# StrongBox: A GPU TEE on Arm Endpoints

Yunjie Deng*
Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
Department of Computer Science and Engineering, Southern University of Science and Technology, China

Chenxu Wang*
Department of Computer Science and Engineering, Southern University of Science and Technology, China
Department of Computing, The Hong Kong Polytechnic University, China

Shunchang Yu
Shiqing Liu
Department of Computer Science and Engineering, Southern University of Science and Technology, China

Zhenyu Ning
Hunan University, China
Department of Computer Science and Engineering, Southern University of Science and Technology, China

Kevin Leach
Institute for Software Integrated Systems, Vanderbilt University, USA

Jin Li
School of Computer Science, Guangzhou University, China

Shoumeng Yan
Zhengyu He
Ant Group, China

Jiannong Cao
Department of Computing, The Hong Kong Polytechnic University, China

Fengwei Zhang†
Department of Computer Science and Engineering, Southern University of Science and Technology, China
Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China

## ABSTRACT

A wide range of Arm endpoints leverage integrated and discrete GPUs to accelerate computation such as image processing and numerical processing applications. However, in spite of these important use cases, Arm GPU security has yet to be scrutinized by the community. By exploiting vulnerabilities in the kernel, attackers can directly access sensitive data used during GPU computing, such as personally-identifiable image data in computer vision tasks. Existing work has used Trusted Execution Environments (TEEs) to address GPU security concerns on Intel-based platforms, while there are numerous architectural differences that lead to novel technical challenges in deploying TEEs for Arm GPUs. In addition, extant Arm-based GPU defenses are intended for secure machine learning, and lack generality. 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. During confidential computation on Arm GPUs, StrongBox provides an isolated execution environment by ensuring exclusive access to the 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, a stark contrast with Intel-based platforms. Furthermore, by characterizing GPU buffers as secure and non-secure, StrongBox reduces redundant security introspection operations to control access to sensitive data used by the GPU, ultimately reducing runtime overhead. 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. Our results show that StrongBox successfully ensures the GPU computing security with a low (4.70% - 15.26%) overhead across several indicative benchmarks.

## CCS CONCEPTS

• **Security and privacy** → **Trusted computing**; *Mobile platform security*.

## KEYWORDS

Arm endpoint GPU; Trusted execution

**ACM Reference Format:**
Yunjie Deng*, Chenxu Wang*, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, and Fengwei Zhang†. 2022. StrongBox: A GPU TEE on Arm Endpoints. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560627

## 1 INTRODUCTION

GPUs are now widely used in general- and high-performance applications such as 3D games [17], video processing and compression [18], mobile Virtual Reality [22], and neural network training and inference [28, 45, 57]. In addition, GPUs are used not only in server and cloud environments [42, 74], but also in small embedded systems [4, 77] such as smartphones and autonomous vehicles to satisfy the sharply-increasing performance demands.

* Yunjie Deng and Chenxu Wang are co-first authors.
† Fengwei Zhang is the corresponding author.


*CCS '22, November 7–11, 2022, Los Angeles, CA, USA.*

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560627

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 the sensitive data processed on victim GPU applications. Combined with the increase in the use of personally identifiable information [23, 25, 82] and sensitive secrets computed with GPUs [29, 86], 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 [48, 79, 81, 89]. Recently, one such technology is Trusted Execution Environments (TEEs) [2, 5, 33, 79]. 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 [51] using modified Intel Software Guard eXtensions (SGX) [33], Graviton [90] and HETEE [93] with customized TEE. 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 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 [51, 90] 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 the 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 [34–38], which severely threatens data security during the computation. Finally, the GPU TEE mechanisms [51, 90, 93] on Intel-based devices entail heavy hardware modification, which, if adopted to Arm devices, would result in poor compatibility. Existing secure computation on Arm endpoint GPUs requires porting the entire GPU driver into the TrustZone and only focuses on specific applications (e.g., deep learning inference [64]). These defects have yet to be properly addressed. As for the defense of GPU chips, NVIDIA recently proposes the H100 GPU [73] to establish a trusted execution environment on GPUs, but this has yet to be demonstrably 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.

To satisfy these requirements, the primary technical challenge is to ensure the exclusivity of secure GPU execution while using a unified memory. We must provide an isolated runtime environment in which secure GPU tasks can engage the GPU even when a compromised OS would otherwise allow an attacker free reign over peripheral devices, drivers, and memory. Following the primary challenge, we must address access control to sensitive data and code in confidential applications. The data and code are scattered throughout the memory, and their access permission must frequently change to interface with the delegated components. Ultimately, the isolated runtime environment must provide acceptable robustness with a low performance overhead. To that end, we present the following:

- Rather than requiring additional hardware or porting the GPU Driver to TEE, StrongBox implements a new access control mechanism based on generic Arm features (i.e., TrustZone and Stage-2 translation) to restrict unauthorized access to GPU MMIO registers and unified GPU memory. The implementation of StrongBox introduces a small TCB, which is described in Section 6.
- StrongBox defines a dynamic access control policy for sensitive data and critical code with fine granularity. Without affecting the stability of the native system, we prevent malicious operations against GPU task code and data. Detailed analysis of performance and security implications are discussed in Sections 6 and 7, respectively.
- To ensure low performance overhead, we optimize redundant security checks by differentiating GPU buffers for the multi-task computation. Section 6 shows that performance overhead is mitigated with our strategy.

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 of 1,366 lines of code, which is orders of magnitude smaller than the state-of-the-art approach [64] of porting a 30K LoC Arm Midgard GPU Driver to TEE. We measure the performance of our prototype using a popular GPU benchmark suite, called Rodinia [30], which has been widely used to evaluate the performance on Arm devices [24, 59, 60]. Next, we examine the robustness of StrongBox through three typical neural network models (LeNet-5 [58], SqueezeNet [49], and MobileNet-v1 [46]), and evaluate the effectiveness of our optimization mechanism. Moreover, we compare StrongBox to the state-of-the-art GPU TEEs in varied aspects, following with the security analysis of StrongBox under the assumed adversary. Our evaluation results indicate that StrongBox

successfully achieves its security guarantees while introducing a reasonably low (4.70% - 15.26%) performance overhead.

We claim the following contributions in this work:

- We present StrongBox, the first unified-memory 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 implement a prototype of StrongBox on an Arm development board without any hardware or architecture modification. We share the source code of StrongBox[1].
- We perform a comprehensive evaluation of StrongBox, and we present a detailed security analysis of our prototype. Our results show that StrongBox effectively protects the sensitive data with a comparable performance overhead.

# 2 BACKGROUND

## 2.1 Arm TrustZone

Arm TrustZone [21] 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 trusted applications or trusted OS, and (2) *normal world*, which is used for untrusted applications or traditional OS. 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. Thus, the TZASC provides access control to the memory accessed by the CPU and each peripheral device. 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 2.2 to address this limitation.

TrustZone also isolates secure and non-secure interrupts in response to device I/O. Specifically, TrustZone uses a Generic Interrupt Controller (GIC) [6] to create two groups of interrupts, Group 0 (accessible only in *secure world*) and Group 1 (accessible in both secure and normal worlds). TrustZone identifies the interrupt and its group when they occur, in turn dispatching the interrupt to the CPU with the related security state. TrustZone protects such configurations from the malicious Non-secure components (e.g., applications, OS, and hypervisor). In off-the-shelf Arm devices, interrupts related to the GPU are initially categorized into Group 1 and handled by a non-secure GPU Driver. In this paper, we control the switching of GPU interrupt state to efficiently process sensitive data and restore the environment.

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

## 2.2 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 [12–14, 16] 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.

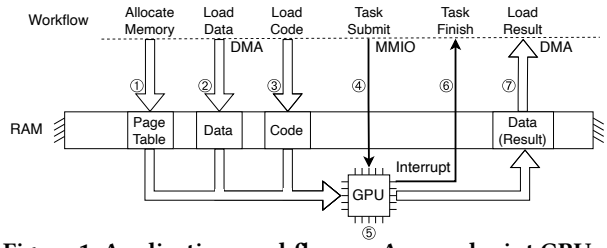


**Figure 1: Application workflow on Arm endpoint GPUs.**

## 2.3 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 [20]), 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. Figure 1 shows the typical execution of a GPU application on an Arm endpoint GPU: 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 (①). Next, data are loaded into the allocated GPU buffers through a Direct Memory Access (DMA) controller (②). Then, the GPU software stack loads the binary code into GPU memory (③). After that, the GPU task start command is sent by configuring the GPU MMIO registers (④). After receiving the submission command, the GPU computes the task based on the code and data, and stores the execution result in specific memory (⑤). 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 (③–⑥). In contrast, data are typically loaded only once, following GPU programming conventions [9, 76, 87]. Note that most Arm endpoint GPUs equip multiple shader cores, simultaneously processing multiple threads belonging to the same GPU task [15, 77]. However, the GPU tasks are executed sequentially on Arm endpoint GPUs. Unlike server GPU software stacks such as NVIDIA CUDA [72], which can concurrently execute tasks without data dependency, studies show that mainstream Arm endpoint GPUs [61] and related

SDKs [53] have yet to support concurrent execution. After processing all tasks, results are directly accessed or exported through DMA (⑦). StrongBox secures the task execution (④ and ⑥) and builds a secure data path (② and ⑦) for data transfer.

# 3 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 the malicious tasks to access the memory of the victim task. Following existing best practices for SGX-based GPU TEEs [51], 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 trust *secure world* and do not consider threats against the trusted kernel or applications. Moreover, we consider cryptographic-based attacks, physical attacks, and side-channel/spy attacks to be beyond the scope of this paper. As with existing GPU TEEs [51, 90, 93], we do not address the Denial-of-Service attacks against long-running applications, though other TrustZone-based TEEs [91] are a potential solution to this attack.

# 4 DESIGN

StrongBox allows users to use confidential GPU applications inside an untrusted system. Recall that these applications include tasks such as face recognition [50], fingerprint recognition [56], and neural network inference [46, 80], all of which entail some degree of potentially-sensitive data. We envision scenarios in which users execute the deployed confidential GPU applications to establish cryptographic keys with StrongBox using a key-management protocol. To defend against data leakage during data transfer, users send the encrypted data to StrongBox. Then StrongBox protects and decrypts the data, allowing the GPU to process them securely. Lastly, users retrieve encrypted results from StrongBox.

Before we elaborate on the design of StrongBox, we discuss three potential alternative design choices. First, we could port the GPU into TrustZone [64]. However, such isolation requires migrating the entire GPU and related software components (e.g., GPU driver and runtime) to *secure world*, which would invariably require a large TCB and expose a large attack surface. Second, we could virtualize the entire GPU. However, such a design requires a hypervisor to support memory virtualization, GPU state virtualization, scheduling, and other critical functions, thus requiring a large TCB and high runtime overhead. Third, we could add extra hardware to support GPU TEEs. Graviton [90] and the recent NVIDIA H100 GPU [73] add extra hardware inside the GPU chip to craft and arrange a confidential environment for GPU. However, such GPU devices involve various architecture differences compared with Arm endpoint GPUs, and such hardware modification would adversely affect compatibility. Based on these issues, we propose StrongBox to secure the GPU computation on Arm endpoints.

## 4.1 Goals

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 4.3). For the latter case, StrongBox manages the access to the unified memory to restrict untrusted access to the task execution environment (see Section 4.4).

**G2: Reducing the size of trusted computing base.** Next, we must maintain a lightweight TEE. Several GPU TEEs and secure computing systems [51, 64, 90] 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 lightweight StrongBox 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 (discussed in Section 4.2).

**G3: Ensuring the 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 [51, 90, 93] 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 — instead, we use features that are widely-available on general Arm devices (discussed in Section 4.2).

## 4.2 StrongBox Overview

Figure 2 illustrates the design of StrongBox, which is divided into software and hardware components. As shown in Figure 2-left, the GPU applications, both non-confidential and confidential, are first processed by the GPU runtime (e.g., OpenCL [20]) and the Host OS. The Host OS transfers essential data through DMA using the GPU driver to handle memory management and task scheduling. Note that the GPU tasks in non-confidential and confidential GPU applications are categorized as non-secure tasks and secure tasks, respectively. The non-secure tasks then proceed to the GPU. However, when a secure task is ready to execute, the GPU driver triggers an `smc` instruction to divert control flow to our StrongBox runtime. The StrongBox runtime is deployed in the secure monitor (EL3) to protect the secure tasks with several

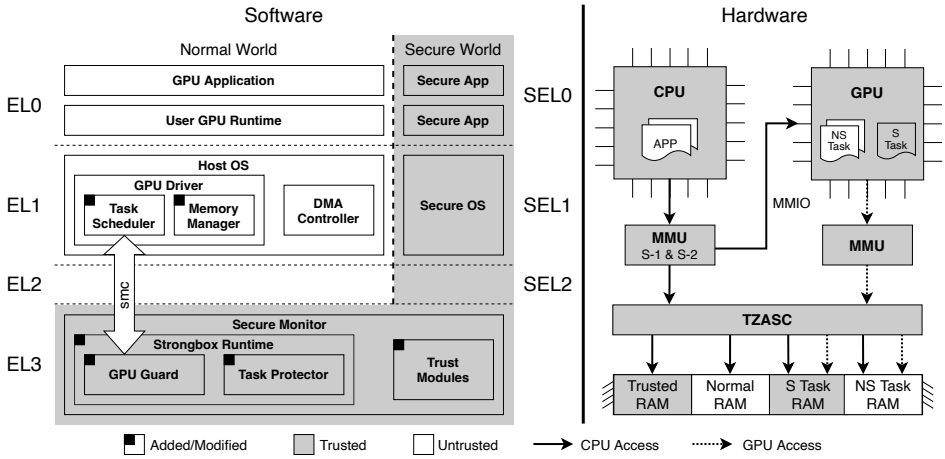


**Figure 2: StrongBox architecture overview**

trusted modules (e.g., secure boot and key management modules). To protect the data security in secure tasks, StrongBox 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. Together, these components help to fulfill our goal of isolated GPU execution and confidential storage. In addition, we reuse the GPU runtime and driver software in the OS (EL1) to reduce the overall TCB size (**G2**). The GPU driver and runtime manage hardware resources and data transfer for both confidential and non-confidential GPU applications (EL0), while the data security is ensured by the thin StrongBox runtime. As a result, our approach ensures that the GPU can execute secure tasks in isolation while executing within a potentially-compromised OS. Note that hypervisors are not deployed on most Arm endpoints, and StrongBox requires no modification to *secure world* (i.e., SEL0 – SEL2).

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 StrongBox runtime and Stage-2 translation table, and (4) Secure Task RAM, which is a fixed, non-secure memory region reserved 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 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.

In the following sections, we describe how StrongBox works with the non-confidential software stack to protect the execution environment, and how the software components behave.

## 4.3 Exclusivity of GPU on 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 3, 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, Stage-2 translation helps lock the system mapping of GPU MMIO registers during computation. By leveraging custom `smc` event handlers and 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, 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, 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 — the GPU driver will not emit the `smc` call to work with our security components.

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 Stage-2 translation to

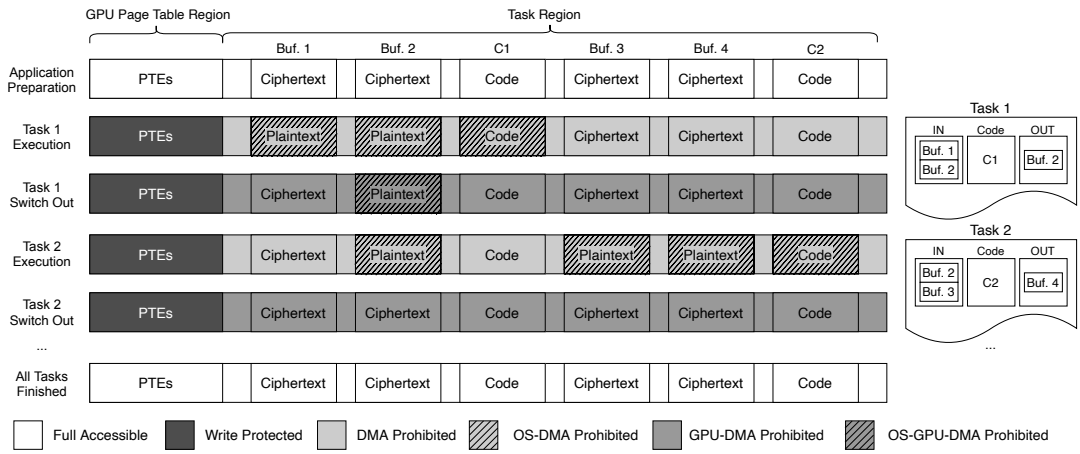


**Figure 3: The changes of access permissions on Secure Task RAM when a confidential GPU application is executed. Task 1 and Task 2 are two example tasks inside the application. Task 1 contains input Buf. 1 and Buf. 2, output Buf. 2, and code segment C1. Task 2 contains input Buf. 2 and Buf. 3, output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

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 Stage-2 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. 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 4.4. 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 [61] and related SDKs [53] have yet to support the concurrent computation of GPU tasks (as mentioned in Section 2.3). Moreover, when processing, StrongBox does not block other `smc` calls that do not interact with the GPU.

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 restores the GPU MMIO and GPU memory access permission. Furthermore, GPU Guard configures the GIC and switches the GPU interrupt back to non-secure state to allow the GPU driver to handle the interrupt. After restoring the MMIO access permissions and interrupt state, the GPU is allowed to process new tasks.

## 4.4 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 [64] 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 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, 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 the access from DMA, GPU, and other peripherals. As for the GPU page table region, StrongBox employs Stage-2 translation to monitor modification requests from the untrusted OS. To avoid potential peripheral attacks, we further leverage TZASC to prohibit write access from peripherals to the GPU page table region. If the content in these regions is incorrectly

allocated, we terminate the application and erase any sensitive data. 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 untrusted peripherals through DMA.
- *OS-DMA Prohibited*: Disallow the read/write operations from both OS and the untrusted peripherals through DMA.
- *GPU-DMA Prohibited*: Disallow the read/write operations from GPU and untrusted peripherals through DMA.
- *OS-GPU-DMA Prohibited*: Disallow the read/write operations from OS, GPU, and untrusted peripherals through DMA.

Figure 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 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 during DMA transfers. 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 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. Specifically, the executed results are encrypted and hashed before reverting to a non-secure state. Thus, plaintext instances of data exist only during secure task execution.

## 4.5 Optimization for Multi-Task Computation

StrongBox also introduces nontrivial overhead on GPU applications. We observe that cryptographic functions and access permission change on GPU buffers incur the most overhead, which can be aggregated in typical GPU applications over multiple tasks. In various multi-task scenarios (e.g., image processing [43, 69, 75], machine learning inference [88], signal processing [41], and cryptography acceleration [65]), data from one task may feed to the next. By default, Task Protector unnecessarily encrypts all data buffers from one task, even though a subsequent secure task will immediately consume the data. Thus, we can optimize such steps by leveraging the input/output relations of GPU buffers.

StrongBox tracks buffer usage of GPU applications and supports a flexible cryptographic policy to handle different types of GPU buffers. More specifically, we detail the policy in Table 1. Task Protector applies one of three policies to data upon the first usage of a GPU buffer. For data requiring confidentiality, we use the encryption policy (`F1`), while the policy `F2` can be applied to the GPU buffers only requiring an integrity check. For the GPU buffers without meaningful data before GPU computation, the policy `F3` can be selected. After the last usage of buffers, StrongBox only needs to encrypt and expose (`L1`) the output buffer as result, while erasing (`L2`) the data in input data and temporary data on GPU buffers. Note that plaintext buffers are maintained with proper protection during the secure task execution and task switching, which only allows the access from the secure state CPU and the authenticated secure GPU tasks. These policies allow flexible memory management while protecting sensitive data.

**Table 1: StrongBox's operations on varied GPU buffers at two timestamps. The `F1 – F3` and `L1 – L2` mean different policies to process a GPU buffer at two timestamps.**

| Timestamp | Operation on GPU buffers |
|---|---|
| First Time to Process | (F1) Enforce Protection + Decryption<br>(F2) Enforce Protection + Integrity Check<br>(F3) Enforce Protection |
| Last Time to Process | (L1) Encryption + Cancel Protection<br>(L2) Erase + Cancel Protection |

The shown policies in Table 1 fully protect against the incorrect specification cases and the attacker who attempts to leak the sensitive data by subverting the defined policy. StrongBox always protects any GPU buffers before sensitive data within are decrypted, and always encrypts or erases sensitive data within GPU buffers before terminating the protection. Furthermore, we consider an attacker who can terminate the GPU application early and leave plaintext data inside the memory. However, the sensitive data is still under protection and is isolated from the untrusted OS. To further guarantee the data security, we perform a secure termination check before creating the next GPU application. During the check, StrongBox first verifies the entire Stage-2 translation table 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. After checking the Stage-2 translation table, StrongBox restores the

TZASC to finish the secure termination check. Our optimization effectively reduces the performance overhead of our GPU TEE.

### 4.6 Trust Establishment

In this subsection, we discuss the assumptions we make to establish a chain of trust to our software components.

**Secure Boot.** The initialization of StrongBox can be guaranteed by secure boot. By verifying the digital signatures of the loaded images, the boot process ensures the integrity and authenticity of the entire StrongBox system. When the code segment of StrongBox is checked after secure boot, we can safely configure the memory region and interrupts for GPU tasks.

**Remote Attestation and Key Management.** We assume StrongBox is deployed by endpoint device vendors, and its load-time integrity is verified by the secure boot. We assume public key infrastructure, including a public/private key pair and certificate, is installed into *secure world* by the vendor. To exchange a secret key, the developer encrypts an AES key with the certified public key. Then, the StrongBox runtime in *secure world* receives the encrypted AES key and decrypts it with the private key. Next, the AES key is used to encrypt and decrypt the secure GPU code and data. We share the same key for tasks within a confidential GPU application and establish a new AES key for the next one. Note that the AES key only provides the confidentiality, and the authenticity of the passed information is not needed since anyone can pass code/data to StrongBox and use the GPU TEE.

## 5 IMPLEMENTATION

We prototype StrongBox on an Arm Juno R2 development board [7] with 8GB DRAM, an embedded Mali-T624 GPU, and the Arm TrustZone extension. We use Linux v4.14.59 with an open-source Midgard GPU Driver [19] in *normal world*, and run Arm Trusted Firmware (ATF) v2.1 in secure monitor. To create an isolated execution environment, we reserve 264MB as Secure Task RAM, including a 256MB region (`0xB0000000–0xBFFFFFFF`) to hold secure tasks and a 8MB region (`0xAF800000–0xAFFFFFFF`) for GPU page table. The Trusted RAM contains the memory space for ATF and an additional 4MB region (`0xA0000000–0xA03FFFFF`) for the 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 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 *secure world* has established a key management system and a communication channel with the user. These steps can be achieved following previous work [52, 84] and we do not claim this as a contribution of our work. We use Advanced Encryption Standard (AES) encryption with a pre-shared 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.

### 5.1 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 256MB 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 [39] (CMA) and use `cma_alloc` 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.

### 5.2 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 Stage-2 translation table entries to prevent any unauthorized access to GPU MMIO. Specifically, it sets the last bit of the corresponding Stage-2 PTEs as 0 to invalidate the mapping of GPU MMIO regions, then invalidates the TLB entries for each CPU core. 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 [19] 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 [6] 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.

## 5.3 Task Protector

Task Protector leverages both TZASC and 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 Stage-2 mapping with TLB invalidation. Any illegal read or write access to the code and data is prohibited by triggering the Stage-2 translation fault.

As part of implementing access control, Task Protector performs cryptographic and integrity-checking operations for each secure task. Thus, 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 [8, 11] 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.

To reduce the performance overhead, we provide an interface to allow the developer manually specify the optimization policy. Recall from Section 4.5 that the developer provides pairs of time points about the first and last usage of sensitive GPU buffers to decrypt and re-encrypt the sensitive data. Since GPU programs require developers to specify which GPU buffers to be processed on each GPU task, such information can be naturally obtained during application development. Therefore, developers must spend a small amount of manual effort to deduce and implement the policy.

# 6 EVALUATION

In this section, we evaluate our prototype of StrongBox based on our implementation (Section 5). We consider six research questions in our evaluation:

**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 neural network models?
**RQ4**. How effective is our approach to optimizing sequential secure GPU tasks?
**RQ5**. How does StrongBox compare to state-of-the-art GPU TEEs and secure computation systems?

**Table 2: Code size of StrongBox.**

| Component | Function | Lines of Code |
|---|---|---|
| GPU Driver | S-2 Initialization | 389 |
| | GPU Driver | 179 |
| ATF | TZASC Initialization | 8 |
| | Cryptographic Operation | 530 |
| | Integrity Verification | 148 |
| | GPU Access Control | 344 |
| | Other Configuration | 175 |
| Total | | 1,773 |

**Table 3: List of the selected Rodinia benchmark suites.**

| | Problem size | Tasks | Seq. Exe. | Memory | Data (In./Out.) |
|---|---|---|---|---|---|
| KNN | 42,764 points | 1 | ✓ | 0.49 MB | 0.33 MB / 0.16 MB |
| PF | 100,000 × 100 points | 5 | ✓ | 38.59 MB | 38.14 MB / 0.44 MB |
| LUD | 2,048 × 2,048 points | 382 | ✓ | 16.00 MB | 16.00 MB / 16.00 MB |
| H3D | 512 × 512 × 8 points | 500 | ✓ | 24.00 MB | 24.00 MB / 8.00 MB |
| LMD | 25 × 25 × 25 boxes | 1 | ✓ | 63.42 MB | 63.42 MB / 23.84 MB |
| GS | 2,048 × 2,048 points | 4,094 | ✓ | 32.01 MB | 32.01 MB / 32.01 MB |

**RQ6**. How much overhead is incurred on system performance?

## 6.1 RQ1: TCB Size of StrongBox

Table 2 shows the code size of StrongBox reported by *cloc* [1], 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 4. 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 30K LoC) nor the OpenCL driver (32MB). 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 becomes compromised, our security mechanism can still secure the sensitive data computed on the GPU.

## 6.2 RQ2: Evaluation on Rodinia Benchmarks

To demonstrate the runtime performance of StrongBox, we consider the Rodinia benchmark suite [30], which offers realistic workload scenarios to measure the performance of GPU computing.

*6.2.1 Experimental Setup.* In total, we select six applications from the Rodinia suite. Table 3 shows the number of tasks and the amount each task consumes during execution. We consider 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 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 2.3.

We show a breakdown of the performance overhead for our prototype. We group the results as **Base** and **Increment**, which represent the native performance and additional time introduced by StrongBox, respectively. For **Base**, we measure the time elapsed by confidential GPU applications in untrusted components (**Untrusted**) and the computation time on the GPU (**GPU**). In **Increment**, we consider the overhead contributed by three components: **GDriver**, which includes additional resource consumption within the GPU driver. **GGuard**, the time elapsed while executing GPU Guard. **TProtect**, which is the sum of time elapsed in Task Protector. We run each of the six benchmark tasks 30 times, with and without StrongBox enabled, and report the average time.

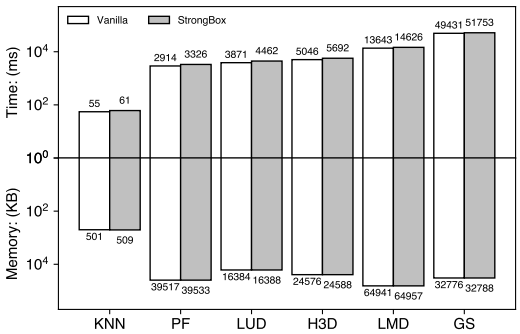


**Figure 4: Runtime performance on six Rodinia benchmarks.**

*6.2.2 Performance Analysis.* Figure 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 5.1, the additional memory consumption from our page-alignment requirement is insubstantial since it is primarily attributable to the number of GPU buffers. Table 4 reports further details for each component. It shows that **GGuard** incurs the smallest time cost in the entire benchmark suite, primarily because GPU Guard requires a small and constant time cost (about 1ms) to process a single task in any application. Moreover, since **GDriver** includes the overhead of scheduling and secure/non-secure switching, it is determined mainly by the number of tasks, which primarily impacts the runtime of the Task Scheduler. Our results reflect this intuition: the GS application with 4,094 tasks introduces 1,561ms overhead, compared to the KNN application with 1 task, which introduces 1ms overhead. In Task Protector, cryptographic and integrity checking operations are proportional to the input and output data size. For instance, **TProtect** component in H3D (total 32MB data) generates approximately half the latency of GS (total 64MB data), despite the fact that GS entails eights of times more tasks than H3D. These results indicate that StrongBox successfully optimizes the redundant slowdown caused by Task Protector in applications that involve multiple secure GPU tasks. We further examine time savings using our optimizations in Section 6.4.

**Table 4: Breakdown (ms) of overhead in StrongBox.**

| | Base | | Increment | | |
|---|---|---|---|---|---|
| | Untrusted | GPU | GDriver | GGuard | TProtect |
| KNN | 54.82 | 0.30 | 1.09 | 0.01 | 4.86 |
| PF | 125.75 | 2788.63 | 12.09 | 0.09 | 399.48 |
| LUD | 507.55 | 3364.18 | 246.22 | 6.51 | 338.10 |
| H3D | 413.13 | 4633.01 | 304.76 | 8.88 | 332.82 |
| LMD | 168.54 | 13474.49 | 6.47 | 0.02 | 977.46 |
| GS | 2140.54 | 47291.35 | 1561.26 | 65.90 | 694.52 |

## 6.3 RQ3: Evaluation on Neural Network Models

To further measure the robustness of StrongBox in typical state-of-the-art GPU applications, we next evaluate the inference overhead on three indicative real-world neural network models of varying complexity: The lightweight LeNet-5 [58], the middle-weight SqueezeNet [49], and the heavy-weight MobileNet-v1 [46]. The size of the selected models is summarized in Table 5. Typically, a neural network model contains several layers, each of which represents one or more tasks. The layers compute the sensitive data on nodes and forward them via links with weights. Following previous neural network secure inference work [64], we strictly guarantee the confidentiality of the input and output data on the nodes and verify the integrity of the model links. Based on this protection strategy, we apply the corresponding protection policy to the node and link buffers. Besides, similar to the basic benchmarks, all neural network GPU tasks are executed sequentially.

Table 5 compares the performance of the neural network models on the native system and StrongBox, and details the time cost in Task Protector protection (**TProtect**). StrongBox introduces low overhead (2.80%) on the tiny network models LeNet-5, and an acceptable overhead on the heavier SqueezeNet and MobileNet (8.48% and 19.67%, respectively). Further analysis indicates that the overhead is dominated by the protection time (**TProtect**) on the node and link buffers due to the heavy memory usage, consistent with the evaluation of basic benchmarks in **RQ2**. Moreover, the link protection generates more overhead than the node protection even though it has less memory usage. The primary reason is the varied protection strategies on the node and link buffers. StrongBox performs access control and integrity verification for the entirety of parameters in linked buffers to ensure the model integrity. Although the access control is implemented on all nodes, only nodes in the head and tail layers require cryptographic and integrity check operations. The computation results in the intermediate layers do not need to be exported, instead requiring only access control. In brief, StrongBox secures inference with reasonably low overhead, especially for models with a small link size.

## 6.4 RQ4: Evaluation on Optimization Policy

Recall from Section 4.5 that StrongBox effectively optimizes protection overhead by eliding redundant cryptographic and access permissions changes of sensitive data, which decreases the overhead caused by Task Protector. To measure improvements in performance from our approach, we compare the time consumption of total execution and Task Protector module (**TProtect**) on the selected Rodinia benchmark suite (whose problem sizes are shown in Table 3) with and without the optimization. As shown in Table 6, StrongBox effectively reduces the performance overhead in all GPU applications. For the multi-task GPU applications, **TProtect**

**Table 5: Problem size and execution time of the selected models.**

| Model | Layers | Seq. Exe. | Nodes | | Link | | Total time | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | Size | TProtect time | Size | TProtect time | Vanilla | StrongBox | Overhead |
| LeNet-5 | 6 | ✓ | 39.29 KB | 0.12 ms | 202.91 KB | 0.98 ms | 333.05 ms | 342.36 ms | 9.31 ms (2.80%) |
| SqueezeNet | 29 | ✓ | 8.60 MB | 13.31 ms | 4.62 MB | 34.36 ms | 784.44 ms | 850.97 ms | 66.53 ms (8.48%) |
| MobileNet-v1 | 17 | ✓ | 21.04 MB | 42.40 ms | 16.23 MB | 101.28 ms | 889.17 ms | 1064.07 ms | 174.90 ms (19.67%) |

introduces the most execution time (54.50% – 97.40%) when computing multi-task benchmarks without optimization. However, the proportion of this slowdown sharply decreases to 1.34% – 12.01% with our optimization. As for single-task applications, StrongBox achieves modest optimization on **TProtect** by differentiating the input and output data buffers rather than simply encrypting/decrypting all GPU buffers. Thus, by addressing the primary cause of the slowdown, StrongBox reduces the performance overhead.

## 6.5 RQ5: Comparison to Other GPU TEEs

Table 7 illustrates how StrongBox achieves our goals under different architectures and ecosystems. Intel-based devices typically employ physically-isolated dedicated GPU devices. In contrast, GPUs on Arm endpoints must share a unified memory with the vulnerable OS. This difference indicates that the Intel-based GPU TEEs can secure the computation by controlling a small-sized GPU MMIO, while Arm-based defense mechanisms must secure a large-sized main memory. In addition, all GPU TEEs support the sequential task execution, while the Intel-based GPU TEEs (i.e., Gravition [90], HIX [51], HETEE [93]) further support the concurrent secure task execution for server GPU. Worse yet, the Intel-based GPU TEEs can leverage the open-source CUDA runtime (e.g., gdev [54]), while the defense mechanism of StrongBox currently must employ a closed-source GPU runtime (specifically, OpenCL [20]). However, StrongBox still guarantees data security on Arm-based GPU platforms with unified memory.

The ecosystem and architecture variance indicate that StrongBox must focus on the protection of endpoint GPU computation. Based on this scenario, our design choices are lightweight and compatible with Arm endpoint GPUs. In particular, StrongBox avoids porting the heavy and vulnerable GPU driver into the TEE (or enclave in Intel-based GPU TEEs). As for compatibility, StrongBox requires no modification to existing devices/ISA or additional hardware for security purposes. In comparison to existing Arm-based secure GPU computation [64], StrongBox aims at general computation rather than specific machine learning inference. Moreover, StrongBox is agnostic to kernel memory management and can preserve the GPU for untrusted usage.

**Table 6: Comparison of execution time (ms) between the non-optimized mechanism and StrongBox on the selected Rodinia benchmark. The TProtect column shows the execution time in Task Protector module and its proportion to the total execution time. Note that other components except the Task Protector module (TProtect) are not modified.**

| Benchmark | | No Optimization | | StrongBox | |
|---|---|---|---|---|---|
| | | TProtect | Total | TProtect | Total |
| Single Task | KNN | 7.31 (11.55%) | 63.30 | 4.86 (7.95%) | 61.10 |
| | LMD | 1,227.88 (8.27%) | 14,854.08 | 977.46 (6.68%) | 14,626.98 |
| Multi Task | PF | 3,495.99 (54.50%) | 6,414.31 | 399.48 (12.01%) | 3,326.04 |
| | LUD | 97,179.42 (95.24%) | 102,032.57 | 338.10 (7.58%) | 4,462.57 |
| | H3D | 196,457.42 (96.87%) | 202,797.82 | 332.82 (5.85%) | 5,692.59 |
| | GS | 2,149,460.48 (97.40%) | 2,206,881.00 | 694.52 (1.34%) | 51,753.57 |

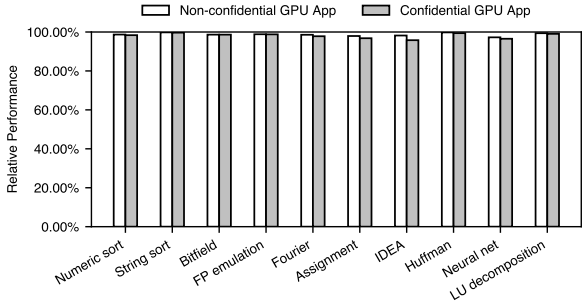


**Figure 5: Relative performance of Nbench applications when concurrently running a non-confidential/confidential GPU application.**

## 6.6 RQ6: Evaluation on System Performance

We select Nbench [27] to measure the system slowdown caused by StrongBox, which is widely used to measure the performance of CPU computation and memory intensive operations [44, 63, 92]. To demonstrate the system overhead, we select a long-running application `LMD` from the Rodinia benchmark [30] 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. Figure 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 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.

# 7 SECURITY ANALYSIS

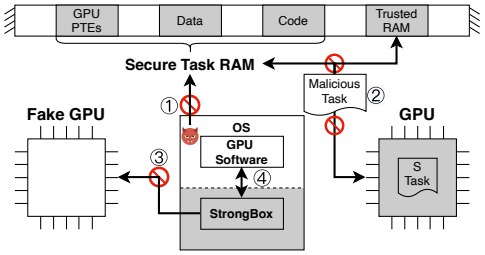


**Figure 6: Four indicative attack scenarios against the process of a confidential GPU application. ① indicates an attack on code, sensitive data, and GPU page tables in Secure Task RAM. ② represents an attack from malicious tasks. ③ represents an attack with a fake GPU. ④ shows Iago [31] attacks.**

**Table 7: Comparison to the state-of-the-art GPU TEEs and secure GPU computation.**

| | Graviton | HIX | HETEE | Secdeep | StrongBox |
|---|---|---|---|---|---|
| **Architecture and ecosystem distinction** | | | | | |
| ISA | Intel | Intel | Intel | Arm | Arm |
| GPU Task Memory | Dedicated | Dedicated | Dedicated | Unified | Unified |
| User-level driver | Open-source | Open-source | - | Closed-source | Closed-source |
| **Design distinction** | | | | | |
| Application | Server computation | Server computation | Server computation | Endpoints DL inference | Endpoints computation |
| Task Execution Model | Sequential/Concurrent | Sequential/Concurrent | Sequential/Concurrent | Sequential | Sequential |
| Position of GPU driver | Untrusted OS | Inside enclave | Inside enclave | Inside TEE | Untrusted OS |
| Hardware changes | Yes | Yes | Yes | No | No |
| Protection to task memory | GPU MMIO | GPU MMIO | GPU MMIO | TZASC + Kernel Mapping | TZASC + S-2 trans. |

## 7.1 Attack on Secure Task RAM

**Sensitive Data and Code.** As shown in Figure 6-①, 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 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.** Figure 6-① 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. Ultimately, our protection successfully prevents her from leaking sensitive data.

## 7.2 Attack with Malicious Tasks

We consider the attacker who attempts to execute an arbitrary number of malicious tasks with malicious code. As shown in Figure 6-②, 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., 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 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.

## 7.3 Attack with Fake GPU

Rather than attacks that directly compromise GPU chips, the attacker may attempt to impersonate a GPU device to spoof GPU state or submission of secure tasks (shown in Figure 6-③). However, we guarantee that StrongBox always interacts with an authentic GPU. Recall from Section 4.3 that StrongBox checks the GPU state registers and writes the task submission command by accessing the GPU MMIO registers. Based on the available manuals [3, 7, 40, 83], the physical address of embedded GPU MMIO registers is fixed and unmodifiable. Therefore, an attacker cannot change the MMIO physical address of the SoC peripherals without physical access to the AXI bus. Consequently, by accessing the unchangeable GPU MMIO physical address, StrongBox obtains the authentic GPU state and submits the secure tasks to actual GPU.

## 7.4 Attack with Compromised GPU Software

Figure 6-④ shows that the attacker may manipulate the untrusted GPU software stacks (i.e., GPU driver and GPU runtime) to launch an Iago-style attack [31], 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.

## 8 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 the 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) [10], 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 three reasons. First, recall from Section 2.3 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 [78]. Lastly, it is possible to mitigate the impact on GPU rendering by temporarily switching to software-based renderers [66].

**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.

## 9 RELATED WORK

**GPU TEEs and Secure GPU Computation.** Studies have exploited the isolation features of GPUs for secure computation. Graviton [90] uses a secure context to provide an isolated execution environment for GPUs. The maintenance of the secure context depends on a modified GPU command processor. HIX [51] extends SGX-based support on GPU enclaves by introducing new SGX instructions to secure GPU MMIO. Hence it still depends on the physical modification of devices. In addition, HIX depends on a substantial GPU software stack (e.g., GPU Driver), which can undermine the security of the enclave. HETEE [93] achieves confidential computation in a heterogeneous system with accelerators. However, the defense mechanism requires hardware devices (e.g., an additional FPGA) for implementing the trusted computation. Recently, NVIDIA released the H100 GPU [73] for confidential AI applications, while it is not proven to be feasible on unified-memory devices. Besides GPU TEEs, recent works have demonstrated privacy-preserving computing on GPUs in machine learning, such as private training [86] and inference [68, 71], while their systems are not applicable to other GPU computing applications. In StrongBox, we support general GPU computation on Arm endpoints.

**TEE-based Computing on Arm.** Recent works leverage TrustZone to secure the execution for machine learning inference [32, 55, 70], one-time password generation [84], and providing legal contracts [67]. However, they are CPU-based computations and do not use the GPU to accelerate secure computation. For the TZASC-dependent TEEs (e.g., SANCTUARY [26] and TrustICE [85]), processing sizable and discontinuous GPU buffers is not realistic due to the limited number of configurable regions in TZASC. Secdeep [64] addresses it by controlling the kernel page table and migrating GPU software stacks into TEE. Meanwhile, it severely undermines the kernel functionality and introduces a large attack surface to TEE. In StrongBox, we secure GPU with fine-grained memory protection, which presents a thin TCB and minimal effects on kernel functionality. Several TEEs (e.g., vTZ [47] and Twinvisor [62]) also create the isolated TEE via Stage-2 translation, while they consider the GPU resource as untrusted and have yet to use it for acceleration. In contrast, StrongBox achieves dynamic and complex memory protection on both the GPU tasks memory and GPU MMIO by leveraging the Stage-2 translation.

## 10 CONCLUSION

In this paper, 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 Stage-2 translation to flexibly manage access to GPU task RAM and GPU memory-mapped interfaces. Moreover, the core components of StrongBox are protected from malicious access from both the (untrusted) kernel and other peripherals. Our design requires no modification to the Arm architecture or any hardware components, providing a higher degree of compatibility than previous GPU TEEs. 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.

## 11 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. This work is partly supported by the National Natural Science Foundation of China under Grant No. 62002151 and No. 62102175, and Science, Technology and Innovation Commission of Shenzhen Municipality under Grant No. SGDX20201103095408029, and the Research Institute for Artificial Intelligence of Things, The Hong Kong Polytechnic University, and HK RGC General Research Fund No. PolyU 15220020.

# REFERENCES


[1] AlDanial. 2021. cloc. https://github.com/AlDanial/cloc.

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

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

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

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

[6] ARM. 2013. ARM Generic Interrupt Controller Architecture Specification version 2.0. https://developer.arm.com/documentation/ihi0048/latest.

[7] ARM. 2016. Juno r2 ARM Development Platform SoC. https://developer.arm.com/documentation/ddi0515/latest.

[8] ARM. 2017. Midgard Architecture. https://gitlab.freedesktop.org/panfrost/mali-isa-docs/-/blob/master/Midgard.md.

[9] Arm. 2019. Arm Mali GPU Best Practices Developer Guide. https://documentation-service.arm.com/static/5e8e117388295d1e18d34ac9?token=.

[10] ARM. 2021. Granule Protection Tables in TF-A. https://www.trustedfirmware.org/docs/tfa_tech_forum_2021_10_21_gpt.pdf.

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

[12] ARM. 2022. Arm Cortex-A53 MPCore Processor Technical Reference Manual. https://developer.arm.com/documentation/ddi0500/latest/.

[13] ARM. 2022. Arm Cortex-A57 MPCore Processor Technical Reference Manual. https://developer.arm.com/documentation/ddi0488/latest/.

[14] ARM. 2022. Arm Cortex-A72 MPCore Processor Technical Reference Manual. https://developer.arm.com/documentation/100095/latest/.

[15] ARM. 2022. ARM Mali-T600 Series GPU OpenCL Developer Guide. https://developer.arm.com/documentation/dui0538/latest/.

[16] ARM. 2022. Cortex-A7 MPCore Technical Reference Manual. https://developer.arm.com/documentation/ddi0464/latest/.

[17] ARM. 2022. Game engine guides. https://developer.arm.com/solutions/graphics-and-gaming/developer-guides/game-engine-guides.

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

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

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

[21] ARM. 2022. TRUSTZONE FOR CORTEX-A. https://www.arm.com/technologies/trustzone-for-cortex-a.

[22] ARM. 2022. VR best practice. https://developer.arm.com/solutions/graphics-and-gaming/developer-guides/vr-best-practice.

[23] ASUS IoT. 2022. ASUS IoT Face Recognition Edge AI Dev Kit. https://iot.asus.com/solutions/facerecognition/.

[24] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F Donaldson, Jeroen Ketema, et al. 2015. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 138–149.

[25] Valentin Bazarevsky, Yury Kartynnik, Andrey Vakunov, Karthik Raveendran, and Matthias Grundmann. 2019. Blazeface: Sub-millisecond neural face detection on mobile gpus. *arXiv preprint arXiv:1907.05047* (2019).

[26] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2019. SANCTUARY: ARMing TrustZone with User-space Enclaves.. In *NDSS*.

[27] BYTE Magazine. 2022. Linux/unix nbench. http://www.tux.org/~mayer/linux/bmark.html.

[28] Qingqing Cao, Niranjan Balasubramanian, and Aruna Balasubramanian. 2017. MobiRNN: Efficient recurrent neural network execution on mobile GPU. In *Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications*. 1–6.

[29] Chin-Chen Chang, Wai-Kong Lee, Yanjun Liu, Bok-Min Goi, and Raphael C-W Phan. 2018. Signature gateway: Offloading signature generation to IoT gateway accelerated by GPU. *IEEE Internet of Things Journal* 6, 3 (2018), 4448–4461.

[30] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[31] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: Why the system call API is a bad untrusted RPC interface. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 253–264.

[32] Jinwoo Choi, Jaeyeon Kim, Chaemin Lim, Suhyun Lee, Jinho Lee, Dokyung Song, and Youngsok Kim. 2022. GuardiaNN: Fast and Secure On-Device Inference in TrustZone Using Embedded SRAM and Cryptographic Hardware. (2022).

[33] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016, 86 (2016), 1–118.

[34] CVE. 2020. CVE-2020-5991. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-5991.

[35] CVE. 2021. CVE-2021-1093. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-1093.

[36] CVE. 2021. CVE-2021-1121. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-1121.

[37] CVE. 2022. CVE-2022-21815. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-21815.

[38] CVE. 2022. CVE-2022-21821. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-21821.

[39] Eklektix, Inc. 2021. A deep dive into cma. https://lwn.net/Articles/486301/.

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

[41] Vaibhav Gandhi, Hongqiang Wang, and Alex Bourd. 2019. Optimization of Fast Fourier Transform (FFT) on Qualcomm Adreno GPU. In *Proceedings of the International Workshop on OpenCL*. 1–2.

[42] Google. 2022. GPUs on Compute Engine. https://cloud.google.com/compute/docs/gpus/.

[43] S Heymann, K Müller, A Smolic, B Froehlich, and T Wiegand. 2007. SIFT implementation and optimization for general-purpose GPU. (2007).

[44] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. 2022. PACSafe: Leveraging ARM Pointer Authentication for Memory Safety in C/C++. *arXiv preprint arXiv:2202.08669* (2022).

[45] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. 2019. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[46] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[47] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. vtz: Virtualizing ARM trustzone. In *26th USENIX Security Symposium (USENIX Security 17)*. 541–556.

[48] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J Rossbach, and Emmett Witchel. 2020. Telekine: Secure computing with cloud gpus. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 817–833.

[49] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[50] Vaibhav Jain and Dinesh Patel. 2016. A GPU based implementation of robust face detection system. *Procedia Computer Science* 87 (2016), 156–163.

[51] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous isolated execution for commodity gpus. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 455–468.

[52] Jin Soo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. 2015. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment.. In *NDSS*.

[53] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. 2022. Band: coordinated multi-DNN inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 235–247.

[54] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. 2012. Gdev: First-class GPU resource management in the operating system. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 401–412.

[55] Eugene Kuznetsov, Yitao Chen, and Ming Zhao. 2021. SecureFL: Privacy Preserving Federated Learning with SGX and TrustZone. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 55–67.

[56] Miguel Lastra, Jesús Carabaño, Pablo D Gutiérrez, José M Benítez, and Francisco Herrera. 2015. Fast fingerprint identification using GPUs. *Information Sciences* 301 (2015), 195–214.

[57] Seyyed Salar Latifi Oskouei, Hossein Golestani, Matin Hashemi, and Soheil Ghiasi. 2016. Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android. In *Proceedings of the 24th ACM international conference on Multimedia*. 1201–1205.

[58] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[59] Hyeonsu Lee, Hyunjun Kim, Cheolgi Kim, Hwansoo Han, and Euiseong Seo. 2020. Idempotence-based preemptive GPU kernel scheduling for embedded systems. *IEEE Trans. Comput.* 70, 3 (2020), 332–346.

[60] Hyeonsu Lee, Jaehun Roh, and Euiseong Seo. 2018. A GPU kernel transactionization scheme for preemptive priority scheduling. In *2018 IEEE Real-Time and*

*Embedded Technology and Applications Symposium (RTAS)*. IEEE, 202–213.

[61] Jingyu Lee, Yunxin Liu, and Youngki Lee. 2021. ParallelFusion: towards maximum utilization of mobile GPU for DNN inference. In *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*. 25–30.

[62] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. 2021. TwinVisor: Hardware-isolated Confidential Virtual Machines for ARM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 638–654.

[63] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. 2019. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*. 177–194.

[64] Renju Liu, Luis Garcia, Zaoxing Liu, Botong Ou, and Mani Srivastava. 2021. SecDeep: Secure and Performant On-device Deep Learning Inference Framework for Mobile and IoT Devices. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation*. 67–79.

[65] Svetlin A Manavski. 2007. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *2007 IEEE International Conference on Signal Processing and Communications*. IEEE, 65–68.

[66] Mesa 3D. 2022. The Mesa 3D Graphics Library. https://www.mesa3d.org/.

[67] Saeed Mirzamohammadi, Yuxin Liu, Tianmei Ann Huang, Ardalan Amiri Sani, Sharad Agarwal, and Sung Eun Kim. 2020. Tabellion: secure legal contracts on mobile devices. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*. 220–233.

[68] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*. 2505–2522.

[69] Perhaad Mistry, Chris Gregg, Norman Rubin, David Kaeli, and Kim Hazelwood. 2011. Analyzing program flow within a many-kernel OpenCL application. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. 1–8.

[70] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. 2020. DarkneTZ: towards model privacy at the edge using trusted execution environments. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*. 161–174.

[71] Lucien KL Ng and Sherman SM Chow. 2021. GForce: GPU-Friendly Oblivious and Rapid Neural Network Inference. In *30th USENIX Security Symposium (USENIX Security 21)*. 2147–2164.

[72] NVIDIA. 2022. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit.

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

[74] NVIDIA. 2022. NVIDIA DATA CENTER GPUs. https://www.nvidia.com/en-us/data-center/data-center-gpus/.

[75] Pierre Paleo, Emeline Pouyet, and Jérôme Kieffer. 2014. Image stack alignment in full-field X-ray absorption spectroscopy using SIFT_PyOCL. *Journal of synchrotron radiation* 21, 2 (2014), 456–461.

[76] Pytorch. 2022. Learning PyTorch with Examples. https://pytorch.org/tutorials/beginner/pytorch_with_examples.html.

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

[78] Ajita Rattani and Reza Derakhshani. 2018. A survey of mobile face biometrics. *Computers & Electrical Engineering* 72 (2018), 39–52.

[79] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1. IEEE, 57–64.

[80] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[81] Reza Shokri and Vitaly Shmatikov. 2015. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 1310–1321.

[82] STMicroelectronics. 2022. Artificial Intelligence (AI) face recognition function pack for STM32Cube. https://www.st.com/en/embedded-software/fp-ai-facerec.html.

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

[84] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. 2015. TrustOTP: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 976–988.

[85] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. 2015. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 367–378.

[86] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. 2021. CRYPTGPU: Fast Privacy-Preserving Machine Learning on the GPU. *arXiv preprint arXiv:2104.10949* (2021).

[87] Tensorflow. 2022. Use a GPU. https://www.tensorflow.org/guide/gpu.

[88] José F Torres, Dalil Hadjout, Abderrazak Sebaa, Francisco Martínez-Álvarez, and Alicia Troncoso. 2021. Deep learning for time series forecasting: a survey. *Big Data* 9, 1 (2021), 3–21.

[89] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. 2019. A hybrid approach to privacy-preserving federated learning. In *Proceedings of the 12th ACM workshop on artificial intelligence and security*. 1–11.

[90] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted execution environments on gpus. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 681–696.

[91] Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. 2022. RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1573–1573.

[92] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2020. A Framework for Software Diversification with ISA Heterogeneity. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 427–442.

[93] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, et al. 2020. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1450–1465.