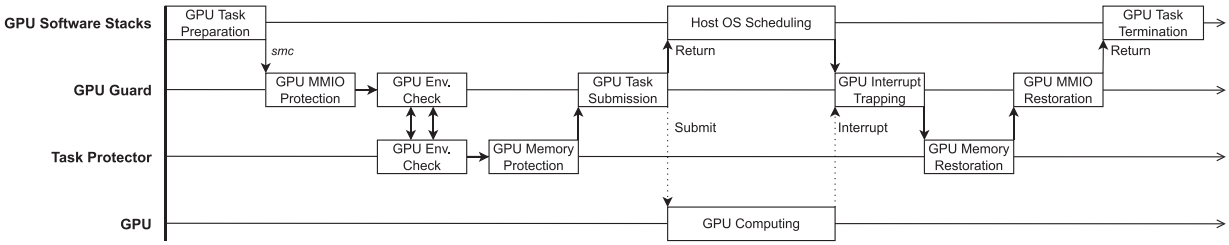

Fig. 2. Lifecycle of a secure task in StrongBox.

for the confidential GPU application to dynamically request memory and create GPU page table mappings. To protect the two trusted regions, StrongBox leverages the Memory Management Unit (MMU) and a specially-configured TZASC. In the MMU, StrongBox performs non-secure Stage-2 translation to control access from the Host OS to GPU MMIO interfaces and to the two trusted regions. Meanwhile, we leverage the TZASC to control access to the two trusted regions from GPU and other peripherals.

Lifecycle of a Secure Task: Fig. 2 illustrates the lifecycle of processing a secure task using StrongBox. Initially, the GPU software stack prepares the secure task and generates an `smc` event, which the GPU Guard handles for further protection. Next, the GPU Guard collaborates with the Task Protector to secure GPU MMIO and GPU memory, then to verify the GPU environment. Once the GPU environment is protected, the GPU Guard submits the secure task to the GPU device. Note that StrongBox does not block the CPU core during GPU computation. When the GPU completes computation, the GPU Guard handles the resulting GPU interrupt, again working with the Task Protector to restore the GPU environment. Finally, the GPU Guard routes control flow to the GPU software stack to terminate the secure task.

B. GPU Exclusivity During Critical Execution

StrongBox's primary security goal is to provide exclusive execution of secure GPU tasks. That is, if any secure task is executing on the GPU, no other task can be scheduled on the GPU simultaneously. As shown in Section III, attackers who control the GPU MMIO can subvert the isolated execution environment. To defend against such an attacker, we adapt state-of-the-art GPU TEEs to Arm GPU devices, which requires addressing two issues. First, while existing GPU TEEs migrate heavy GPU driver code into an enclave or TEE to control the GPU, we keep this code in the untrusted kernel, and instead react to specific `smc` events that route control to the StrongBox runtime. Second, we use existing Arm features to control the access of GPU hardware — specifically, the non-secure Stage-2 translation helps lock the system mapping of GPU MMIO registers during computation. By leveraging custom `smc` event handlers and non-secure Stage-2 translation, we can prevent highly-privileged attackers from gaining control of the GPU or executing malicious tasks.

StrongBox reuses the existing GPU driver in the untrusted kernel, and instead secures execution using lightweight software components. To achieve this, we must work with the GPU driver to reroute control under several cases related to the creation, management, and execution of secure GPU tasks. First, we design a dedicated scheduling rule for secure tasks. Once a secure task is ready to execute, any non-secure computation, are forced to reschedule and wait for the completion of the submitted secure task. For the running tasks, the GPU driver repeatedly evaluates the contents of GPU registers to determine if any tasks are executing. Once we determine that the GPU is not executing any task, the GPU driver uses a dedicated `smc` call which

signals for the protection and security check in StrongBox runtime. In contrast, normal, non-secure tasks can submit as usual.

Recall that the GPU driver is untrusted because it is part of the untrusted OS — however, we can mitigate attacks that compromise the GPU driver. When we receive an `smc` call, we use our GPU Guard to detect and eliminate threats. GPU Guard defends against these attacks by isolating and securely introspecting the execution environment. Before submitting secure tasks to the GPU, GPU Guard confines the access to GPU MMIO via the non-secure Stage-2 translation to prohibit unauthorized access from the untrusted OS. Any malicious operations against the GPU MMIO interfaces (e.g., modifying GPU registers or submitting a task) are captured by generating page-fault exceptions, while trusted operations in StrongBox are not affected. After locking the GPU MMIO, GPU Guard guarantees the GPU environment security. First, GPU Guard checks the GPU task state registers to ensure no hidden tasks. The checked results are unable to be tampered with by TOCTTOU attacks due to the locked GPU MMIO. Next, GPU Guard works with Task Protector to further check the other critical GPU registers (e.g., page table base register and GPU task code register). Task Protector also checks the memory containing the loaded task's GPU page table, code, and data regions described in Section IV-C. The page table memory is locked and checked by StrongBox before the first secure task executes and is unlocked after completing the last secure task. The check prevents an attacker from mapping the sensitive GPU buffer addresses into out-of-control memory. In addition, we perform integrity checks for the code and data regions before submitting each secure task to the GPU. Meanwhile, to handle GPU interrupts in StrongBox, the security state of GPU interrupts is switched from non-secure to secure via GIC. Finally, GPU Guard submits the prepared tasks to the GPU through writing tasks submission register. Then, the GPU will carry out the prepared task as expected. After submitting the secure task, StrongBox returns to the GPU driver and releases the CPU. Therefore, StrongBox does not block the CPU core during GPU computation. For secure task synchronization, StrongBox requires the GPU driver to schedule the GPU tasks to be submitted, while it does not support the concurrent submission since mainstream Arm endpoint GPUs [54] and related SDKs [55] have yet to support the concurrent computation of GPU tasks (as mentioned in Section II-C). Moreover, when processing, StrongBox does not block other `smc` calls that do not interact with the GPU.

Secure Termination: When computation completes, the GPU sends an interrupt (which is previously configured as secure) to notify StrongBox. Thus, the StrongBox runtime intercepts the GPU interrupt and performs secure termination, which consists of three steps: First, the Task Protector encrypts plaintext data that are ready to export. The Task Protector also restores the corresponding non-secure Stage-2 translation. Note that we preserve the translation protection for the remaining data since they are computed in the following secure tasks. Second, for the final secure task of a confidential GPU application, StrongBox verifies the entire Stage-2 translation to detect whether the protection of any GPU task region has yet to be removed. If any exist, StrongBox erases the plaintext in these regions and restores the protection in Stage-2 translation and TZASC. Third, to return control of the GPU to the GPU driver, the GPU Guard configures the non-secure Stage-2 translation to restore the GPU MMIO. It also configures the GIC and switches the GPU interrupt back to non-secure state to allow the GPU driver to handle the interrupt. After the secure termination process, the GPU is allowed to process new tasks.

C. Dynamic and Fine-Grained Protection

StrongBox must ensure the confidentiality of sensitive data and the integrity of secure GPU tasks that store data in the Secure Task RAM. Thus, an attacker may try to access the unified memory that stores sensitive data inside the GPU buffer. Alternatively, an attacker may attempt to modify the GPU page table entries (PTEs), exporting sensitive data to unprotected regions. To guarantee the security, a straight-forward method is to statically protect the entire task memory with one or more TZASC slots. However, this leads to two challenging issues. First, such static protection can severely undermine the functionality of the GPU driver. For instance, it prevents the GPU driver from processing the non-confidential metadata of the secure tasks. Second, the layout of sensitive data and code are physically scattered and dynamically-changed in Secure Task RAM for different GPU applications. Thus, static TZASC partitions may not work in our unified memory scenario where memory management must be flexible. Another solution based on existing Arm-based secure computing [31] is to port the GPU software stacks into TrustZone; however, this incurs large TCB and breaks our design principle of minimal TCB. Thus, we need an alternative to using static TZASC partitions.

Instead, we develop a dynamic and fine-grained memory protection mechanism by combining the non-secure Stage-2 translation and TZASC. We explicitly divide the Secure Task RAM into two physically continuous regions: Task region and GPU page table region. For the Task region, the non-secure Stage-2 translation dynamically performs page-level protection to critical memory containing data and code in different stages, and we use a TZASC slot to manage access from DMA, GPU, and other peripherals. As for the GPU page table region, StrongBox employs the non-secure Stage-2 translation to monitor modification requests. To avoid potential peripheral attacks, we further leverage TZASC to prohibit write access from peripherals to the GPU page table region. We categorize access permission of these two regions into six types:

- *Full Accessible:* Allow any read/write operations.
- *Write Protected:* Allow the read operations from any component, but monitor the write operations.
- *DMA Prohibited:* Disallow the read/write operations from peripherals through DMA.
- *OS-DMA Prohibited:* Disallow the read/write operations from both OS and peripherals through DMA.
- *GPU-DMA Prohibited:* Disallow the read/write operations from GPU and peripherals through DMA.
- *OS-GPU-DMA Prohibited:* Disallow the read/write operations from OS, GPU, and peripherals through DMA.

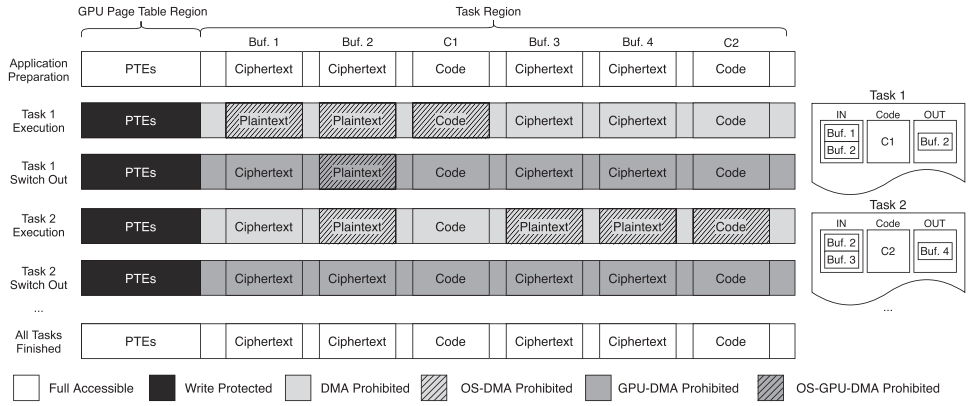

Fig. 3. Changes of access permissions on Secure Task RAM when a confidential GPU application is executed. Task 1 and Task 2 are two example tasks inside the application. Task 1 contains input Buf. 1 and Buf. 2, output Buf. 2, and code segment C1. Task 2 contains input Buf. 2 and Buf. 3, output Buf. 4, and code segment C2.

Fig. 3 illustrates the evolution of access permission during the life-cycle of a confidential GPU application. The initial access permission of the Secure Task RAM is configured as *Full Accessible* to allow preparing an application for submission via the GPU software stack. During task execution and switching, the GPU page table region is configured as *Write Protected* to avoid potential leakage of sensitive data. Task Protector traps modifications to the GPU page table and introspects any malicious memory mappings, (e.g., double mapping and mapping to untrusted regions). Moreover, since the GPU page table is initially prepared by the GPU driver, Task Protector verifies the entire page table before running the first secure task. As for the task region, access permissions for each GPU buffer and code region can vary. Upon execution of a secure task, we configure the entire task region as *DMA Prohibited* except the memory of the executed task, which is *OS-DMA Prohibited* to secure the subsequent encryption and integrity verification of code and data regions. During the task switching, the GPU buffers are encrypted by default (e.g., Buffer 1 in Task 1, and Buffer 2, 3, and 4 in Task 2). For any buffer that is used in subsequent secure tasks (e.g., Buffer 2 in Task 1), StrongBox supports retaining plaintext and configures the plaintext memory as *OS-GPU-DMA Prohibited*, and all memory except the plaintext data memory is configured as *GPU-DMA Prohibited* until the submission of next secure task. After all tasks are finished, all sensitive plaintext data are encrypted, and the entire Secure Task RAM is configured as *Full Accessible* to allow the user to load the result. Furthermore, for security purposes, StrongBox prevents secure tasks in different confidential GPU applications from sharing the Secure Task RAM. Any task in other confidential GPU applications cannot start until the previous confidential GPU application finishes all secure tasks and safely terminates. This organization of memory provides strong isolation guarantees for any sensitive data that is used by a secure GPU task.

Next, we design a secure data path to avoid data leakage. Any sensitive data transferred via DMA requires encryption and integrity checks using Hash-based Message Authentication Code (HMAC). Task Protector performs secure introspection to decrypt or encrypt the sensitive data with the shared keys, and calculates each HMAC according to the plaintext data or code. Since the memory of secret keys, intermediate or plaintext data, and the corresponding task page table are protected by our non-secure Stage-2 translation and TZASC mechanism, TOCTTOU attacks against computed hash values are infeasible. Next, Task Protector notifies GPU Guard to continue with task submission, or abort it due to verification failures. When a secure GPU task is finished, Task Protector restores the execution environment. The executed results are encrypted and hashed before reverting to a non-secure state, while plaintext data exist only during secure task execution.

Key Management: We further discuss the key management process in StrongBox. StrongBox requires the sensitive data and execution results to be encrypted during data transfer and storage because they are trivially accessed by an adversary who controls the untrusted OS. To guarantee data security, StrongBox does not provide a fixed pre-shared key to the endpoint user. Instead, we follow a key exchange approach (e.g., Diffie-Hellman key exchange [56]) with the user to obtain the shared keys, which are further used to encrypt the data and calculate the HMAC. To complete the key exchange process, we assume a public key infrastructure, including a public/private key pair and certificate, is installed into the secure monitor by the vendor. With this infrastructure, the user encrypts an AES key with the certified public key. Next, StrongBox runtime in the secure monitor receives the encrypted AES key and decrypts it with the private key. We allow the GPU tasks within one confidential GPU application to share the same AES key, but we must establish a new AES key for each subsequent confidential GPU application. Based on this, the key management modules ensure data confidentiality during data transfer and storage.

D. TCB Reduction

Besides maintaining a thin TCB by reserving the GPU software stacks in *normal world*, we further reduce the TCB by focusing on the secure OS and secure applications. Our insight is that the size of *secure world* has largely increased in the past decade. For instance, the size of the widely-used secure OS, OP-TEE [57], has extended from 60 K to 300 K LoC. Although such increment satisfies various architecture features

and user requirements, it also exposes a large attack surface. The increasing records of vulnerabilities [58], [59], [60] further demonstrates that secure OSes and applications cannot be completely trusted. Moreover, recent work [53] shows that the secure OS has excessive privilege to access unauthorized regions such as the non-secure memory and the firmware of the secure monitor. To that end, *secure world* can potentially be compromised by secure world attackers, compromsing integrity and security, including the GPU computation from our previous StrongBox model. Unfortunately, our previous StrongBox prototype and most TEE designs are vulnerable to such attacks since they impliclty trust *secure world*. In addition, TZASC and the non-secure Stage-2 translation are unable to defend against the vulnerable secure OSes and secure applications. One possible solution is to interfere with the memory management in a secure OS (e.g., removing the mapping to sensitive data). However, this partially influences the functionality of secure OSes and generates moderate overhead.

The Armv8.4 architecture contains a novel secure virtualization feature, which provides a potential solution to defend against such attacks. Although virtualizing *secure world* may not be intended for Arm endpoints, we leverage the access control mechanism, called secure Stage-2 translation, to restrict unprivileged access from secure OS and secure applications. This mechanism is initially provided on Armv8.4 or later, but can be disabled due to the lack of a secure hypervisor. Thus, in our design, we enable secure Stage-2 translation and create a translation table to protect the sensitive data and the StrongBox runtime against such a *secure world* attacker. Compared with establishing a heavy hypervisor, our mechanism is lightweight with low performance overhead.

To leverage the secure Stage-2 translation, we first reserve a memory region for the translation table and allocate it with a TZASC slot. The TZASC mainly confines the access from the peripherals and non-secure CPU. However, it is infeasible to confine the *secure world* components. Thus, we further create a translation table to protect these components. Specifically, we invalidate the mapping to the physical address of the secure Stage-2 translation table. If the attacker intends to bypass such translation by modifying the translation table itself, secure Stage-2 translation traverses the existing table and obtains an invalid entry. Thus, the modification is terminated by a translation fault.

Once protecting the translation table, we next secure GPU computation. Specifically, we must secure (1) the exclusive execution environment and (2) the sensitive data from the *secure world* attacker. For the first aspect, we leverage the secure Stage-2 translation to invalidate the access to GPU MMIO, StrongBox runtime and the MMIO to TrustZone components (e.g., TZASC). Thus, the attacker cannot leak sensitive data by tampering with the runtime configuration. For the second aspect, we secure the sensitive code, data, and page table entries in Secure Task RAM. Since the Secure Task RAM is reserved for GPU computation without interaction with *secure world*, we statically invalidate the mapping of the entire region. Note that we also secure the shared keys and the temporary data in cryptographic operations.

V. Implementation

We implement a 64-bit StrongBox prototype on an Arm Juno R2 development board [61] with 8 GB DRAM, an official Mali-T624 GPU, and the Arm TrustZone extension. We use Linux v4.14.59 with an open-source Midgard GPU Driver [62] in *normal world*, and run Arm Trusted Firmware (ATF) v2.1 in secure monitor. To create an isolated execution environment, we reserve 264 MB as Secure Task RAM, including a 256 MB region (`0xB0000000–0xBFFFFFFF`) to hold secure tasks and a 8 MB region (`0xAF800000–0xAFFFFFFF`) for GPU page table. The Trusted RAM contains the memory space for ATF and an additional 4 MB region (`0xA0000000–0xA03FFFFF`) for the non-secure Stage-2 translation table. In ATF, we deploy StrongBox runtime to configure two hardware components: TZC-400, which is an implementation of TZASC, and GIC-400, which handles the GPU interrupts. To setup the non-secure Stage-2 translation, we create a flat mapping for the entire memory region except the Trusted RAM. In addition, three major registers (`HCR_EL2`, `VTTBR_EL2`, and `VTCR_EL2`) are configured to enable the translation, thus providing an important mechanism for securing sensitive data used in sensitive applications. We also secure GPU tasks using cryptographic and integrity checking operations. We assume that TrustZone has established a key management system and a communication channel with the user. These steps can be achieved following previous work [63], [64] and we do not claim this as a contribution of our work. We use Advanced Encryption Standard (AES) encryption with a 128-bit key for cryptographic operations on the sensitive data. For integrity verification, we use the SHA-256 algorithm to compute hashes of various data. These operations can be accelerated using hardware-assisted instructions and SIMD extensions in Armv8.

The TCB reduction is implemented on the prototype above running OP-TEE v3.6.0 [57] in secure world. Since no off-the-shelf development board supports the S-EL2 feature, we prototype it using Arm FVP [39] with Armv8.4 extensions enabled. The implementation consists of two parts: (1) preparing the defense against secure OS and applications, and (2) enabling the defense mechanism when entering *secure world*. To prepare the defense, we first reserve a 2 MB memory region for the secure Stage-2 translation table. Since the available TZASC slots are limited, we reuse an existing TZASC slot to protect our secure Stage-2 translation table against untrusted peripherals. Specifically, we allocate the secure Stage-2 translation table in `0xA0400000–0xA05FFFFF`, which is adjacent to the non-secure Stage-2 translation table. Next, we extend the slot to cover both the secure and non-secure tables. As this slot still allows CPU access from arbitrary exception levels and security states, we further restrict both the secure and non-secure OS/applications to access each translation table. In addition, during the setup of the secure Stage-2 translation, we also invalidate the access to (1) Trusted RAM (2) Secure Task RAM, and (3) MMIO to GPU/TZASC/GIC, while creating a flat mapping to the remaining regions. As for enabling the defense mechanism, we configure the essential system registers (`HCR_EL2`, `VSTTBR_EL2`, and `VSTCR_EL2`) to enable the

secure Stage-2 translation. We implement the context switching for hypervisor-layer (including the S-EL2) registers since ATF has yet to support it on OP-TEE.

A. GPU Driver

To fulfill the protection policy for secure GPU tasks, we modify the `kbase_mem_alloc_page` function in the Midgard Driver to allocate pages of secure tasks in the aforementioned 256 MB region of Secure Task RAM, while the non-secure tasks take the remaining non-reserved DRAM space. Moreover, we find that the original Memory Manager in Midgard GPU Driver maintains a memory pool for GPU tasks, and requests additional pages from the kernel once the pool is exhausted. Therefore, we explicitly create an extra secure memory pool in the GPU driver to assign the reserved memory for secure tasks. We manage this pool with Contiguous Memory Allocation [65] (CMA) and use the `cma_alloc` function to allocate the page in reserved memory. In addition, we must guarantee that any two GPU buffers cannot share the same page. Otherwise, the protection restoration of one GPU buffer can lead to unintentional leakage to other GPU buffers on the same page. Unfortunately, this guarantee can be violated during buffer creation in the closed-source OpenCL library. To address this concern, we allocate an additional page for each GPU buffer and redirect the non-aligned buffer pointer to the next page-aligned address. In this way, we force the start address of all GPU buffers to be page-aligned, which ensures different GPU buffers do not share the same page. We further confirm page-alignment requirements are fulfilled with an additional check in our security modules.

Besides the Memory Manager, we modify the original scheduler in the GPU driver to assist to create the isolated execution environment of secure tasks. Upon the arrival of a submitted secure task, the Task Scheduler blocks and reschedules the submission of any other tasks via a lock. Next, the scheduler in StrongBox checks the GPU state registers and waits until all running GPU tasks are finished. Once the GPU is idle, the scheduler submits the secure task to GPU Guard and Task Protector for further protection.

B. GPU Guard

During the process of critical GPU applications, GPU Guard prevents unauthorized access to the GPU. Once it receives the specific `smc` call, it first configures the non-secure Stage-2 translation table entries to prevent any unauthorized access to GPU MMIO. Specifically, it sets the last bit of the corresponding non-secure Stage-2 PTEs as 0 to invalidate the mapping of GPU MMIO regions, then invalidates the TLB entries for each CPU core. For secure Stage-2 translation, we also configure the corresponding PTEs as 0 to invalidate the access from secure OS and secure applications. Thus, the attacker who attempts to access GPU registers through the GPU MMIO will fail in a translation fault. To switch to secure execution, StrongBox leverages the GPU driver to set the control and critical state register, then safely verifies critical registers. We follow the source code of the GPU driver [62] to sanitize critical registers, such as `JS_STATUS` (which shows the GPU state), `JS_HEAD_NEXT` (which stores the location of secure task code), and `AS_TRANSTAB` (which stores the GPU page table base) in StrongBox runtime. To submit a task, GPU Guard writes a start command (`0x1`) to the `JS_COMMAND_NEXT` register. To intercept the GPU interrupt, we use the GIC [45] to mark it as a secure interrupt. On our Juno board, the ID of the task complete interrupt is 65. Thus, we configure the `GICD_IGROUPR` register of this interrupt to the secure state (`0x0`). Once the secure task is complete, GPU Guard receives the interrupt, waits for the data process in Task Protector, and resets the interrupt to non-secure state (`0x1`) before returning to the OS.

C. Task Protector

Task Protector leverages both TZASC and the Stage-2 translation to restrict the access of Secure Task RAM, which contains the GPU page table and task regions. In the GPU page table slot, we reject writing operations from all peripherals and DMA by disabling most bits in TZASC `NSAIDW` registers except the bits CPU (`AP`). As for writing operations from the untrusted OS, we monitor modification through exceptions, and verify whether the writing operation is illegal in the exception handler. Besides protecting the GPU page table, we check the GPU page table base register `AS_TRANSTAB` for each secure task. In the task slot, we leverage the TZASC to manage read and write access from DMA, GPU, and other peripherals by configuring the corresponding bits in both the `NSAIDW` and `NSAIDR` registers. Moreover, random access to data and code from the untrusted OS is limited by dynamically changing the non-secure Stage-2 mapping with TLB invalidation. Any illegal read or write access to the code and data is prohibited by triggering the non-secure Stage-2 translation fault. As for secure Stage-2 translation, we invalidate the access to the entire Secure Task RAM including the task memory and the corresponding GPU page table.

As part of implementing access control, Task Protector performs cryptographic and integrity-checking operations for each secure task. Our prototype supports using the agreed-upon algorithm to perform this functionality, such as AES-128 algorithm for cryptographic operation and SHA-256 in integrity verification. However, we encounter two technical issues in verifying the code integrity of secure GPU tasks: (1) the task pointer does not simply point to the code segment, and (2) the code length is not given. For the former problem, we analyze the content pointed to by the task pointer via reverse engineering. We eliminate the flag bits in `JS_HEAD_NEXT` register and find the start address of the secure GPU task. Thus, we obtain the code pointer at the offset `0x138` of the start address. To calculate the code length, we leverage an unofficial study [66], [67] describing the instruction format. To further guarantee the execution order integrity of GPU tasks, we combine the task code contents with the task index to generate the code signature. Moreover, we generate the signature of output GPU buffers and provide the total number of executed secure tasks. This way, we verify the integrity of secure tasks code, execution order, and execution result.

VI. Evaluation

In this section, we evaluate our prototype of StrongBox based on our implementation (Section V). We consider four research questions in our evaluation:

TABLE I
CODE SIZE OF STRONGBOX

Component	Function	Lines of Code
GPU Driver	Non-secure S-2 Initialization	170
	GPU Driver	179
ATF	TZASC Initialization	8
	Cryptographic Operation	530
	Integrity Verification	148
	GPU Access Control	344
	Secure S-2 Initialization	170
	Hypervisor-layer Context Switching	147
	Other Configuration	175
Total		1,871

RQ1: How large is the TCB required for STRONGBOX?

RQ2: How much overhead is incurred on GPU benchmarks?

RQ3: How much overhead is incurred on system performance?

RQ4: How much overhead is incurred by the defense mechanism against the *secure world* adversary?

A. RQ1: TCB Size of STRONGBOX

Table I shows the code size of STRONGBOX reported by *cloc* [68], a utility that reports standard lines of code. Recall that the Trusted Code Base (TCB) consists of code that initializes and configures system registers and address translation, as well as cryptographic operations and access control. The code in the TCB implements our software modules as described in Section IV. STRONGBOX relies on Arm Trusted Firmware (ATF) to securely boot the device, perform remote attestation, and conduct other trust establishment operations. To reduce the attack surface, STRONGBOX's TCB does not include the large Arm Midgard GPU driver (approximately 30 K LoC), the OP-TEE secure OS (approximately 340 K LoC) with secure applications, and the OpenCL driver (32 MB). As a result, our TCB is orders of magnitude smaller than state-of-the-art GPU TEE systems that assume these are trusted. In contrast, in STRONGBOX, even if the driver and the secure OS become compromised, our security mechanism can still secure the sensitive data computed on the GPU.

B. RQ2: Evaluation on Rodinia Benchmarks

To demonstrate the runtime performance of STRONGBOX, we consider the Rodinia benchmark suite [33], which offers realistic workload scenarios to measure the performance of GPU computing. In total, we select six applications from the Rodinia suite: one lightweight application (K-Nearest Neighbor), three medium-weight applications (LU Decomposition, Pathfinder, and Hotspot 3D), and two heavy-weight applications (Gaussian and LavaMD). Together, these six applications cover a swath of use cases for Arm-based GPU devices that consume sensitive input, temporary, and output data that we can use our system to protect. In our evaluation, we directly load the encrypted input data into the GPU buffer and receive the encrypted output results. Thus, we apply the corresponding protection policy to allow the GPU securely process the plaintext data. Moreover, we slightly modify the Rodinia GPU application code by replacing a part of the original OpenCL APIs with our wrapped API to suit

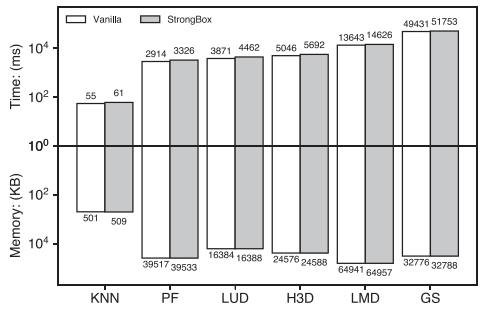

Fig. 4. Runtime performance on six Rodinia benchmarks.

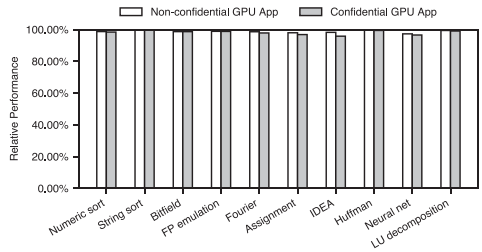

Fig. 5. Relative performance of Nbench applications when concurrently running a non-confidential/confidential GPU application.

STRONGBOX (e.g., adhering to our page-alignment requirement and creating a shared buffer to receive protection policies). Furthermore, we have checked that the GPU tasks inside the applications are executed sequentially, which is consistent with the GPU task execution flow in Section II-C. Fig. 4 shows a comparison between a system with and without STRONGBOX enabled across both execution time and memory consumption. It shows that STRONGBOX introduces low (4.70%–15.26%) overhead in across applications of varied sizes. As expected from Section V-A, the additional memory consumption from our page-alignment requirement is insubstantial since it is primarily attributable to the number of GPU buffers.

C. RQ3: Evaluation of System Performance

We select Nbench [69] to measure the system slowdown caused by STRONGBOX, which is widely used to measure the performance of CPU computation and memory intensive operations [43], [70], [71]. To demonstrate the system overhead, we select a long-running application `LMD` from the Rodinia benchmark [33] to concurrently execute each Nbench application. The time elapsed during the `LMD` application is approximately half of each Nbench application but is mainly composed of GPU computation. We measure the performance degradation of Nbench applications when concurrently running the non-confidential and confidential `LMD` application.

Fig. 5 shows the normalized results of the Nbench applications, whose performance degradation are slight when running with both non-confidential (average 1.28%) and confidential GPU application (average 1.91%). Thus, STRONGBOX incurs a small performance degradation on system-wide computation, which is mainly explained by two reasons: first, STRONGBOX

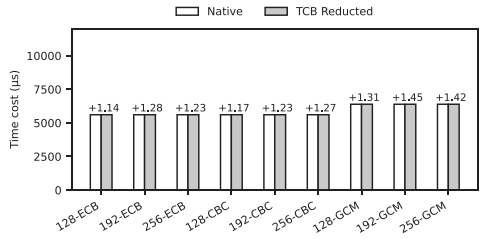

Fig. 6. Performance on AES benchmarks. Numbers are the overhead (in microsecond) introduced by the TCB reduction.

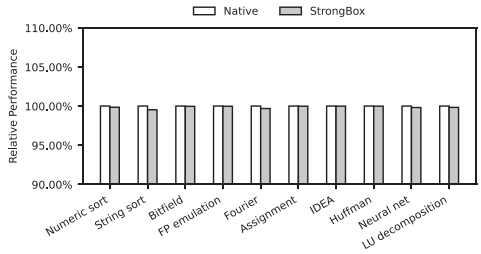

Fig. 7. Relative performance on Nbench applications. Note that we leverage the non-secure Stage-2 translation to emulate the performance overhead of the secure Stage-2 translation.

releases the CPU resource during the GPU computation, which is the primary time cost in most GPU applications; second, StrongBox does not block other CPU resources when processing the secure GPU tasks. Overall, the security benefits of StrongBox incur a small overhead to system-wide performance.

D. RQ4: Evaluation of the Defense Mechanism

Setup: Based on the design and implementation, our defense mechanism's performance overhead is composed of two parts: (1) the save and restore time of the hypervisor-layer registers when context switching, and (2) the runtime overhead in *secure world* when enabling the secure Stage-2 translation. Considering that the existing off-the-shelf development boards have yet to support the S-EL2 feature, we emulate its performance with non-secure hypervisor layer: to emulate the context switching of S-EL2 registers, we save and restore the corresponding non-secure registers (e.g., writing the `VTTBR_EL2` register to emulate restoring the `VSTTBR_EL2` register). As for the performance overhead introduced by secure Stage-2 translation, we port our defense mechanism to *normal world* and measure the time taken by non-secure Stage-2 translation. We measure the performance overhead on our Arm Juno R2 development board, and the results are detailed as follows.

Context Switching: We measure the performance overhead with an AES application from OPTEE *xtest* [37], which is widely used in other TEE works [53], [72], [73]. Specifically, we run three types of encryption (ECB, CBC, GCM), each of which is processed with three key sizes (e.g., the `128-ECB` indicates AES-ECB cryptographic operations with 128-bit key), while the other parameters (e.g., buffer size) are default. Fig. 6 shows the execution results. The overhead is relatively negligible (less than 0.1%) on the selected benchmarks. The major reason is that the context switching in each AES process is not frequent and takes a small proportion of the time in the entire AES application. In addition, the overhead is relatively stable (1.17–1.45 microseconds) when running benchmarks with different key lengths and cryptographic algorithms. This is because the execution times of context switching are fixed in each AES application.

Secure Stage-2 Translation: We next select Nbench application to measure the performance overhead incurred by the Stage-2 translation. Fig. 7 shows a normalized performance when enabling and disabling the defense mechanism. In total, our defense mechanism introduces a negligible overhead (on average 0.14%) when it is ported into *normal world*. Since the secure Stage-2 translation mirrors the features in non-secure Stage-2 translation, the results indicate that our defense mechanism incurs negligible overhead to *secure world*.

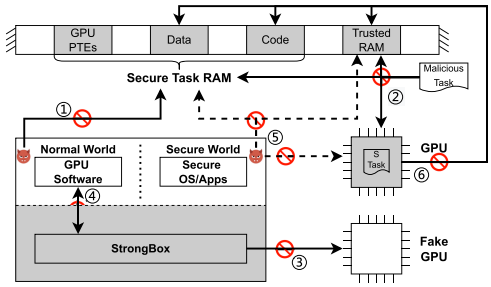

Fig. 8. Six attack scenarios against the confidential GPU application. ① indicates an attack on code, sensitive data, and GPU page tables in Secure Task RAM. ② represents an attack from malicious tasks. ③ represents an attack with a fake GPU. ④ shows Iago [74] attacks. ⑤ indicates the attack from the compromised *secure world* OS and applications. ⑥ indicates the misuse of secure tasks.

VII. Security Analysis

A. Attack on Secure Task RAM

Sensitive Data and Code: As shown in Fig. 8-①, a privileged attacker may attempt to directly access the sensitive data inside the GPU buffer during the execution of the confidential GPU applications. To defend against such data leakage, StrongBox designs a trusted data path between the user application and the GPU execution environment via both cryptographic algorithms and access control. The sensitive data are encrypted with the secret key exchanged between the users and StrongBox. Thus, an attacker without the secret key cannot leak sensitive data. For the subsequent decryption and verification in StrongBox, the plaintext regions are strictly protected by the TZASC and the non-secure Stage-2 translation, with which unauthorized access from the compromised OS or peripherals is restricted. Furthermore, she may terminate the GPU application early, temporarily leaving the plaintext data inside memory. However, these data are still protected. Next, she may attempt to create a malicious secure GPU task to steal the vestigial plaintext inside the latest victim GPU application that terminates unexpectedly. However, the secure termination check, which enforces cleanup of protected memory, is always performed before creating a new GPU application. Consequently, the confidentiality of sensitive data is fully maintained by the StrongBox. In addition, she

may tamper with task integrity by injecting malicious code or modifying provided data. To address this, StrongBox verifies the HMAC for the content in the secure task. If the provided signature fails to match the HMAC value, StrongBox terminates the application and clears the memory.

GPU Page Table: Fig. 8-① also shows that the attacker may subvert the GPU page table by double mapping or mapping the critical GPU address (e.g., GPU buffer) to an unprotected region. However, since the page table is strictly protected when processing secure tasks, the malicious mappings are detected before computing secure tasks. Note that TOCTTOU attacks here are infeasible as the regions have been protected before checking. In addition, she may change the base address of the page table during the process of multi-task applications, while such an attack is detected by comparing the value of corresponding registers between the secure introspection of adjacent tasks. For peripheral attacks, StrongBox configures the TZASC to deny illegal access to the GPU page table region from other peripherals except for the GPU.

B. Attack With Malicious Tasks

We consider the attacker who attempts to execute an arbitrary number of malicious tasks with malicious code. As shown in Fig. 8-②, she may perform two types of attacks. First, she directly launches a malicious confidential GPU application and uses the malicious secure tasks to subvert the StrongBox runtime and critical configurations (e.g., non-secure Stage-2 translation table) in the Trusted RAM. Second, she crafts malicious GPU tasks and attacks the confidential GPU applications (i.e., access sensitive data, code, and GPU page table inside Secure Task RAM). Thus, we propose corresponding defenses against these attacks: (1) To secure the runtime and configuration, StrongBox tightly restricts the access to the Trusted RAM from the peripherals, including the GPU. (2) As for protecting Secure Task RAM, StrongBox ensures that only secure GPU tasks can access the Secure Task RAM, and disallows the GPU to access the Secure Task RAM after secure tasks are switched out. Note that she may fake a malicious task as a secure task in the current confidential GPU application, while it fails the code HMAC check. Moreover, to tamper with the isolated execution environment, she may submit malicious tasks during the secure computation or hide these tasks before switching to the secure tasks. Thus, StrongBox first deprives access to GPU MMIO from the untrusted OS via the non-secure Stage-2 translation, invalidating any malicious tasks submission from the GPU driver. Furthermore, to preclude hidden malicious tasks, StrongBox requires an additional check on GPU status via the protected GPU MMIO interfaces. When detecting a hidden task, StrongBox safely terminates the GPU application.

C. Attack With Fake GPU Device

The attacker may attempt to impersonate a GPU device to spoof GPU state or submission of secure tasks (shown in Fig. 8-③). Since the fake GPU can be emulated by *secure world* components, it can bypass the access control in TZASC to process GPU data. However, we guarantee that StrongBox always interacts with an authentic GPU. To defend against this type of attack, StrongBox checks the GPU state registers and writes the task submission command by accessing the GPU MMIO registers. Based on available manuals [61], [75], [76], [77], the physical address of embedded GPU MMIO registers is fixed and unmodifiable. Therefore, an attacker can only physically change the MMIO physical address of the SoC peripherals with physical access to the AXI bus.

Nevertheless, we find that the Arm official implementation of the secure monitor [78] performs address translation (called EL3 Stage-1 translation) instead of directly accessing the physical address of GPU MMIO. Thus, a *secure world* attacker may subvert the EL3 Stage-1 translation table, misleading the secure monitor to access a fake GPU. To defend against this attack, we carefully protect the translation table and the secure monitor firmware in the secure Stage-2 translation. The attacker may attempt to replace the entire EL3 Stage-1 translation table with a malicious one. However, the attacker cannot modify the table base register (i.e., `TTBR0_EL3`) that is only accessible to the secure monitor, thus the malicious table would never be used in place of a legitimate one.

D. Attack With Compromised GPU Software

Fig. 8-④ shows that the attacker may manipulate the untrusted GPU software stacks (i.e., GPU driver and GPU runtime) to launch an Iago-style attack [74], which can be achieved in three possible ways: (1) manipulating the return values of memory allocation to the unprotected regions, (2) providing the incorrect values of GPU registers to tamper with the critical GPU configurations, and (3) providing incorrect order to execute the secure tasks or simply dropping/replying result. For memory-based Iago attacks, StrongBox verifies the validity of the allocated memory. We ensure that the allocated memory for secure GPU buffers is inside the Secure Task RAM and does not overlap with other GPU buffers. As for GPU register configurations, StrongBox protects the GPU MMIO registers and checks the critical GPU register states. Furthermore, we verify both the task code contents and the task index to guarantee both the code integrity and execution order. Besides, we provide the signature of output GPU buffers and the number of executed secure tasks. In this way, we can detect changes to execution order or the result dropping/replying.

E. Attack From Secure OS and Applications

The attacker who controls the secure OS and applications can reproduce and strengthen most of the attacks above from *secure world*. One exception is the Iago-style attack since StrongBox is not designed to rely on a secure OS for functionality and security. Therefore, we detail them as follows and analyze the security of our defense mechanism.

Secure Task RAM and Trusted RAM: As shown in Fig. 8-⑤, the attacker may tamper with the sensitive data and code of the confidential GPU applications from the compromised secure OS or secure application. In addition, she may modify the GPU page table to map the sensitive data into an unprotected *secure world* memory. To achieve this, she can create the unauthorized

mapping of the Secure Task RAM by modifying the translation table of the secure OS. Although such read or write operations are allowed in TZASC, they are prohibited when performing the secure Stage-2 translation. Alternatively, she may assist in bypassing the access control of one world from another world. Specifically, she may tamper with the Stage-2 translation table of another world to validate its mapping to the physical address of the protected region (e.g., Secure Task RAM). To defend against the attacks, we cancel the mapping of the secure/non-secure Stage-2 translation table in both translation tables. Moreover, she cannot modify the corresponding registers to subvert the non-secure/secure Stage-2 translation (e.g., the translation table base register or control register) since she lacks the hypervisor-layer privilege. As for TZASC protection, we also disable access from secure OS/applications in secure Stage-2 translation.

In addition to access control, she can threaten the cryptographic and integrity verification process (i.e., attack on Trusted RAM). Specifically, she may access the keys, signatures, and temporary data generated in the cryptographic and integrity verification process. With these data, she may easily decrypt the ciphertext data and tamper with the integrity. To bypass this process, she can compromise StrongBox runtime (e.g., injecting malicious codes into the Task Protector module). Therefore, to guarantee its security, we prohibit the mapping of the secure monitor (including StrongBox runtime, Trust modules, the memory of shared keys, etc.) in our secure Stage-2 translation.

Malicious Tasks: Besides directly accessing the confidential GPU application and its page table, Fig. 8-⑤ also shows that she may submit the malicious GPU tasks or modify the GPU configurations to hide the running tasks. However, since the secure Stage-2 deprives access to GPU MMIO, she still fails to hide and execute the malicious tasks. As for faking the malicious tasks as secure tasks to be executed, this attack is challenging to bypass the HMAC check due to the lack of essential signatures.

F. Attack From Abusing Secure Tasks

Fig. 8-⑥ shows that the attacker may abuse secure tasks to access sensitive data from the other confidential GPU applications. Since the attacker creates an entire confidential GPU application consisting of one or more attack tasks, these tasks do not fail the HMAC check and are successfully submitted to the GPU hardware. However, since StrongBox forbids running any other secure tasks before processing the attack tasks, the attacker cannot leak the encrypted data from the other secure tasks. Moreover, Fig. 8-⑥ also shows that the attacker may compromise the StrongBox runtime via the attack tasks (e.g., mapping the Trusted RAM in the GPU page table and accessing it via GPU). However, such an attack fails the TZASC check and thus cannot leverage the GPU to access the Trusted RAM. Note that the attack may abuse the exclusive access of the GPU to block the submission of other GPU tasks, which would result in a Denial-of-Service that is beyond the scope of this article. Nevertheless, the user can require StrongBox to perform the secure termination process, terminating the attack tasks and execute other secure tasks.

VIII. Discussion

Hypervisor-enabled Arm Devices: StrongBox is not suitable for Arm cloud platforms. The primary reason is that cloud GPUs generally own dedicated memory and are connected through PCIe. Moreover, our current prototype cannot directly work in Arm endpoints with a non-secure or secure hypervisor. Although StrongBox does not block the functionality of the hypervisor, it requires non-trivial restriction (e.g., secure the hypervisor firmware and remove the code to access Stage-2 registers) on the untrusted hypervisor to guarantee StrongBox security, introducing a large TCB. However, in future Armv9 endpoints, StrongBox can leverage the new feature, called Granule Protection Table (GPT) [79], to prevent the secure GPU computation from untrusted accessing. By configuring the GPT entries for Secure Task RAM, StrongBox preserves dynamic and fine-grained memory protection without needing Stage-2 translation. Meanwhile, the access control of GPU MMIO can be achieved with similar configurations on GPT entries.

Temporary Exclusivity of GPU: StrongBox requires a temporary exclusivity of the GPU for secure task computation, while it only causes minimal influence on system performance due to four reasons. First, recall from Section II-C that the current Arm endpoints GPU and related SDKs have yet to support concurrent task execution. Thus, parallel executing secure GPU tasks belonging to the same application is natively unsupported. Second, the secure tasks in practical are lightweight, and hence the exclusivity of GPU is transient. For instance, face recognition on mobile devices typically takes less than one second [80]. Third, it is possible to mitigate the impact on GPU rendering by temporarily switching to software-based renderers [81]. Lastly, for the secure GPU applications with dependency on normal GPU tasks, StrongBox allows to process these normal GPU tasks during the switch of secure tasks since the access to the plaintext data is carefully controlled.

Mitigating Performance Overhead: StrongBox achieves a reasonably low performance overhead with compatibility. However, such overhead can be reduced on specific devices. For the cryptographic and integrity-check operations, we can accelerate them with the equipped hardware. Another choice is to build a trusted channel between the user and StrongBox runtime. User leverages a trusted camera to directly transfer the plaintext biometric information into protected GPU buffers without additional encryption.

IX. Related Work

GPU TEEs and Secure Computation: Studies have exploited the isolation features of GPUs for secure computation. Graviton [24] and HETEE [25] leverage the extensive hardware modification (i.e., modifying GPU chip and adding a FPGA for secure purpose) to implement the trusted computation, while HIX [23] extends SGX-based support on GPU enclaves. Such modification severely reduce the compatibility and may not adapt to endpoints. LITE [82] proposes a co-design framework between CPU TEE and its GPU TEE though it is not adapted to endpoint GPUs. Existing Arm TEEs (e.g., SANCTUARY [72], TrustICE [83], Inktag [84], Trustshadow [85] and vTZ [73])

leverage the non-secure [73] and secure [40] Stage-2 translation to achieve access control, or protect the untrusted applications with traditional TrustZone techniques [84], [85]. However, they are yet to consider the confidential computation on endpoint GPU. Cronus [41] leverages the secure Stage-2 translation to guarantee the server-side GPU computation but it is not adapted on endpoint GPUs. In contrast, StrongBox achieves dynamic and complex memory protection on endpoint GPU computation, and prevents the malicious access from the secure OS and secure applications by leveraging this translation.

Defense against Secure World: Most works [72], [73], [83], [86] defend against the threat from the non-secure components but consider Secure OS and secure applications as trusted. Recent studies [40], [53], [87], [88] start to confine and isolate the integrated and potentially vulnerable Secure OS. TEEv [88] implements a minimal controller on the same privilege of Secure OS (i.e., S-EL1), while PrOS [87] monitors them in EL3. They require a non-trivial modification on TEE firmware to remove the mapping of critical regions, while StrongBox confines them via the secure Stage-2 translation without severe modifications on Secure OS or applications. Our defense mechanism against the secure OS is similar to Twinvisor [40] and Hafnium [89] but introduce a smaller TCB (i.e., no hypervisor). Rezone [53] partitions the integrated TEE with two additional peripherals: Platform Partition Controller (PPC) and Auxiliary Control Unit (ACU), while StrongBox is compatible to Arm endpoints with S-EL2 extension.

X. Conclusion

In this article, we propose a novel GPU TEE on Arm-based devices called StrongBox. Our approach provides three key outcomes: Ensuring data confidentiality, protecting task integrity, and providing an isolated computing environment. To fulfill our goals, we leverage TrustZone and non-secure Stage-2 translation to flexibly manage access to GPU task RAM and GPU MMIO interfaces. Moreover, the core components of StrongBox are protected from malicious access from both the untrusted kernel and other peripherals. In addition, we further reduce the TCB of *secure world* OS and applications by leveraging the novel secure virtualization feature. Our design requires no modification to the Arm architecture or any hardware components, providing a higher degree of compatibility than previous GPU TEEs. We also address the threats from the compromised secure OS and secure applications. To better understand StrongBox, we measure the performance of a prototype implemented on an off-the-shelf development board, and analyze the security guarantees provided by StrongBox across a wide range of attack scenarios. Our evaluation shows that StrongBox successfully defends against potential attacks while introducing a low (4.70%–15.26%) overhead across several indicative benchmarks.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments.

References

[1] ARM, "Game engine guides," 2022. [Online]. Available: https://developer.arm.com/solutions/graphics-and-gaming/developer-guides/game-engine-guides
[2] ARM, "Mali texture compression tool," 2022. [Online]. Available: https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-texture-compression-tool
[3] ARM, "VR best practice," 2022. [Online]. Available: https://developer.arm.com/solutions/graphics-and-gaming/developer-guides/vr-best-practice
[4] S. S. Latifi Oskouei, H. Golestani, M. Hashemi, and S. Ghiasi, "CNNdroid: GPU-accelerated execution of trained deep convolutional neural networks on android," in *Proc. 24th ACM Int. Conf. Multimedia*, 2016, pp. 1201–1205.
[5] Q. Cao, N. Balasubramanian, and A. Balasubramanian, "MobiRNN: Efficient recurrent neural network execution on mobile GPU," in *Proc. 1st Int. Workshop Deep Learn. Mobile Syst. Appl.*, 2017, pp. 1–6.
[6] C. Holmes, D. Mawhirter, Y. He, F. Yan, and B. Wu, "GRNN: Low-latency and scalable RNN inference on GPUs," in *Proc. 14th EuroSys Conf.*, 2019, pp. 1–16.
[7] NVIDIA, "NVIDIA DATA CENTER GPUs," 2022. [Online]. Available: https://www.nvidia.com/en-us/data-center/data-center-gpus/
[8] Google, "GPUs on compute engine," 2022. [Online]. Available: https://cloud.google.com/compute/docs/gpus/
[9] Apple, "Discover metal enhancements for A14 bionic," 2022. [Online]. Available: https://developer.apple.com/videos/play/tech-talks/10858/
[10] Qualcomm, "Adreno graphics processing units," 2022. [Online] Available: https://developer.qualcomm.com/software/adreno-gpu-sdk/gpu/
[11] V. Bazarevsky, Y. Kartynnik, A. Vakunov, K. Raveendran, and M. Grundmann, "BlazeFace:Sub-millisecond neural face detection on mobile GPUs," *IEEE Trans. Inf. Forensics Secur.*, vol. 9, no. 1, pp. 62–71, 2013.
[12] ASUS IoT, "ASUS IoT face recognition edge AI dev kit," 2022. [Online]. Available: https://iot.asus.com/solutions/facerecognition/
[13] STMicroelectronics, "Artificial intelligence (AI) face recognition function pack for STM32Cube," 2022. [Online]. Available: https://www.st.com/en/embedded-software/fp-ai-facerec.html
[14] S. Tan, B. Knott, Y. Tian, and D. J. Wu, "CRYPTGPU: Fast privacy-preserving machine learning on the GPU," in *Proc. IEEE Symp. Secur. Privacy*, 2021, pp. 1021–1038.
[15] C.-C. Chang, W.-K. Lee, Y. Liu, B.-M. Goi, and R. C.-W. Phan, "Signature gateway: Offloading signature generation to IoT gateway accelerated by GPU," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4448–4461, Jun. 2019.
[16] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 1310–1321.
[17] T. Hunt et al., "Telekine: Secure computing with cloud GPUs," in *Proc. 17th USENIX Symp. Networked Syst. Des. Implementation*, 2020, pp. 817–833.
[18] S. Truex et al., "A hybrid approach to privacy-preserving federated learning," in *Proc. 12th ACM Workshop Artif. Intell. Secur.*, 2019, pp. 1–11.
[19] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, 2015, pp. 57–64.
[20] ARM, "ARM security technology building a secure system using TrustZone technology," 2009. [Online]. Available: https://developer.arm.com/documentation/PRD29-GENC-009492/latest/
[21] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.
[22] AMD, "AMD secure encrypted virtualization (SEV)," 2022. [Online]. Available: https://developer.amd.com/sev/
[23] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity GPUs," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 455–468.
[24] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on GPUs," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 681–696.
[25] J. Zhu et al., "Enabling rack-scale confidential computing using heterogeneous trusted execution environment," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1450–1465.
[26] CVE, "CVE-2022-21815," 2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-21815
[27] CVE, "CVE-2021-1121," 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-1121
[28] CVE, "CVE-2021-1093," 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-1093

[29] CVE, "CVE-2022-21821," 2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-21821

[30] CVE, "CVE-2020-5991," 2020. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-5991

[31] 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 *Proc. Int. Conf. Internet-of-Things Des. Implementation*, 2021, pp. 67–79.

[32] NVIDIA, "NVIDIA confidential computing," 2022. [Online]. Available: https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/

[33] S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.

[34] H. Lee, H. Kim, C. Kim, H. Han, and E. Seo, "Idempotence-based preemptive GPU kernel scheduling for embedded systems," *IEEE Trans. Comput.*, vol. 70, no. 3, pp. 332–346, Mar. 2021.

[35] R. Baghdadi et al., "PENCIL: A platform-neutral compute intermediate language for accelerator programming," in *Proc. IEEE Int. Conf. Parallel Architecture Compilation*, 2015, pp. 138–149.

[36] H. Lee, J. Roh, and E. Seo, "A GPU kernel transactionization scheme for preemptive priority scheduling," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2018, pp. 202–213.

[37] ARM, "OP-TEE Test," [Online]. Available: https://github.com/OP-TEE/optee_test

[38] Y. Deng et al., "StrongBox: A GPU TEE on arm endpoints," in *Proc. 29th ACM SIGSAC Conf. Comput. Commun. Secur.*, 2022, pp. 769–783.

[39] ARM, "Fixed virtual platforms," [Online]. Available: https://www.arm.com/en/products/development-tools/simulation/fixed-virtual-platforms

[40] D. Li, Z. Mi, Y. Xia, B. Zang, H. Chen, and H. Guan, "TwinVisor: Hardware-isolated confidential virtual machines for ARM," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, 2021, pp. 638–654.

[41] J. Jiang et al., "CRONUS: Fault-isolated, secure and high-performance heterogeneous computing for trusted execution environment," in *Proc. IEEE/ACM 55th Int. Symp. Microarchitecture*, 2022, pp. 124–143.

[42] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, "PACStack: An authenticated call stack," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 357–374.

[43] 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 *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 177–194.

[44] ARM, "Trustzone for cortex-A," 2022. [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-a

[45] ARM, "ARM generic interrupt controller architecture specification version 2.0," 2013. [Online]. Available: https://developer.arm.com/documentation/ihi0048/latest

[46] ARM, "GICv3 and GICv4 software overview," 2023. [Online]. Available: https://developer.arm.com/documentation/dai0492/latest

[47] ARM, "Cortex-A7 MPCore technical reference manual," 2022. [Online]. Available: https://developer.arm.com/documentation/ddi0464/latest/

[48] ARM, "Arm Cortex-A53 MPCore processor technical reference manual," 2022. [Online] Available: https://developer.arm.com/documentation/ddi0500/latest/

[49] ARM, "Arm Cortex-A57 MPCore processor technical reference manual," 2022. [Online] Available: https://developer.arm.com/documentation/ddi0488/latest/

[50] ARM, "Arm Cortex-A72 MPCore processor technical reference manual," 2022. [Online] Available: https://developer.arm.com/documentation/100095/latest/

[51] ARM, "Mali GPU user-space binary drivers," 2023. [Online] Available: https://developer.arm.com/downloads/-/mali-drivers/user-space

[52] ARM, "Arm architecture reference manual Armv8, for Armv8-A architecture profile," 2022. [Online] Available: https://developer.arm.com/documentation/ddi0487/latest/

[53] D. Cerdeira, J. Martins, N. Santos, and S. Pinto, "REZONE: Disarming TrustZone with TEE privilege reduction," in *Proc. 31st USENIX Secur. Symp*., 2022, pp. 2261–2279.

[54] J. Lee, Y. Liu, and Y. Lee, "ParallelFusion: Towards maximum utilization of mobile GPU for DNN inference," in *Proc. 5th Int. Workshop Embedded Mobile Deep Learn.*, 2021, pp. 25–30.

[55] J. S. Jeong et al., "Band: Coordinated Multi-DNN inference on heterogeneous mobile processors," in *Proc. 20th Annu. Int. Conf. Mobile Syst., Appl. Serv.*, 2022, pp. 235–247.

[56] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976.

[57] ARM, "OP-TEE Trusted OS," 2023. [Online]. Available: https://github.com/OP-TEE/optee_os

[58] CVE, "CVE-2019-1010295," 2021. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-1010295

[59] CVE "CVE-2019-1010296," 2021. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-1010296

[60] CVE, "CVE-2021-44149," 2021. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44149

[61] ARM, "Juno r2 ARM development platform SoC." [Online]. Available: https://developer.arm.com/documentation/ddi0515/latest

[62] ARM, "Open source Mali midgard GPU kernel drivers," 2022. [Online]. Available: https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-drivers/midgard-kernel

[63] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "SeCReT: Secure channel between rich execution environment and trusted execution environment," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[64] H. Sun, K. Sun, Y. Wang, and J. Jing, "TrustOTP: Transforming smartphones into secure one-time password tokens," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 976–988.

[65] I. Eklektix., "A deep dive into CMA," 2022. [Online]. Available: https://lwn.net/Articles/486301/

[66] ARM, "Midgard architecture," 2017. [Online]. Available: https://gitlab.freedesktop.org/panfrost/Mali-isa-docs/-/blob/master/Midgard.md

[67] ARM, "Mali-G78 GPUs valhall instruction set documentation released after reverse-engineering work," 2021. [Online]. Available: https://www.cnx-software.com/2021/07/23/mali-g78-gpu-valhall-instruction-set-documentation-reverse-engineering/

[68] AlDanial, "cloc," 2021. [Online]. Available: https://github.com/AlDanial/cloc

[69] BYTE Magazine, "Linux/unix nbench," 2022. [Online]. Available: http://www.tux.org/mayer/linux/bmark.html

[70] X. Wang, S. Yeoh, R. Lyerly, P. Olivier, S.-H. Kim, and B. Ravindran, "A framework for software diversification with ISA heterogeneity," in *Proc. 23rd Int. Symp. Res. Attacks, Intrusions Defenses*, 2020, pp. 427–442.

[71] K. Hohentanner, P. Zieris, and J. Horsch, "CryptSan: Leveraging ARM pointer authentication for memory safety in C/C++," in *Proc. 38th ACM/SIGAPP Symp. Appl. Comput.*, 2023, pp. 1530–1539.

[72] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with User-space Enclaves," in *Proc. 26th Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.

[73] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM TrustZone," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 541–556.

[74] S. Checkoway and H. Shacham, "Iago Attacks: Why the system call API is a bad untrusted RPC interface ," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 253–264, 2013.

[75] FuZhou Rockchip Electronics Co.,Ltd., "Rockchip RK3288 technical reference manual Part1," 2017. [Online]. Available: http://opensource.rockchips.com/images/8/8f/Rockchip_RK3288_TRM_V1.2_Part1-2017 0321.pdf

[76] I. Amlogic., "S905 datasheet," 2016. [Online]. Available: https://dn.odroid.com/S905/DataSheet/S905_Public_Datasheet_V1.1.4.pdf

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

[78] ARM, "Arm-trusted-Firmware," [Online]. Available: https://github.com/ARM-software/arm-trusted-firmware

[79] ARM, "Granule protection tables in TF-A," 2021. [Online]. Available: https://www.trustedfirmware.org/docs/tfa_tech_forum_2021_10_21_gpt.pdf

[80] A. Rattani and R. Derakhshani, "A survey of mobile face biometrics," *Comput. Elect. Eng.*, vol. 72, pp. 39–52, 2018.

[81] 3D Mesa, "The mesa 3D graphics library," 2022. [Online] Available: https://www.mesa3d.org/

[82] A. W. B. Yudha, J. Meyer, S. Yuan, H. Zhou, and Y. Solihin, "LITE: A Low-cost practical inter-operable GPU TEE," in *Proc. 36th ACM Int. Conf. Supercomputing*, 2022, pp. 1–13.

[83] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "TrustICE: Hardware-assisted isolated computing environments on mobile devices," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2015, pp. 367–378.

[84] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "InkTag: Secure applications on an untrusted operating system," in *Proc. 18th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2013, pp. 265–278.

[85] L. Guan et al., "TrustShadow: Secure execution of unmodified applications with arm trustzone," in *Proc. 15th Annu. Int. Conf. Mobile Syst., Appl., Serv.*, 2017, pp. 488–501.
[86] J. Jang et al., "PrivateZone: Providing a private execution environment using ARM TrustZone," *IEEE Trans. Dependable Secure Comput.*, vol. 15, no. 5, pp. 797–810, Sep./Oct. 2018.
[87] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek, "PrOS: Light-weight privatized secure OSes in ARM TrustZone," *IEEE Trans. Mobile Comput.*, vol. 19, no. 6, pp. 1434–1447, Jun. 2020.
[88] W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang, "TEEv: Virtualizing trusted execution environments on mobile platforms," in *Proc. 15th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2019, pp. 2–16.
[89] ARM, "Hafnium," 2023. [Online]. Available: https://www.trustedfirmware.org/projects/hafnium/

Chenxu Wang received the bachelor's degree in computer science and engineering from the Southern University of Science and Technology. He is currently working toward the Joint PhD degree with the Southern University of Science and Technology and The Hong Kong Polytechnic University. His research interests include virtualization and trusted execution environment on arm architecture.

Yunjie Deng received the bachelor's degree in computer science and engineering from the Southern University of Science and Technology. He is currently working toward the master's degree with the Southern University of Science and Technology. His research interests include trusted execution environment and GPU computing.

Zhenyu Ning received the PhD degree in computer science from Wayne State University in 2020. He is currently an associate professor with Hunan University. His research interests include security and privacy, system security, mobile security, IoT security, trusted execution environment, and hardware-assisted security.

Kevin Leach received the PhD degree in computer engineering from the University of Virginia. He is currently an assistant professor of computer science with Vanderbilt University. His research interests include systems security, specifically the debugging transparency problem, conversational artificial intelligence, program analysis, medical informatics, and Big Data applications.

Jin Li (Senior Member, IEEE) received the BS degree in mathematics from Southwest University in 2002 and the MS degree in mathematics and the PhD degree in information security from Sun Yat-sen University, in 2004 and 2007, respectively. He is currently a professor and the executive dean of the Institute of Artificial Intelligence and Blockchain, Guangzhou University. He has authored or coauthored more than 100 papers in international conferences and journals, including IEEE INFOCOM, *IEEE Transactions on Information Forensics and Security*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, and ESORICS. His works are cited more than 18000 times at Google Scholar and the H-Index is 50. His research interests include security in artificial intelligence and applied cryptography. He is also the program chairs and committee member for many international conferences. He was the recipient of the NSFC Outstanding Youth Foundation in 2017.

Shoumeng Yan received the PhD degree from Northwestern Polytechnic University. He is currently a principal engineer with Ant Group and the senior director of secure and trustworthy computing. He has authored or coauthored many papers in international conferences, including USENIX Security, USENIX ATC, ACM CCS, and ACM ASPLOS. His research interests include OS, TEE, and domain specific accelerators.

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

Jiannong Cao (Fellow, IEEE) is currently an Otto Poon Charitable Foundation professor of data science and the chair professor of distributed and mobile computing with the Department of Computing, 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.

Fengwei Zhang (Senior Member, IEEE) is currently an associate professor with the Department of Computer Science and Engineering, Southern University of Science and Technology (SUSTech). His primary research interests include 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 with the Department of Computer Science, Wayne State University.