Mole: Breaking GPU TEE with GPU-Embedded MCU

Hongyi Lu[*†‡]
hluaw@cse.ust.hk
Research Institute of Trustworthy Autonomous Systems
Southern University of Science and Technology
China

Yunjie Deng[*]
Sukarno Mertoguno
{yunjie.deng,karno}@gatech.edu
Georgia Institute of Technology
USA

Shuai Wang[§]
shuaiw@cse.ust.hk
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
China

Fengwei Zhang[§¶]
zhangfw@sustech.edu.cn
Department of Computer Science and Engineering
Southern University of Science and Technology
China

Abstract

Graphics Processing Units (GPUs) are extensively used for applications such as machine learning, scientific computing, and graphics rendering. To protect sensitive data processed by GPUs, Trusted Execution Environments (TEEs) for GPUs have been proposed. GPU TEEs, built with hardware-based isolation primitives, can defend against high-privilege attackers like OS kernels. However, in this paper, we present Mole, a novel attack that compromises the security of GPU TEEs on Arm Mali GPUs by exploiting the GPU-embedded Microcontroller Unit (MCU). By injecting a malicious firmware into the MCU, an attacker can bypass GPU TEEs' security guarantees. We evaluated Mole with state-of-the-art GPU TEE proposals under multiple real-world attack scenarios such as in-GPU AES encryption and object detection tasks. Our evaluation shows that Mole can successfully extract sensitive data or manipulate the computation results of GPU TEEs. We responsibly disclosed our findings to the authors of the affected GPU TEE proposals and received acknowledgments from all of them. Moreover, our findings prompted Arm to enhance the security of its GPU firmware supply-chains.

CCS Concepts

• **Security and privacy → Systems security**; **Embedded systems security**; **Trusted computing**.

Keywords

GPU Security, Trusted Execution Environment, Firmware Attack

[*]Hongyi Lu and Yunjie Deng contributed equally to this work.
[†]Also with Department of Computer Science and Engineering, Southern University of Science and Technology.
[‡]Also with Department of Computer Science and Engineering, Hong Kong University of Science and Technology.
[§]Shuai Wang and Fengwei Zhang are the corresponding authors.
[¶]Also with Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology.

CCS '25, Taipei, Taiwan, China

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3744823

ACM Reference Format:
Hongyi Lu, Yunjie Deng, Sukarno Mertoguno, Shuai Wang, and Fengwei Zhang. 2025. Mole: Breaking GPU TEE with GPU-Embedded MCU. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan, China.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3744823

1 Introduction

Over the past decade, Graphics Processing Units (GPUs) have evolved significantly, moving beyond their original role in graphics rendering to become critical components in high-performance computing [12, 29, 55, 64]. Furthermore, to meet the escalating performance demands of edge computing, GPUs are no longer confined to desktops and servers but are now extensively integrated into mobile and embedded systems, including smartphones and autonomous vehicles. Despite the widespread adoption, GPU security remains a critical concern. Numerous studies have highlighted that the absence of robust isolation mechanisms leaves GPUs susceptible to a variety of attacks [26, 33, 48, 54, 57, 70]. In response to these security breaches, GPU Trusted Execution Environments (TEEs) have been proposed. To provide an isolated environment for sensitive data processing, GPU TEEs typically employ existing TEE mechanisms, such as Intel SGX [66] and Arm TrustZone [25], or even design their own customized trusted hardware [62, 72].

To avoid putting bloated drivers into the Trusted Codebase (TCB), GPU TEEs often adopt a design that protects only security-critical driver routines (referred to as shim), while leaving the remainder of the driver stack in the untrusted domain. Many existing GPU TEEs follow this approach, such as StrongBox [25], CAGE [63], and MyTEE [34]. These shim-style GPU TEEs incorporate only security-critical routines (e.g., memory allocation) into the TCB, while delegating non-critical operations (e.g., performance monitoring) to the untrusted domain. This design minimizes the TCB size and maintains compatibility with unmodified applications, achieving a balance between security and practicality.

This work, for the first time, uncovers a novel attack scheme that undermines the security guarantees of such shim-style GPU TEEs on Arm. Through a comprehensive analysis of modern Arm GPU, we discover that the Arm Mali GPU [21] incorporates an under-documented Microcontroller Unit (MCU), which serves as a bridge between the CPU and GPU. This MCU is responsible for receiving GPU tasks from the CPU and dispatching them to the

GPU's compute unit. Furthermore, we find that the presence of the MCU undermines the security model of the shim-style GPU TEEs. To reduce the TCB size, shim-style GPU TEEs normally delegate the initialization tasks to the untrusted domain, given that no sensitive data is being processed during initialization, and the routine is no longer executed afterward. However, the MCU is initialized by the untrusted GPU driver and plays an important role in the execution of GPU tasks. A malicious OS can easily leverage this security oversight to load malicious firmware into the MCU, effectively breaking the security guarantees of the GPU TEE.

Through a series of experiments, we reveal the internal design of this MCU and its associated security implications, which form the foundation of our attack. We also conduct a systematic study of real-world devices and identify that numerous devices from four System-on-Chips (SoC) vendors and 15 device vendors use this MCU. The affected devices include mobile phones, Internet-of-Thing (IoT) devices, and tablets. Furthermore, we demonstrate the practicality of our attack in real-world scenarios, including key extraction from an in-GPU AES implementation and result tampering of a widely used object detection model (YOLOv4-tiny). All attacks are conducted under the protection of GPU TEE, replicating the setup of prior works [25, 34, 63]. Finally, we have reported our findings to the authors of the affected GPU TEEs, all of whom have confirmed the validity of our attack [11].

Our findings show the intricacy of GPU TEEs from both the following perspectives. From a software perspective, the MCU introduces significant challenges to the low TCB requirement of GPU TEEs. The code governing the MCU is tightly integrated with the in-kernel GPU driver, making it extremely challenging to isolate and protect from the OS. Additionally, the lack of publicly available design details for the MCU complicates efforts by developers to assess its security. From a hardware perspective, shim-style GPU TEEs underestimate the complexity of modern GPUs and focus solely on protecting GPU tasks from the CPU and other system devices. As a result, the MCU, as an integral component of the GPU, is overlooked in their threat models, leaving it vulnerable to exploitation by the OS to launch attacks on GPU TEEs. We hope that our work can raise awareness of the MCU (as well as similar hidden controllers) and advocate that future TEE proposals take such components into consideration.

In summary, our contributions are as follows:

- This paper presents the first attack scheme targeting GPU TEEs on Arm devices. By exploiting an under-documented MCU inside the Arm GPU, we successfully compromise the security guarantees of GPU TEEs.[1]
- Technically, we provide a comprehensive analysis of the Arm GPU's MCU architecture. We demonstrate that the GPU software stack, considered untrusted by Arm GPU TEEs, can control the GPU MCU—a trusted component that undermines isolation.
- To assess the impact of our attack, we conduct a systematic study of the affected GPU TEE proposals and perform real-world experiments, including key extraction and result tampering, to validate the practicality of our attack.

[1]We release the codebase of Mole on our site [11].

2 Preliminary and Motivation

In this section, we describe the hardware components of a modern Arm GPU and review Arm's security features employed in GPU TEE. We then present the motivation behind our proposed attack.

2.1 Arm GPU Components

We now describe the hardware components of the Arm Mali GPU, which are critical to our discussion. As shown in Fig. 1, the Arm Mali GPU comprises three internal components, the compute unit, the rendering unit and an MCU. Notably, since 2021, all Arm Mali GPUs embed an MCU to offload the scheduling of GPU tasks from the driver. When a GPU task is submitted to the driver, the driver sends an Interrupt Request (IRQ) to notify the MCU, which then dispatches the task to the compute unit or render unit as needed. This design simplifies the kernel driver and enhances performance [21]. Despite its crucial role in GPU-CPU communication, the MCU is under-documented, and its security implications are overlooked in the threat model of existing GPU TEEs.

The overlooking of the MCU in the existing GPU TEEs motivates us to investigate its security implications and whether it can be exploited by the adversary to compromise GPU TEEs.

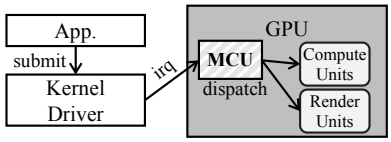


Figure 1: The components of Arm GPU.

2.2 Arm Security Features

As illustrated in Fig. 2, the features employed in current GPU TEEs include TrustZone (Sec. 2.2.1), Two-Stage Address Translation (Sec. 2.2.2), System Memory Management Unit (Sec. 2.2.3), and Confidential Compute Architecture (Sec. 2.2.4).

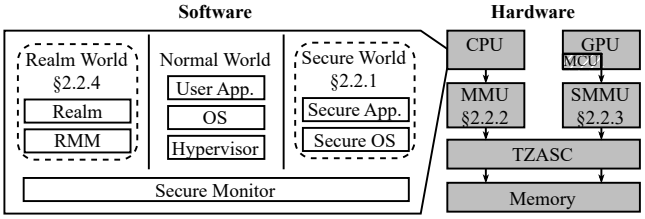


Figure 2: Arm security features.

2.2.1 TrustZone. Arm introduced TrustZone [20] to provide hardware isolation mechanism for sensitive codebases on its devices. Specifically, TrustZone partitions execution into two worlds: the *secure world* and the *normal world*. The secure world provides a hardware-based TEE for trusted OS and applications, while the normal world hosts conventional OS and untrusted applications; access to secure service from the normal world is restricted to controlled mechanisms, such as privileged `smc` instruction. TrustZone's hardware components enforce strict isolation for sensitive

applications within the secure world. One such component is the TrustZone Address Space Controller (TZASC). Integrated into the memory controller, the TZASC regulates all traffic between the main processor and memory. Additionally, to manage traffic between peripherals and memory, the TZASC assigns a Non-Secure Access Identity (NSAID) to each peripheral device, such as the GPU. When a peripheral accesses content stored in the memory, the TZASC verifies whether its NSAID has the necessary permissions. However, while the TZASC accounts for both CPU and peripheral access, it cannot distinguish between accesses from different components of the same device. For example, the TZASC cannot differentiate between the accesses from the GPU MCU and those from the GPU compute units.

2.2.2 Two-Stage Address Translation. Though the TZASC can regulate memory access from both the CPU and peripheral devices, it supports only eight configurable regions—far fewer than required by GPU TEEs. To address this limitation, GPU TEEs like StrongBox [25] utilize Arm's two-stage address translation. Arm defines two stages for address translation [19], Stage-1 and Stage-2. Stage-1 translates a virtual address (VA) to an intermediate physical address (IPA), typically used to isolate the OS's address space from applications. Stage-2 further translates the IPA to a concrete physical address (PA). By leveraging two-stage address translation, GPU TEEs like StrongBox can isolate GPU tasks from privileged adversaries, such as the OS kernel. For example, after a user submits a secure GPU task to the GPU TEE, the GPU TEE can prevent the kernel from accessing the GPU task by invalidating the corresponding Stage-2 translation table entries. However, as we will demonstrate in Sec. 4, this isolation is effective only for the CPU and not for peripheral devices like the GPU.

2.2.3 System Memory Management Unit. On Arm devices, the peripheral devices like GPU typically share the same main memory with the CPU. To regulate the Direct Memory Access (DMA) requests from these peripherals, Arm introduced the System Memory Management Unit (SMMU). Similar to CPU's MMU, SMMU translates the device's virtual address to the physical address and verifies whether the access is permitted. By deploying and configuring the SMMU between peripherals and the main memory controller, privileged software can prevent memory corruption attacks from these peripheral devices. The SMMU is also used in GPU TEEs to isolate the GPU from other potentially malicious components. However, like the TZASC, the SMMU cannot differentiate between accesses from different components of the same device.

2.2.4 Confidential Compute Architecture. In the Armv9 architecture, Arm introduced the Confidential Compute Architecture (CCA). CCA [14] retains the separation between the normal world and the secure world while introducing a new *realm world*. In the realm world, confidential realms can be created and managed by the Realm Management Monitor (RMM). To isolate the realms from each other, CCA introduces a hardware isolation primitive called Granule Protection Check (GPC). In GPC, each physical memory page is treated as a granule. When a software component accesses a physical address, GPC performs an additional check against the Granule Protection Table (GPT) to determine if the access is permitted, beyond existing checks from the MMU and TZASC. To enable flexible isolation, the GPT follows a two-level hierarchical structure, similar to the page table, where internal nodes point to the next level of the GPT and the leaf nodes store granule permission. By leveraging GPC, CCA provides a more flexible and fine-grained isolation mechanism compared to TrustZone. However, like the TZASC and SMMU, CCA lacks the capability to handle the internal components inside a device. Based on our examination of the documentation[2], CCA integrates the GPT into the SMMU to support memory isolation for peripherals, meaning it inherits the SMMU's approach to device protection. As the SMMU cannot distinguish between accesses from internal components of a device, CCA is unable to mitigate the security threat originating from the MCU.

2.3 Motivation

Two key observations motivate our work on exploiting the embedded MCU to attack GPU TEEs.

Powerful MCU. Our investigation reveals that the MCU is a general-purpose processor based on Armv7-M instruction set; it not only enhances GPU performance but also becomes the potential target of malicious code. With a compromised MCU, the adversary can exploit it to spy on or modify any GPU task. Unfortunately, current GPU TEE designs lack adequate protection for the GPU MCU, creating a significant attack surface for adversaries.

Ambiguous Isolation Boundary. As discussed in Sec. 2.2, Arm's security features, such as TZASC and SMMU, exhibit an ambiguous isolation boundary for components *inside* the same peripheral device. Though these features are designed to isolate *secure* components from *non-secure* components, they cannot establish a clear isolation boundary *within* the same devices (e.g., between the GPU and the GPU MCU). This limitation enables the MCU to bypass the isolation enforced by the GPU TEE and directly access sensitive GPU data, as no clear isolation boundary exists between the GPU MCU and the GPU.

Motivation. Overall, these observations motivate us to investigate the security risks posed by the GPU MCU to current GPU TEEs. Since Arm's security features cannot distinguish between accesses from the GPU and its internal MCU, a compromised MCU can undermine current Arm GPU TEE due to insufficient isolation.

3 Arm GPU TEE and Threat Model

In this section, we first review state-of-the-art GPU TEEs on the Arm platform in Sec. 3.1. We then perform a detailed security analysis of these GPU TEEs and explain why a key assumption is invalid in Sec. 3.2. Based on this analysis, we present the threat model of our attack in Sec. 3.3.

3.1 SoTA GPU TEEs on Arm

Table 1 summarizes existing GPU TEEs on the Arm platform. As Table 1 shows, Arm GPU TEEs can be broadly categorized into two groups based on their design choices: VM-style and shim-style. Fig. 3 illustrates the key differences between these two designs of GPU TEEs. We discuss each of them below.

VM-style GPU TEEs. As illustrated in the left part of Fig. 3, VM-style GPU TEEs utilize virtualization techniques to create a dedicated VM for each GPU Trusted Application (TA). By isolating the

[2]CCA has not yet been implemented on commercial hardware.

Table 1: Existing GPU TEEs on Arm platform.

Category	Name	Security Mechanism	TCB LoC
Shim-style	StrongBox	TZASC + Stage-2	1.7K
	CAGE	GPC + SMMU	1.3K
	MyTEE	Stage-2	4K
VM-style	Cronus	TZASC	72.6K
	GR-T	TZASC + Remote VM	>10M
	ACAI	GPC + SMMU	25.5M

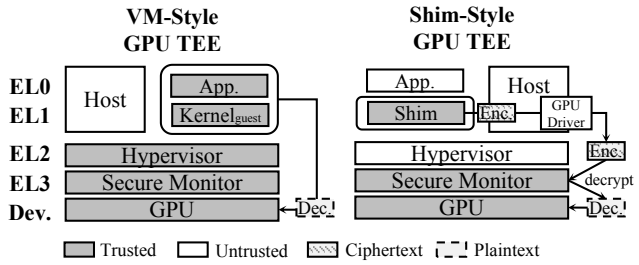


Figure 3: Comparison between two types of GPU TEEs.

entire GPU software stack in a confidential VM, the TA can directly send sensitive data to the GPU through the guest kernel's bundled driver, which is considered part of the TCB.

Cronus [40] and GR-T [53] are two representative VM-style GPU TEEs based on Arm TrustZone technology. Cronus extends TrustZone with a secure hypervisor to manage multiple confidential VMs in the secure world and coordinate their access to the GPU. GR-T, on the other hand, employs a remote VM to send GPU commands to a cloud server, where a trusted client executes these commands on an actual GPU. ACAI [58] is another VM-style GPU TEE built on the emerging Arm CCA technology. ACAI extends CCA's GPC and RMM to protect GPU TAs from untrusted adversaries.

Despite providing strong isolation and compatibility, VM-style GPU TEEs have a significant drawback: they require a large TCB. In a VM-style GPU TEE, the entire software stack (e.g., GPU driver and guest kernel) is bundled into a monolithic trusted VM. As shown in Table 1, ACAI treats the whole VM as trusted, which is over 25.5M LoC. GR-T also introduces a non-trivial TCB of size exceeding 10M LoC. Such a bloated trusted codebase is highly error-prone and exposes a large attack surface.

Shim-style GPU TEEs. In response to the bloated TCB in VM-style GPU TEEs, shim-style GPU TEEs (e.g., StrongBox [25], CAGE [63], and MyTEE [34]) have been proposed. As shown in the right part of Fig. 3, instead of bundling the entire guest kernel into the TCB, shim-style GPU TEEs deploy a minimal shim layer in the TCB for critical routines, such as GPU task management, while leaving most non-critical routines, such as performance monitoring, to the untrusted world. Since the shim layer has limited functionality, it must reuse part of the GPU driver in the untrusted world. To prevent the untrusted GPU driver from accessing the sensitive data of secure GPU tasks, shim-style GPU TEEs encrypts the sensitive data before exposing it to the untrusted GPU driver. Subsequently, shim-style GPU TEEs decrypt the sensitive data before sending it to the GPU, since the GPU cannot directly operate on the encrypted data.

StrongBox [25] is the first shim-style GPU TEE on Arm endpoints. It leverages TrustZone's TZASC and Stage-2 address translation to protect GPU TAs from untrusted adversaries. CAGE [63] is another shim-style GPU TEE that utilizes the latest Arm CCA technology to protect GPU TAs. MyTEE [34], designed for devices with limited security features, adopts a novel design that relies solely on Stage-2 address translation to protect GPU TAs. These shim-style GPU TEEs include only a minimal shim layer in the TCB. As shown in Table 1, the TCB of StrongBox is just 1.7K LoC, which is orders of magnitude smaller than the TCB of VM-style GPU TEEs.

Security of Shim-style GPU TEEs. Though shim-style GPU TEEs achieve the design goal of low TCB, they reuse the GPU driver in the untrusted world to handle so-called non-critical routines, such as performance monitoring. In the following section, we will analyze this shim-style design and show that this design choice introduces a critical security flaw.

3.2 Security Analysis

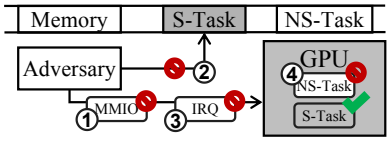


Figure 4: Security primitives in shim-style GPU TEEs.

Despite minor differences between StrongBox [25], CAGE [63], and MyTEE [34], their shim layers share a similar design: delegating security-critical routines (e.g., memory management) to the shim layer while offloading non-critical routines (e.g., GPU initialization) to the untrusted world. Thus, shim-style GPU TEEs achieve their security goal (i.e., a minimized TCB) while maintaining compatibility (i.e., the GPU driver is reused).

To protect these security-critical routines in the shim layer, the following security primitives are deployed: ① GPU control isolation (e.g., MMIO isolation); ② Memory protection (e.g., AES encryption, GPC, and Stage-2); ③ Interrupt isolation (e.g., GIC); ④ Security-aware scheduler. Fig. 4 shows how these security primitives mitigate different types of attacks launched by an adversary. For instance, when a secure task is executing, the adversary might attempt to interfere with the secure task by tampering with the GPU control registers via MMIO. To prevent this, the GPU control isolation (i.e., ① in Fig. 4) is used to block the adversary from accessing the GPU control registers during secure GPU tasks. Similar security primitives are also applied to prevent other types of attacks, such as memory corruption (②), forged interrupts (③), and malicious scheduling (④).

Table 2: Security primitives used in shim-style GPU TEEs.
①: *GPU Ctrl* ②: *Memory* ③: *Interrupt* ④: *Scheduler* ×: *None.*

GPU TEE	GPU Mgmt.	S-Task	NS-Task
StrongBox	×	①②③④	④
CAGE	×	①②③④	④
MyTEE	×	①②③	×

We also perform a detailed analysis of the usage of these security primitives of StrongBox, CAGE, and MyTEE for GPU management and GPU task execution. Our findings are summarized in Table 2. Though shim-style GPU TEEs do not deploy any security primitives for GPU management routines, this is not considered as a security issue. This is because the GPU management routines do not process any sensitive data or chronologically overlap with the execution of secure GPU tasks. Moreover, excluding these GPU management routines from the shim layer significantly reduces the complexity of the shim layer. Otherwise, the shim layer would become bloated and expose a large attack surface. For example, performance monitoring routine alone in Arm GPU driver comprises 7K LoC, which is four times the size of the TCB of the entire shim layer in GPU TEEs like CAGE (1.3K LoC).

From the analysis above, it seems that, by introducing a minimal shim layer into the TCB, shim-style GPU TEEs achieve a good balance between security and compatibility. However, this design relies on a key assumption: *the GPU management routines left outside the TCB are not security-critical*. As we will discuss in the following paragraph, this assumption is invalid and can be exploited to compromise the security guarantees of shim-style GPU TEEs.

Invalid Assumption in Shim-style GPU TEEs. Though most GPU management routines are *not* security-critical (e.g., performance monitoring) and thus cannot be exploited by adversaries, there is one exception: *firmware management*. Specifically, the firmware management routine is responsible for loading the firmware into the GPU MCU and performing necessary initialization. As discussed in Sec. 3.2, all existing shim-style GPU TEEs assume that the firmware management routine is not security-critical; it neither processes any sensitive data nor chronologically overlaps with the execution of secure GPU tasks. However, as we will show in Sec. 4, the MCU is not constrained by existing security primitives on the Arm platform and can therefore access the protected GPU memory. Moreover, since the firmware is *persistently* executed in the GPU MCU, even though the firmware management routines do not chronologically overlap with secure GPU tasks, malicious firmware inside the MCU continues to execute during the secure GPU tasks. This effectively invalidates the security assumption made by shim-style GPU TEEs and creates a new attack surface for adversaries. Note that shim-style GPU TEEs might also suffer other attack surfaces like fault injection [49, 59], as their power management routines are also exposed to the untrusted world. However, unlike Mole, such attacks must be conducted during the secure GPU tasks. Previous works like StrongBox mitigate such attacks by locking the power-control registers during secure GPU tasks.

Table 3: Trusted Components in Arm GPU TEEs.

Category	Name	Trusted Components	Vulnerable
Shim-style	StrongBox	Shim	✔
	CAGE	Shim	✔
	MyTEE	Shim	✔
VM-style	Cronus	Driver + microOS	✘
	GR-T	Remote VM + Local Shim	✘
	ACAI	VM	✘

We also perform an in-depth study of Arm GPU TEEs regarding their trusted components. Our findings are summarized in Table 3. In all shim-style GPU TEEs, only a minimal shim layer is trusted, and the firmware management routine is left outside the TCB. Therefore, an adversary can load a malicious firmware into the GPU MCU during GPU initialization and use it as a trampoline to access the sensitive data processed by the GPU TEE. As a result, we mark shim-style GPU TEEs like StrongBox as vulnerable in Table 3. However, the same does not apply to the VM-style GPU TEEs, as they adopt a stronger security assumption by treating the entire driver as trustworthy and free of vulnerability. As a consequence, an adversary cannot exploit the GPU driver to arbitrarily manipulate the GPU MCU. We thus mark VM-style GPU TEEs like Cronus as not vulnerable in Table 3.

Deployment of Shim-style GPU TEEs. Though shim-style GPU TEEs are still academic prototypes, industry involvement (e.g., CAGE/StrongBox is contributed by authors from Alibaba) suggests future production deployment. Our research highlights the complexity of this design and warns the community and vendors about the serious, yet overlooked, threat. In addition to GPU, this shim-style design is frequently explored by researchers to protect other peripherals while minimizing the TCB [47, 71]. For example, Zhou et al. [71] isolate the USB peripherals by outsourcing most of the USB driver to the untrusted world, achieving a TCB reduction of 99%. Therefore, we emphasize that Mole's security implications should not be underestimated, as this shim-style paradigm is not only being developed by industry (e.g., Alibaba) but is also a popular research direction in academia. Mole highlights that careful consideration of hardware complexity remains crucial when employing this approach.

3.3 Threat Model

Since Mole is based on the invalid assumption made by shim-style GPU TEEs, our threat model aligns with the threat model of shim-style GPU TEEs. We assume a high-privileged adversary who has control over the GPU driver outside the shim layer's TCB. Consequently, we assume the attacker has control over the GPU firmware management, which is responsible for loading the firmware into the GPU MCU. We note that this threat model aligns with the threat model of shim-style GPU TEEs [25, 34, 63]. Moreover, we also analyze the codebase of the vulnerable GPU TEEs (i.e., StrongBox, CAGE, and MyTEE) and confirm that the firmware management routine is left outside the TCB and can be exploited by the attacker.

Clarification and Notation. Since Mole is not applicable to VM-style GPU TEEs, for the remainder of this paper, we will focus on the shim-style GPU TEEs. For brevity, we will use GPU TEEs to refer to shim-style GPU TEEs and only specify shim-style GPU TEEs or VM-style GPU TEEs when necessary.

4 Security Implications of GPU MCU

As mentioned in Sec. 3.2, the presence of the GPU MCU invalidates the assumptions of GPU TEEs and leads to exploitation. In this section, we thoroughly explore the design of the GPU MCU, reveal its security implications, and demonstrate the feasibility of implementing Mole.

Setup. We explore the GPU MCU design through experiments on a RK3588 IoT board [56] and validate our findings on a Google Pixel 8 [32], a popular mobile phone with a Tensor G3 processor.

Both devices are equipped with the Arm Mali GPU required by Mole and standard Arm security features for GPU TEEs. As for GPU TEEs, we choose StrongBox as the target since the CCA used by CAGE is not available on commercial hardware, and MyTEE relies solely on Stage-2 isolation, which is a subset of StrongBox. That said, we received confirmation from authors of CAGE, MyTEE and StrongBox that Mole is applicable to their designs [11].

We also perform a comprehensive study of the Android Open Source Project [4], MediaTek [8] and Linux upstream [5], and find they use similar or even identical firmware from Arm upstream. This demonstrates that the design of embedded MCUs among different devices is consistent, indicating our findings are applicable to different real-world devices. The following sections are organized as follows: Sec. 4.1 introduces the architecture of the MCU; Sec. 4.2 discusses the possibility of tampering with the MCU firmware; Sec. 4.3 explores the effectiveness of different memory protection mechanisms against the MCU; Sec. 4.4 investigates how to trigger malicious code through in-MCU interrupt handling mechanism.

4.1 MCU Architecture

Since limited information is available about the MCU on the Arm Mali GPU, we conducted a thorough reverse-engineering of the firmware binary and an in-depth analysis of the GPU kernel driver. This part was performed manually by two experienced authors and took 2 months to complete. We summarize our findings in Fig. 5.

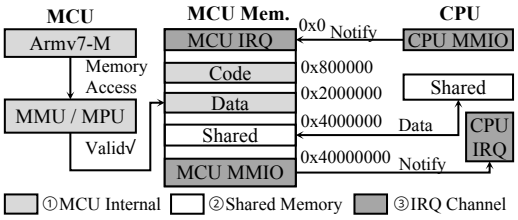


Figure 5: GPU MCU architecture.

As shown in ① MCU Internal of Fig. 5, the GPU MCU is a Cortex-M [18] series MCU with the Armv7-M instruction set [15]. During the analysis of the GPU kernel driver, we find that the MCU is equipped with a Memory Management Unit (MMU). The MMU supports three levels of page tables and partitions the MCU's memory into different sections with specific permission (e.g., code and data). Note that standard Cortex-M MCUs do not have an MMU; this is a customized design for GPU. Unlike a CPU MMU, this MMU supports up to 16 *address spaces* with 16 different page tables. The 0_{th} address space is reserved for the GPU MCU while the rest are used for different GPU tasks. Since the MMU can only be configured by the OS kernel, it effectively isolates the GPU MCU from memory used by GPU tasks; we will discuss how we deactivate this isolation by manipulating MCU's page table in Sec. 4.3 under the assumption of GPU TEEs. In addition to MMU, the MCU also includes a Memory Protection Unit (MPU) [16] to check its memory access. Unlike MMU, MPU is a standard feature of Cortex-M series, supporting up to 8 regions with configurable permission. Since the MPU is configured by the firmware, its protection can be easily disabled by the malicious firmware.

As illustrated in ② Shared Memory of Fig. 5, the GPU MCU and CPU share a memory region for communication. This shared memory is mapped to the same physical memory in both the CPU and the MCU. On the MCU, the address is fixed at 0x4000000, while on the CPU, the address is dynamically allocated by the untrusted kernel driver. This shared memory is used to transfer metadata for GPU tasks between the CPU and the GPU. Note that GPU TEEs protect the shared memory using security primitives like CCA [14] during secure GPU tasks. However, as we will discuss in Sec. 4.3, the MCU is not constrained by these security primitives on the CPU side.

The MCU follows an event-driven execution model and is only active upon incoming events. As shown in ③ IRQ Channel of Fig. 5, the CPU can send IRQs to wake up the MCU to process GPU tasks. Similarly, the MCU can send an IRQ to CPU by writing to its IRQ controller located at the MMIO address 0x40000000. The MCU uses IRQs to notify the CPU of events like task completion. Moreover, our investigation shows that this MCU follows the same IRQ definition as Cortex-M [18] with 15 types of exceptions and 239 external interrupts. We will discuss how we leverage this IRQ mechanism to trigger malicious code in Sec. 4.4.

Implication 1. The GPU MCU is sufficiently sophisticated to execute arbitrary malicious code, potentially compromising the protection of GPU TEEs.

4.2 Tampering with Firmware

In this section, we investigate whether the untrusted kernel can tamper with the firmware of the GPU MCU and load malicious code. As Sec. 3.2 points out, to reduce the TCB, GPU TEEs rely on the untrusted kernel to initialize the GPU, including loading the firmware. To verify whether the untrusted kernel can indeed tamper with the firmware, we conducted the following experiment. In GPU TEEs, the firmware is loaded by the untrusted kernel from the `/lib/firmware` directory. With the knowledge from Sec. 4.1, we add a simple shellcode prior to the `reset` handler in the firmware binary to write a fixed value 0xdeadbeef to the shared memory between GPU and CPU. As the `reset` handler is executed once the MCU is booted up, we should observe the fixed value in the shared memory if the firmware is successfully tampered with. Then, we replace the original firmware with the patched one to initialize the GPU and observe 0xdeadbeef in the shared memory, indicating a successful tampering. Neither the GPU TEE nor other components (e.g., the GPU itself) implement any verification mechanism to detect firmware tampering. To further verify our results, we reviewed the source code of StrongBox [25], CAGE [63] and MyTEE [34], and found that none of them equips a verification mechanism for the GPU MCU firmware.

During the experiment, we observe that a hash value named `GIT_SHA` is hard-coded in the firmware, which is displayed in the kernel log. However, our investigation shows that this hash is neither used for firmware verification nor protected by any security primitives (e.g., secure storage or digital signature). Moreover, even if the hash were used for verification, the adversary can tamper with the firmware in a Time-of-Check-Time-of-Use (ToCToU) manner; we will discuss this in Sec. 9.2.2.

Implication 2. Due to the lack of a verification mechanism, the untrusted kernel can arbitrarily tamper with the firmware of the GPU MCU.

4.3 Bypassing Memory Protection

In this section, we aim to answer two key questions: 1) How can the secure memory be mapped into the address space of the MCU? 2) Can the GPU MCU bypass the security primitives deployed by GPU TEEs (e.g., SMMU, TZASC, Stage-2 MMU)? Though GPU TEEs also encrypt the sensitive data with cryptographic primitives like AES [52], the sensitive data must be decrypted for the GPU to process. As a result, the MCU can access sensitive data while the GPU is processing the decrypted plaintext.

In the first question, the main challenge is that the GPU MMU isolates the memory of the GPU MCU and other GPU tasks into different address spaces (see Sec. 4.1); the GPU MCU only has access to the metadata of the GPU tasks for scheduling purposes. Moreover, during secure GPU tasks, GPU TEEs like StrongBox prevent the untrusted kernel from further configuring the GPU MMU. This blocks an adversary from directly mapping the secure memory into the address space of the MCU. To circumvent this isolation, we design the following attack vector. When the untrusted kernel initializes the GPU, we insert a page into the page table hierarchy of the GPU MMU and also map this page into the address space of the MCU. Thus, by modifying the content of that page, the MCU can map the secure memory into its address space dynamically.

For the second question, we conduct experiments with the security primitives used in existing GPU TEEs, TZASC, Stage-2 translation and SMMU.[3] For the TZASC, we follow the setup in StrongBox and CAGE. On the GPU TEE side, we reserve a 1 MB secure memory region for the GPU TEE and fill it with 0xdeadbeef as the secret. We then configure TZASC to block any non-secure access to this region. As GPU must be able to access this region to process secure tasks, we configure the TZASC register to allow access with GPU's NSAID. Additionally, the GPU and CPU share a memory region at 0x4000000 for non-secure communication (see Sec. 4.1). We confirm that this setup aligns with existing GPU TEEs that use TZASC. On the MCU side, we use the above-mentioned attack vector to map the secure memory into MCU (at 0x4200000) dynamically and use the shellcode in Fig. 6 to access the secure memory. Then, similar to the shellcode insertion described in Sec. 4.2, we insert this piece of code before the `reset` exception handler, which will be executed once the MCU boots up. As shown in Fig. 6, this simple shellcode copies data from the secure memory (i.e., 0x4200000) to the non-secure shared memory (i.e., 0x4000000). We then read the non-secure memory to verify that the MCU successfully retrieved the secret value (i.e., 0xdeadbeef) under the protection of GPU TEEs.

In addition to the TZASC, we also conduct the same experiment with Stage-2 translation and SMMU. Specifically, we use the same setup as StrongBox, CAGE, and MyTEE. We configure the Stage-2 MMU to block any access from the untrusted world and the SMMU

[3]To our knowledge, only these security primitives are available on commercial hardware; Arm CCA has not yet been implemented in hardware; CAGE and ACAI use emulators or TrustZone to simulate CCA hardware.

```
ldr  r0, 0x4200000   // r0 = &secure
ldr  r1, [r0, 0]     // r1 = *r0
ldr  r2, 0x4000000   // r2 = &non_secure
str  r1, [r2, 0]     // *r2 = r1
```

Figure 6: Shellcode to access secure memory.

to block any access from untrusted peripherals. We repeat the above experiment and obtain the same result.

Implication 3. The GPU MCU can bypass the security primitives deployed by GPU TEEs and access the secure memory.

4.4 Triggering Malicious Code

In this section, we investigate how to trigger the malicious code using MCU's interrupts. As mentioned in Sec. 4.1, the interrupt handling is critical to the execution of the GPU MCU. Specifically, the GPU MCU follows an event-driven model, where the MCU is triggered by different events like task submission. As a result, though Sec. 4.2 and Sec. 4.3 have already demonstrated that the MCU can be exploited by adversaries to execute malicious code and even access secure memory, we still need to know which IRQ event can be used to trigger the malicious code.

```
ldr  r1, IRQ address    // r1 = IRQ address
ldr  r0, 0x4200000      // r0 = &share memory
str  r1, [r0, 0]        // *r0 = r1
```

Figure 7: Shellcode to explore IRQ handler.

To find the answer, we conduct the following experiments. According to the investigation in Sec. 4.1, the GPU MCU uses the same IRQ mechanism as a standard Cortex-M MCU [17], with 15 exceptions (e.g., `reset`) and 239 interrupts (e.g., `timer`), of which 15 IRQs are used by the MCU. We then prepend the shellcode in Fig. 7 to every IRQ handler with the techniques described in Sec. 4.2; this shellcode simply writes the address of the IRQ handler it prepends to the shared memory. We also build a user-space program to submit various GPU tasks to the GPU and observe which IRQ handler is triggered during the execution of the GPU tasks. As a result, we find that IRQ 2 is triggered when the GPU receives a new task, and IRQ 0 is triggered when the GPU completes the task and is ready to notify the CPU. Fig. 8 depicts this IRQ handling process. When the CPU sends a new task to the GPU, IRQ 2 is triggered on the MCU side, and the MCU then dispatches the task to the available compute unit. Similarly, when the compute unit completes the task, IRQ 0 is triggered on the MCU side, and the MCU then notifies the CPU to collect the result.

Implication 4. By compromising IRQ handlers, adversaries can trigger malicious code during critical events, such as secure task submission.

Conclusion. Given the security implications introduced by the GPU MCU, we conclude that this neglected component of the GPU

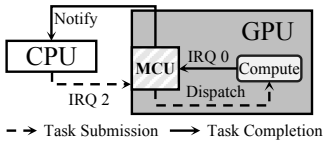


Figure 8: IRQ handling of GPU MCU.

poses a significant security threat to current GPU TEEs. In the following sections, we will present Mole, a novel attack against GPU TEEs, based on these findings.

5 Mole

In this section, we first revisit the design of a typical GPU TEE and analyze how Mole compromises its security properties. Then we describe the workflow of Mole.

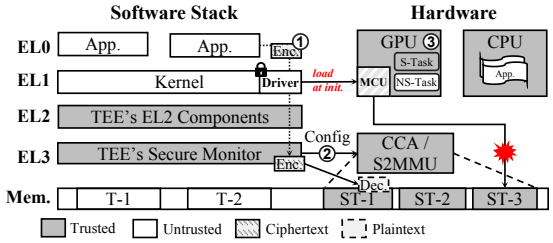


Figure 10: Revisited GPU TEE workflow.

5.1 Revisit GPU TEE

Fig. 10 depicts the workflow of a typical GPU TEE: ① A user application transmits an encrypted GPU task to the secure monitor in EL3. Though shim-style GPU TEEs rely on the untrusted driver to handle miscellaneous routines like task submission, the encryption ensures that the secure GPU task is protected. ② After the driver finishes configuring the task, the shim layer in secure monitor decrypts the encrypted data, configures the memory as protected to block CPU access and loads the plaintext into the memory for GPU to process. ③ The GPU begins to process these secure tasks and writes the results back to the protected memory.

This process ensures that the sensitive data is always protected by cryptographic encryption (e.g., AES) or memory protection primitives (e.g., CCA), preventing any untrusted entity from accessing the sensitive data. However, this design neglects the presence of GPU MCU, which is an embedded part of the trusted GPU but is initialized and configured by the untrusted kernel. As shown in Fig. 10, during the initialization of the GPU, the untrusted kernel is responsible for loading the firmware binary into the GPU MCU. Moreover, our experiment in Sec. 4 shows that there is no mechanism to prevent the untrusted kernel from loading a tampered firmware, and the MCU, as a trusted part of GPU TEE, can access the sensitive data of secure GPU tasks. Though GPU TEEs deploy security primitives like TZASC to prevent untrusted kernel from configuring GPU, these security primitives are only effective during the execution of secure GPU tasks (see Sec. 3.2). Therefore, the untrusted kernel can easily load a tampered MCU firmware during GPU initialization, allowing the trusted MCU to execute malicious code during the secure GPU tasks and compromise the protection.

In sum, though GPU TEEs are well-designed to prevent untrusted adversaries (e.g., the kernel) from accessing secure GPU tasks, it fails to account for the possibility that trusted components (e.g., GPU MCU) can be exploited to compromise the GPU TEE.

5.2 Mole Workflow

In this section, we introduce the workflow of Mole. Fig. 9 illustrates three major stages of Mole: 1) Firmware Initialization, 2) Data Exfiltration, and 3) Result Tampering.

Firmware Initialization. As shown in Fig. 9, Mole begins with the initialization of the malicious firmware. During the GPU initialization, the untrusted kernel loads a tampered firmware into the GPU MCU. Simultaneously, a shared memory region is also established between the GPU and CPU for communication, and a secure memory region is reserved for GPU TEE.

Data Exfiltration. One of the main attack scenarios of Mole is the exfiltration of sensitive data from secure GPU tasks. As illustrated by Fig. 9, the exfiltration process consists of three steps: ① The victim application submits a secure GPU task to the GPU for processing. Though the task memory is encrypted, its address is exposed to the untrusted GPU driver. ② The kernel places the victim address into the shared memory for the MCU. Note that the untrusted kernel cannot directly access this address. ③ Upon the task execution (i.e., when MCU receives IRQ 2, see Sec. 4.4), the malicious MCU inside GPU reads the secure memory at the victim address and stores the results into the shared memory. The untrusted kernel can then read the results from the shared memory.

Result Tampering. Similar to the data exfiltration, Mole can also tamper with the results of secure GPU tasks. As shown in Fig. 9, the result tampering process also consists of three steps: ① The kernel prepares the tampered result in the shared memory. ② Upon the task completion (i.e., when MCU receives IRQ 0, see Sec. 4.4), the malicious MCU overwrites the original result in the secure memory with the tampered one in the shared memory. ③ The user application reads the tampered result from the secure memory.

6 Implementation

The implementation of Mole consists of two main components: a Rust-based parser (940 lines of code) for interpreting and tampering with the closed-source MCU firmware, and a kernel module (1,400 lines of C code) for loading the malicious firmware. Below, we elaborate on the key aspects of the implementation.

Bypassing the MCU MPU. As discussed in Sec. 4.1, the MCU is equipped with a Memory Protection Unit (MPU) to restrict its memory access. Our analysis reveals that the MPU is configured to allow code execution only within the address range 0x800000 to 0x820000. Memory outside this range is marked with the NX (No-Execute) bit; any attempt to execute code from such memory triggers an access violation fault. This poses a significant challenge for Mole when attempting to inject arbitrary code into the MCU's memory. However, we discover that the MPU is configured by the MCU firmware itself, which means it can also be modified by the firmware. Fig. 11 illustrates the original MPU configuration code in the firmware. Specifically, the code configures two registers:

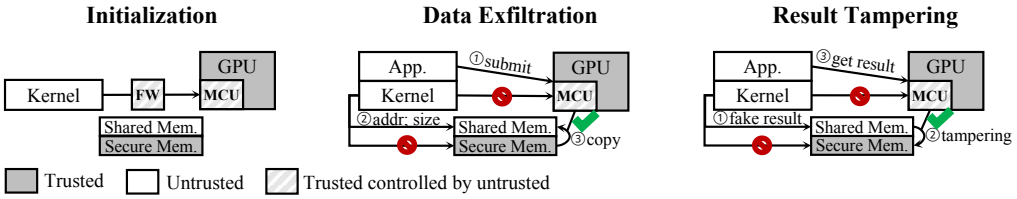


Figure 9: Mole workflow.

the MPU base register (MPU_RBAR) and the MPU range register (MPU_RASR). By modifying the value of MPU_RASR, whose bits[5:1] determine the size of the configured range (calculated as $2^{\text{bits}[5:1]+1}$), we are able to extend the executable memory region. By setting MPU_RASR to 0x50b0023, we increase the size of the executable region to 0x840000, providing ample space for our injected code.

```
ldr  r0, 0xe000ed9c // r0 = <MPU_RBAR>
ldr  r1, 0x800015   // base: 0x800000; valid: 1; id: 5
str  r1, [r0, 0]    // *r0 = r1
ldr  r0, 0xe000eda0 // r0 = <MPU_RASR>
ldr  r1, 0x50b0021  // size: 0x20000; NX: 0; valid: 1
str  r1, [r0, 0]    // *r0 = r1
```

Figure 11: MPU configuration code.

Shellcode Injection. With the MPU bypassed, we are able to inject custom shellcode into the MCU firmware. Fig. 12 illustrates the core component of the shellcode used to implement Mole.

```
ldr {r0, r1, r2}, {src_begin, src_end, dst_begin}
_loop:
ldr r3, [r0], 4  // tmp = *r0; r0 += 4
str r3, [r2], 4  // *r2 = tmp; r2 += 4
cmp r0, r1       // if r0 < r1
blt _loop2       // goto _loop
```

Figure 12: Mole-injected shellcode.

As shown in Fig. 12, the shellcode accepts three parameters from the untrusted memory provided by the privileged adversary: src_begin, src_end, and dst_begin. The shellcode performs a straightforward copy operation, transferring data from the source (src) to the destination (dst). In a data exfiltration scenario, the adversary can configure src to point to the memory region containing sensitive data and dst to the untrusted memory. Conversely, in a result tampering scenario, the adversary can prepare tampered data in the untrusted memory, set src to this location, and direct dst to the memory region where the result is stored.

7 Evaluation

We evaluate Mole by addressing the following research questions:

RQ1: How many real-world devices are equipped with the GPU MCU and vulnerable to Mole?

RQ2: How does Mole perform in real-world scenarios?

RQ3: How much overhead does Mole impose on the victim application?

7.1 RQ1: GPU MCU on Real-World Devices

Mole leverages the MCU embedded in the Arm Mali GPU to execute the attack. This MCU is included in all Arm Mali GPUs released after 2021. To assess the prevalence of such MCUs in real-world devices, we conducted a comprehensive survey of devices equipped with GPU MCUs. As shown in Table 4, Mali GPUs—one of the most widely used Arm GPU architectures—are extensively integrated into a variety of devices. For example, Google has adopted Arm Mali GPUs in all its mobile phones since the Pixel 6 series [32]. Similarly, MediaTek's System-on-Chip (SoC) designs incorporate Arm Mali GPUs, which are utilized by numerous mobile phone manufacturers, including Vivo, Redmi, Honor, and Huawei [36, 61, 68]. Additionally, Rockchip has integrated Mali GPUs into its latest IoT devices [56].

MCU in GPUs. While this work focuses on Arm Mali GPUs, we clarify that the MCU design is not exclusive to Arm Mali GPUs. For instance, NVIDIA incorporates an embedded RISC-V chip [9], and AMD utilizes a Micro Engine Scheduler (MES) chip [3]. However, hardware vendors like NVIDIA appear to enforce a "security through obscurity" practice by restricting technical disclosure regarding their MCU architectures. These vendors implement certain kinds of verification mechanisms to restrict the MCU access exclusively to their proprietary tools. Although we have not yet bypassed the verification implemented in these MCUs, we believe the findings of this work could potentially be generalized to other GPU architectures and provide valuable insights for future research.

RQ1 Ans. Arm Mali GPUs are utilized in a wide range of devices from various SoC and device vendors, including Huawei, Xiaomi, Vivo, and others. This widespread adoption underscores the broad applicability of Mole.

7.2 RQ2: Real-World Attack Scenarios

To evaluate the attack capabilities of Mole in real-world scenarios, we select in-GPU AES encryption and the widely-used object detection model YOLOv4-tiny as the victim applications. We select these cases because they are also used in recent research attacking TEEs [23, 69]. Technically, Mole directly transmits TEE-protected memory to the untrusted kernel, indicating a fluent extension to more complicated cases. The experiment setup aligns with the configuration described in Sec. 4.

Table 4: Real-world devices with Arm Mali GPUs.

SoC Vendor	SoC	Device Vendor	Usage	Representative Devices
HiSilicon	Kirin 8000	Huawei	Mobile Phone	Huawei Nova 12,13 Huawei Nova Flip
Rockchip	RK3588, RK3588s	Radxa, SinoVoip Forlinx, Rockchip	IoT Device	Rock Pi 5B, 5ITX, BPI-RK3588 TB-RK3588X, OK3588-C
Google	Tensor G2,G3,G4	Google	Mobile Phone Tablet	Pixel 6,7,8,9 Pixel Tablet 1,2
MediaTek	MT6886, MT6878, MT6895 MT6896, MT6983, MT6985 MT6989	Vivo, Redmi, Honor Oppo, Realme, Huawei OnePlus, Motorola Xiaomi	Mobile Phone Tablet	Vivo V27, S17e, V40e, Z9s, X90 Oppo A94, Reno 12, K10, FindX7 Redmi Note 13, 14, K60, K70 Realme 70, 13+, P1, Neo7 Motorola 50 series, OnePlus Ace 2V Xiaomi 14T Pro, Huawei Nova 8, Honor 70 Pro

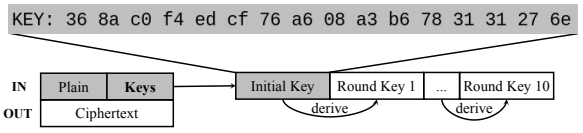


Figure 13: AES key extraction.

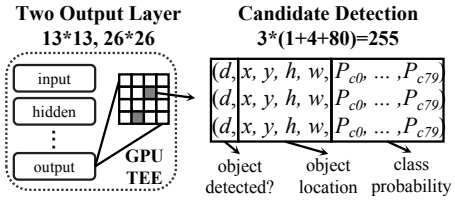


Figure 14: Output format of YOLOv4-tiny.

7.2.1 AES Key Extraction. As shown in Fig. 13, the GPU kernel for AES-128 encryption [1] accepts two input buffers: plaintext and keys and one output buffer: ciphertext. According to the AES standard [52], the key buffer stores a 16-byte initial key along with ten 16-byte round keys derived from the initial key. To compromise the AES encryption, we only need to use Mole to extract the initial key from the key buffer. We executed the AES-128-ECB algorithm to encrypt a 128 MB data buffer while simultaneously using Mole to exfiltrate the initial key from the key buffer. With Mole active, the AES encryption application completed the encryption in 843.3 ms, while Mole introduced an additional overhead of 0.03 ms. This overhead is negligible compared to the total encryption time.

7.2.2 Tampering with Object Detection. To tamper with the detection results of the YOLOv4-tiny model, we first analyze the model's output format. Subsequently, we prepare tampered results in the same format and use Mole to replace the original results, as described in Sec. 5.2. As shown in Fig. 14 and the output of the YOLOv4-tiny model consists of two layers, each containing 13×13 and 26×26 bounding boxes, respectively. Each bounding box produces three candidate vectors, each comprising 85 values (totaling 3×85=255 values). These values include the confidence score d, the coordinates of the detected object (x, y, h, w), and the probabilities P that the detected object belongs to c_n from P_{c_0} to $P_{c_{79}}$. Based on the quality of these candidate vectors, one is selected as the final output. Using this understanding, we prepare tampered results with dimensions of $(13 \times 13, 3 \times 85)$, and $(26 \times 26, 3 \times 85)$, ensuring they conform to the output format of YOLOv4-tiny.

We prepared three distinct types of attack scenarios by replacing the original outputs. Fig. 15 illustrates the effects of these attack vectors. Fig. 15a displays the original output of YOLOv4-tiny, which accurately detects three objects in the image: a person, a car, and a stop sign. In Fig. 15b, the detection results are tampered with by Mole, where the identities of the stop sign and the car are swapped. Fig. 15c demonstrates how Mole can remove the person and the stop sign from the detection results entirely. Finally, Fig. 15d shows a scenario where Mole relocates the detected person to an incorrect position. As shown in Fig. 15, Mole can easily tamper with the outputs of models like YOLOv4-tiny. Since such models are widely used in critical applications such as autonomous driving, we conclude that Mole poses a significant threat to real-world GPU applications. For instance, an attacker could remove the person and stop sign from the detection results, potentially causing regulatory violations or even fatal accidents in autonomous driving scenarios. In terms of performance overhead, the tampering process takes approximately 0.1 ms, which is negligible compared to YOLOv4-tiny's detection time of 5388.3 ms.

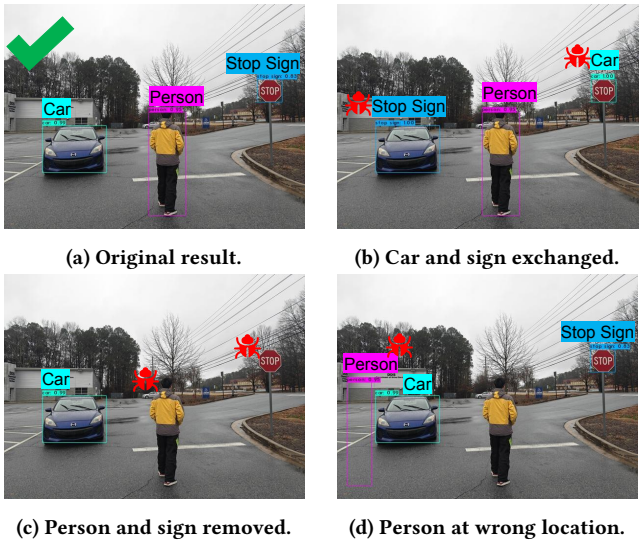
Clarification. We clarify that Mole's tampering is fully *deterministic* and *independent* of the input content. Thus, our attack demonstration here is not limited to YOLOv4-tiny. It is a general attack that is applicable to other object detection tasks and, more broadly, to other neural network-based applications.

RQ2 Ans. Mole is capable of extracting secret keys from in-GPU AES applications and tampering with the outputs of neural network models such as YOLOv4-tiny. These capabilities highlight the serious threat Mole poses to real-world GPU TEE applications.

7.3 RQ3: Mole Efficiency

The efficiency of Mole affects its stealthiness. If the execution time of the GPU task increases significantly due to Mole, the adversary risks detection. In this section, we evaluate Mole using benchmarks from StrongBox and CAGE [30, 31, 35, 37, 44].[4] We follow the same

[4]MyTEE focuses on secure I/O in general and does not include GPU-specific benchmarks.



(a) Original result. (b) Car and sign exchanged.

(c) Person and sign removed. (d) Person at wrong location.

Figure 15: Mole-tampered results of YOLOv4-tiny.

experimental setup as in RQ2 and Sec. 4. In all benchmarks, Mole is configured to exfiltrate all sensitive data sent to the GPU.

7.3.1 Rodinia Benchmark. We selected six test cases from the Rodinia benchmark [31]. Specifically, we chose one lightweight test case (K-nearest neighbors), three medium-weight test cases (LU decomposition, Pathfinder, and Hotspot3D), and two heavyweight test cases (Gaussian and LavaMD). This selection aligns with the setups used in StrongBox [25] and CAGE [63]. Table 5 presents the overhead introduced by Mole in terms of execution time and the size of exfiltrated data.

Table 5: Overhead of Mole on Rodinia benchmark.

		Time (ms)			
Test	**Size(MB)**	**GPU TEE**	**Mole**	**Slow.**	**Spd(MB/s)**
KNN	0.49	139.3	153.8	14.5	33.8
LUD	16.0	1929.2	2235.1	398.7	40.1
H3D	24.0	4161.8	4693.8	532.0	45.1
PF	38.1	1065.8	1873.4	807.6	47.2
GS	32.0	19914.0	20665.9	751.9	42.6
LMD	63.4	5318.6	6713.9	1395.3	45.4

From Table 5, Mole exfiltrates all sensitive data sent to the GPU at an average speed of 41.1 MB/s. In a real-world scenario, an attacker could exfiltrate sensitive data in multiple batches to further optimize the process. For instance, exfiltrating 1 MB of sensitive data with Mole takes approximately 20 ms. Additionally, as demonstrated in Sec. 7.2, real-world attacks typically target a small amount of data (e.g., cryptographic keys), which can be exfiltrated in a negligible time of less than 1 ms.

7.3.2 Neural Network. We also evaluate the overhead of Mole using three popular neural network models: LeNet-5, SqueezeNet, and MobileNet-v1 [35, 37, 44]. This setup aligns with the configurations used in StrongBox and CAGE. Additionally, we evaluate a more computationally intensive model, YOLOv4-tiny [30]. In these experiments, Mole is configured to exfiltrate all weights from the neural network models.

Table 6: Overhead of Mole on neural network models.

		Time (ms)			
Test	**Size(MB)**	**GPU TEE**	**Mole**	**Slow.**	**Spd(MB/s)**
LeNet-5	0.47	1833.9	1854.7	20.9	22.5
SqueezeNet	10.0	2425.9	2637.6	211.8	47.3
MobileNet-v1	16.2	8628.2	8976.5	348.4	46.6
YOLOv4-tiny	48.2	6295.8	7338.6	1042.8	46.2

Table 6 shows the overhead introduced by Mole in terms of execution time and the size of exfiltrated data. From the table, Mole incurs a maximum slowdown of approximately one second. Similar to the Rodinia benchmark, Mole exfiltrates sensitive data at an average speed of 40.5 MB/s. For transparently extracting model weights, the adversary can exfiltrate sensitive data in multiple batches to obtain all weights from the neural network models without severely affecting performance. For tampering with the output, as described in Sec. 7.2, the adversary only needs to intercept a small amount of data from the neural network model (e.g., detection results), which can be completed in a negligible time of less than 1 ms.

Intact Functionality. In addition to the performance overhead, we confirm that in all benchmarks, the functionality of the victim application remains *unaffected* by Mole. Furthermore, the sensitive data extracted by Mole precisely matches the data transferred between the victim application and the GPU.

RQ3 Ans. Mole can exfiltrate data from a GPU TEE at an average rate of over 40 MB/s without impacting the functionality of the victim application, making it sufficiently efficient for real-world attacks.

8 Related Works

In this section, we review other related works on GPU security. Note that we have already reviewed highly relevant works on Arm GPU TEEs in Sec. 3, where we analyzed their designs in detail.

GPU-based Attacks. Several prior works have examined the security of GPU applications. Lee et al. [46] discovered that both NVIDIA and AMD GPUs fail to initialize the newly allocated memory, potentially leaking data from previous users. A similar attack focuses on large language models [57], leveraging the leftover local memory in GPUs to leak sensitive information. Researchers have also focused on buffer overflow vulnerabilities. Miele [48] demonstrated that buffer overflow vulnerabilities in GPU could be exploited to overwrite function pointers and thereby subvert the control flow. A subsequent study [26] provided a detailed analysis on heap overflow vulnerabilities. Park et al. [54] presented the first attack on deep neural network, revealing that code pages on NVIDIA GPUs are writable. Most recently, Guo et al. [33] successfully located the return address on GPUs, demonstrating that return-oriented programming (ROP) attacks are feasible on GPUs.

Apart from memory corruption, there have also been studies exploring potential side-channel attacks on GPUs. Jiang et al. [41, 42, 43] presented a series of works that recover AES keys using GPU side-channels. Similarly, Trident [13] leveraged a cache-conflict side-channel to extract key from GPU implementations of AES. Wang et al. [65] exploited the timing side-channel of compression

algorithms used in GPU communication to recover data. Naghibijouybari et al. [50, 51] investigated the security risks of sharing a single GPU among multiple users. Though NVIDIA developed the vGPU technology [2] in response to these attacks, Zhang et al. [70] showed that the last-level TLB is still shared among different vGPUs, which could be used to establish covert channels between colluding entities.

The above discussion highlights that existing works mainly focus on memory corruption and side-channel attacks on GPU. To the best of our knowledge, Mole is the first work that discusses the security risks brought by the embedded MCU in modern GPUs.

Firmware Attack. Prior research has demonstrated the feasibility of conducting attacks through firmware manipulation. For instance, Cui et al. [24] exploited a remote firmware update mechanism to deploy malicious firmware on HP LaserJet printers. Similarly, iSeeYou [22] bypassed hardware safeguards to disable the camera indicator on Apple devices, enabling covert surveillance. More recently, FANDEMIC [60] introduced a firmware-based attack on power management circuits, which could cause permanent hardware damage. Additionally, Ibrahim et al. [38] and Wu et al. [67] conducted comprehensive analyzes of IoT security vulnerabilities, highlighting weaknesses in firmware update mechanisms across various devices and their companion applications.

GPU-oriented Defenses. In addition to GPU TEEs designed for integrated GPUs on Arm platforms, similar efforts have been made for dedicated GPUs from NVIDIA and AMD [39, 62, 72]. Volos et al. [62] proposed Graviton, a GPU TEE that incorporates customized hardware to secure the communication between the GPU and the CPU. HETEE [72] adopts a similar approach to secure the GPU-CPU communication over the PCIe bus. Jang et al. [39] explored the possibility of re-purposing the SGX technology to secure GPU memory. These works primarily focus on securing the plaintext *communication* between the GPU and the CPU to defend against privileged adversaries *outside* the GPU itself. In contrast, Mole exploits the MCU *inside* the GPU to launch attacks, conceptually bypassing the defenses proposed in these works. Given that the MCU design is widely adopted in modern GPUs [3, 9], Mole could potentially be extended to dedicated GPUs from NVIDIA and AMD.

In addition to GPU TEEs, a series of works aim to enhance the memory safety of GPU applications [27, 28, 45]. ClArmor and GMOD [27, 28] introduced a Canary-like mechanism on GPU to prevent stack corruption. Lee et al. [45] implement a bounds-checking mechanism on GPUs using customized hardware.

9 Discussion

In this section, we discuss the potential extensions and countermeasures of Mole.

9.1 Potential Mole Extension

There are several potential extensions of Mole, including attacks involving an unprivileged adversary and supply chain poisoning.

Attack with Unprivileged Adversary. In the current design of Mole, we assume a high-privileged adversary, such as the OS kernel, which aligns the threat model of GPU TEEs. We now explore the attack model in the context of an unprivileged adversary. To date, there are 47 CVEs associated with the Arm Mali GPU driver, indicating that an attacker could use these vulnerabilities to overwrite the GPU MCU firmware [7] and launch Mole. Though the MCU is restricted by the GPU's MMU and SMMU, which limits its access to arbitrary system memory, the MCU could tamper with the GPU computation with its current capability (e.g., task scheduling) or cause system freezes by launching an IRQ flooding attack.

Supply Chain Poisoning. During the development of Mole, we observed that the firmware of the GPU MCU is distributed in a binary format. Furthermore, our investigation in Sec. 4.2 reveals that the firmware lacks verification mechanism. The absence of verification implies that an attacker could forge firmware and distribute it along the supply chain. To understand the potential impact of this threat, we investigate the distribution channels of major vendors for the GPU firmware; Table 7 summarizes our findings.

Table 7: Distribution channels of MCU firmware; Bundled means the firmware is bundled with pre-installed software.

Vendor	Usage	Distribute via	Verified	#Ref.
Google	Mobile Phone Tablet	Bundled	✘	0
MediaTek	Mobile Phone Tablet	Bundled GitHub	✘	2
HiSilicon (Huawei)	Mobile Phone	Bundled	✘	0
Rockchip	IoT Device	GitHub GitLab	✘	**41**

From Table 7, we observe that the firmware of the GPU MCU is primarily distributed in two ways: either bundled with pre-installed software or hosted on a public code repository. As shown in Table 7, most vendors opt to bundle the firmware directly with the software, which generally ensures its integrity. However, MediaTek and Rockchip also host their firmware on public code repositories such as GitHub [8, 10]. Notably, Rockchip uses GitHub/GitLab as its primary distribution channel for its firmware, resulting in 41 downstream references to the firmware repository [10]. Moreover, when we contacted MediaTek, they denied that the firmware distributed on GitHub originates from them, despite its misleading naming.[5] This highlights the challenges of maintaining a healthy and transparent supply chain for firmware distribution. To make matters worse, as shown in Table 7, neither vendors nor Arm provides an authoritative way to verify the integrity of the firmware. The lack of verification could potentially introduce the risk of supply chain poisoning, enabling an attacker to forge firmware and distribute it through the public repository with a deceptively similar name.

9.2 Countermeasures

9.2.1 Secure Firmware Management. As discussed in Sec. 3, Mole is not applicable to VM-style GPU TEEs because the firmware management in such systems is handled by the trusted GPU driver bundled within the VM. Shim-style GPU TEEs could also adopt a similar design by relocating the firmware management to the trusted shim layer and restricting untrusted kernel's access to the memory region containing GPU firmware. By doing so, the adversary would

[5] By the time of writing, we are still responsibly discussing with MediaTek and wishing to address it without causing any security breaches.

no longer have access to the firmware, thereby preventing them from tampering with it to launch Mole.

Limitation. However, moving the firmware management into the trusted shim layer is not a trivial task. The GPU driver on Arm contains nearly 250 KLoC [6] and the firmware-related code is scattered across the driver, comprising over 60 KLoC. Separating this code from the GPU driver is therefore a daunting task (if at all possible). Moreover, deploying this countermeasure also faces the problem of bloating the TCB of the GPU TEEs. Existing shim-style GPU TEEs (e.g., CAGE) typically have a small TCB of less than 5 KLoC, which is easier to audit and verify its correctness. By contrast, the firmware-related code in the GPU driver is significantly larger (over 60 KLoC). Worse still, codebases of this size are more prone to vulnerabilities. For instance 47 CVEs have been identified in the Arm Mali GPU driver [7] between 2021 and 2024, making it an unsuitable candidate for the TCB of GPU TEEs.

9.2.2 Firmware Verification. Since Mole operates by running a tampered firmware on the GPU MCU, an intuitive countermeasure is to verify the authenticity of the firmware. As shown in Fig. 16, upstream vendors like Arm could provide a cryptographic signature for the firmware. Then, a verifier could then be implemented to validate the signature before loading the firmware, ensuring its authenticity. To guarantee the security of the verification process, the verifier should be implemented in the GPU TEE. Additionally, the adversaries might attempt to tamper with the firmware *after* the verification. To prevent such ToCToU attacks, GPU TEE must ensure that the memory region containing the firmware is protected from untrusted components. In this case, the firmware management routines are included in the TCB, as the untrusted components, such as the OS kernel, are no longer allowed to access the firmware.

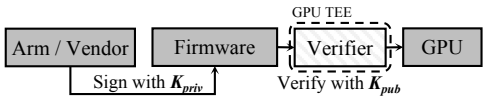


Figure 16: Firmware verification process.

Limitation. The main drawback of this countermeasure is similar to that discussed in Sec. 9.2.1. To prevent ToCToU attacks, the untrusted components such as the OS kernel must be restricted from accessing the firmware. Consequently, the GPU TEE must assume the responsibility for firmware management, including tasks like reloading the firmware or updating the firmware configurations. This approach could significantly increase the TCB size of the GPU TEE, thereby introducing additional vulnerabilities.

9.2.3 MCU-level Isolation. Another potential countermeasure involves partitioning the MCU into two isolated modes: a Non-secure MCU (NS-MCU) for handling normal GPU tasks from untrusted users and a Secure MCU (S-MCU) for managing secure tasks from the GPU TEE. Similar to TrustZone, each mode has its own memory space and execution context. In this design, only the S-MCU is permitted to access the pages allocated to the secure GPU Tasks, and only trusted components (e.g., the GPU TEE) are allowed to configure S-MCU. This approach extends the isolation boundary to the MCU level, preventing MCU from compromising GPU TEE.

Limitation. To begin with, this approach requires non-trivial hardware modifications, introducing additional costs. Furthermore, implementing the isolation between NS-MCU and S-MCU is not straightforward. This is because the MCU is responsible for dispatching the tasks among various components within the GPU (e.g., compute units and render units). If the MCU is partitioned into two separated modes, the dispatching mechanism would face challenges in synchronizing the GPU status between the secure mode and the non-secure mode. This could lead to performance degradation or even introduce new security vulnerabilities.

10 Responsible Disclosure

We have reported our findings to the authors of StrongBox [25], CAGE [63] and MyTEE [34]. They have all acknowledged the threat posed by Mole and confirm that additional countermeasures are necessary to defend against it [11]. We have also reported the potential supply chain risks of the GPU firmware to Arm, MediaTek, and Rockchip. Arm acknowledged the potential supply-chain contamination and promised that a signature would be released for firmware verification. MediaTek and Rockchip acknowledged that Arm's effort benefits users who download firmware from potentially untrusted sources (e.g., GitHub).

11 Conclusion

We introduce a novel attack scheme named Mole, which undermines the security guarantees provided by state-of-the-art GPU TEEs. By loading a tampered firmware into the GPU MCU, Mole can extract sensitive data or manipulate the results of the TAs protected by GPU TEEs. Our evaluation demonstrates that Mole is virtually undetectable to end-users due to its minimal latency. Finally, we reported our findings to the related parties and have received acknowledgements from them. We hope that Mole will raise awareness of these hidden MCUs. We also advocate that future TEE proposals take these components into account.

Acknowledgements

We would like to thank the anonymous reviewers, Dr. Adrian Rowland and the members of COMPASS lab for their valuable feedback. This work is partly supported by the National Natural Science Foundation of China under Grant No. 62372218 and No. U24A6009. This work was also in part supported by Ant Group. HKUST authors are supported in part by a RGC CRF grant under the contract C6015-23G.

References

[1] 2018. opencl-aes. https://github.com/zhoukaidev/opencl-aes.git
[2] 2020. NVIDIA Multi-Instance GPU (MIG) User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/
[3] 2024. AMD Micro Engine Scheduler. https://gpuopen.com/download/documentation/micro_engine_scheduler.pdf
[4] 2024. Android Open Source Project GPU Firmware. https://android.googlesource.com/device/google/coral/refs/heads/master/firmware/g610/
[5] 2024. Linux GPU MCU Firmware. https://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git/tree/arm/mali/arch10.8
[6] 2024. Mali GPU Driver. https://developer.arm.com/downloads/-/mali-drivers/valhall-kernel
[7] 2024. malicve. https://developer.arm.com/Arm%20Security%20Center/Mali%20GPU%20Driver%20Vulnerabilities
[8] 2024. MediaTek GPU MCU Firmware on GitHub. https://github.com/hailongfan-mtk/linux_fw_sof/tree/main/arm/mali/arch10.8

[9] 2024. *NVIDIA RISC-V Story*. https://riscv.org/wp-content/uploads/2024/12/Tue1100_Nvidia_RISCV_Story_V2.pdf

[10] 2024. Rockchip GPU MCU Firmware on GitHub. https://github.com/JeffyCN/mirrors/tree/libmali/firmware/g610

[11] 2025. MOLE Website. https://sites.google.com/view/mole-gpu

[12] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[13] Jaeguk Ahn, Cheolgyu Jin, Jiho Kim, Minsoo Rhu, Yunsi Fei, David Kaeli, and John Kim. 2021. Trident: A Hybrid Correlation-Collision GPU Cache Timing Attack for AES Key Recovery. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 332–344. https://doi.org/10.1109/HPCA51647.2021.00036

[14] Arm. 2024. Arm Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture

[15] Arm. 2024. Armv7-M Architecture. https://developer.arm.com/documentation/ddi0403/latest/

[16] Arm. 2024. Armv7-M Memory Protection Unit. https://developer.arm.com/documentation/101407/latest/Debugging/Debug-Windows-and-Dialogs/Core-Peripherals/Armv7-M-cores/Armv7-M--Memory-Protection-Unit

[17] Arm. 2024. Exception types. https://developer.arm.com/documentation/dui0497/a/the-cortex-m0-processor/exception-model/exception-types

[18] Arm. 2024. M-Profile Architectures. https://www.arm.com/architecture/cpu/m-profile

[19] Arm. 2024. Stage 2 translation. https://developer.arm.com/documentation/102142/0100/Stage-2-translation

[20] Arm. 2024. TrustZone for Cortex-A. https://www.arm.com/technologies/trustzone-for-cortex-a

[21] Boris Brezillon. 2024. PanCSF: A new DRM driver for Mali CSF-based GPUs. https://www.collabora.com/news-and-blog/news-and-events/pancsf-a-new-drm-driver-for-mali-csf-based-gpus.html

[22] Matthew Brocker and Stephen Checkoway. 2014. {iSeeYou}: Disabling the {MacBook} webcam indicator {LED}. In *23rd USENIX Security Symposium (USENIX Security 14)*. 337–352.

[23] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[24] Ang Cui, Michael Costello, and Salvatore J Stolfo. 2013. When Firmware Modifications Attack: A Case Study of Embedded Exploitation.. In *NDSS*, Vol. 1. 1–1.

[25] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, et al. 2022. Strongbox: A gpu tee on arm endpoints. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 769–783.

[26] Bang Di, Jianhua Sun, and Hao Chen. 2016. A Study of Overflow Vulnerabilities on GPUs. In *Network and Parallel Computing: 13th IFIP WG 10.3 International Conference, NPC 2016, Xi'an, China, October 28-29, 2016, Proceedings* (Xi'an, China). Springer-Verlag, Berlin, Heidelberg, 103–115. https://doi.org/10.1007/978-3-319-47099-3_9

[27] Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. 2018. GMOD: A dynamic GPU memory overflow detector. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 1–13.

[28] Christopher Erb, Mike Collins, and Joseph L Greathouse. 2017. Dynamic buffer overflow detection for GPGPUs. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 61–73.

[29] Tianhao Fu, Zehua Yang, Zhisheng Ye, Chenxiang Ma, Yang Han, Yingwei Luo, Xiaolin Wang, and Zhenlin Wan. 2023. A Survey on the Scheduling of DL and LLM Training Jobs in GPU Clusters. *Chinese Journal of Electronics* 34, 3 (2023), 801–905.

[30] Github. 2024. darknet. https://github.com/AlexeyAB/darknet

[31] Github. 2024. gpu-rodinia. https://github.com/yuhc/gpu-rodinia

[32] Google. 2024. Pixel 8. https://store.google.com/us/product/pixel_8?hl=en-US

[33] Yanan Guo, Zhenkai Zhang, and Jun Yang. 2024. GPU Memory Exploitation for Fun and Profit. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 4033–4050. https://www.usenix.org/conference/usenixsecurity24/presentation/guo-yanan

[34] Seung-Kyun Han and Jinsoo Jang. 2023. MyTEE: Own the Trusted Execution Environment on Embedded Devices.. In *NDSS*.

[35] Andrew G Howard. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[36] Huawei. 2024. Huawei Website. https://consumer.huawei.com/en/

[37] Forrest N Iandola. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[38] Muhammad Ibrahim, Andrea Continella, and Antonio Bianchi. 2023. Aot-attack on things: A security analysis of iot firmware updates. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS& P)*. IEEE, 1047–1064.

[39] 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.

[40] Jianyu Jiang, Ji Qi, Tianxiang Shen, Xusheng Chen, Shixiong Zhao, Sen Wang, Li Chen, Gong Zhang, Xiapu Luo, and Heming Cui. 2022. CRONUS: Fault-isolated, secure and high-performance heterogeneous computing for trusted execution environment. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 124–143.

[41] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2016. A complete key recovery timing attack on a GPU. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 394–405. https://doi.org/10.1109/HPCA.2016.7446081

[42] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2017. A Novel Side-Channel Timing Attack on GPUs. In *Proceedings of the Great Lakes Symposium on VLSI 2017* (Banff, Alberta, Canada) *(GLSVLSI '17)*. Association for Computing Machinery, New York, NY, USA, 167–172. https://doi.org/10.1145/3060403.3060462

[43] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2019. Exploiting Bank Conflict-based Side-channel Timing Leakage of GPUs. *ACM Trans. Archit. Code Optim.* 16, 4, Article 42 (Nov. 2019), 24 pages. https://doi.org/10.1145/3361870

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

[45] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. 2022. Securing GPU via region-based bounds checking. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 27–41. https://doi.org/10.1145/3470496.3527420

[46] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *2014 IEEE Symposium on Security and Privacy*. 19–33. https://doi.org/10.1109/SP.2014.9

[47] Shih-Wei Li, John S Koh, and Jason Nieh. 2019. Protecting cloud virtual machines from hypervisor and host operating system exploits. In *28th USENIX Security Symposium (USENIX Security 19)*. 1357–1374.

[48] Andrea Miele. 2015. Buffer overflow vulnerabilities in CUDA: a preliminary analysis. arXiv:1506.08546 [cs.CR] https://arxiv.org/abs/1506.08546

[49] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1466–1482.

[50] Hoda Naghibijouybari, Khaled N Khasawneh, and Nael Abu-Ghazaleh. 2017. Constructing and characterizing covert channels on gpgpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 354–366.

[51] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2139–2153.

[52] NIST. 2001. Advanced Encryption Standard (AES). https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf

[53] Heejin Park and Felix Xiaozhu Lin. 2023. Safe and Practical GPU Computation in TrustZone. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 505–520.

[54] Sang-Ok Park, Ohmin Kwon, Yonggon Kim, Sang Kil Cha, and Hyunsoo Yoon. 2021. Mind control attack: Undermining deep learning with GPU memory exploitation. *Computers & Security* 102 (2021), 102115. https://doi.org/10.1016/j.cose.2020.102115

[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA.

[56] Radxa. 2024. Radxa ROCK 5 ITX. https://radxa.com/products/rock5/5itx/

[57] Tyler Sorensen and Heidy Khlaaf. 2024. LeftoverLocals: Listening to LLM Responses Through Leaked GPU Local Memory. arXiv:2401.16603 [cs.CR] https://arxiv.org/abs/2401.16603

[58] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, Mark Kuhne, Fabio Aliberti, and Shweta Shinde. 2024. {ACAI}: Protecting Accelerator Execution with Arm Confidential Computing Architecture. In *33rd USENIX Security Symposium (USENIX Security 24)*. 3423–3440.

[59] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2017. {CLKSCREW}: Exposing the perils of {Security-Oblivious} energy management. In *26th USENIX Security Symposium (USENIX Security 17)*. 1057–1074.

[60] Ryan Tsang, Doreen Joseph, Qiushi Wu, Soheil Salehi, Nadir Carreon, Prasant Mohapatra, and Houman Homayoun. 2022. FANDEMIC: Firmware Attack Construction and Deployment on Power Management Integrated Circuit and Impacts on IoT Applications.. In *NDSS*.

[61] Vivo. 2024. Vivo Website. https://www.vivo.com/en

[62] 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.

[63] Chenxu Wang, Fengwei Zhang, Yunjie Deng, Kevin Leach, Jiannong Cao, Zhenyu Ning, Shoumeng Yan, and Zhengyu He. 2024. CAGE: Complementing Arm CCA with GPU Extensions. In *Network and Distributed System Security (NDSS) Symposium*.

[64] Li WANG, Xuewei WU, Yanhui WANG, Zhe XIAO, Liang LI, and Aiguo FEI. 2023. On UAV Serving Node Deployment for Temporary Coverage in Forest Environment: A Hierarchical Deep Reinforcement Learning Approach. *Chinese Journal of Electronics* 32, 4 (2023), 760–772.

[65] Yingchen Wang, Riccardo Paccagnella, Zhao Gang, Willy R. Vasquez, David Kohlbrenner, Hovav Shacham, and Christopher W. Fletcher. 2024. GPU.zip: On the Side-Channel Implications of Hardware-Based Graphical Data Compression . In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 3716–3734. https://doi.org/10.1109/SP54263.2024.00084

[66] Xiaolong Wu, Dave Jing Tian, and Chung Hwan Kim. 2023. Building GPU tees using CPU secure enclaves with gevisor. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. 249–264.

[67] Yuhao Wu, Jinwen Wang, Yujie Wang, Shixuan Zhai, Zihan Li, Yi He, Kun Sun, Qi Li, and Ning Zhang. 2024. Your firmware has arrived: A study of firmware update vulnerabilities. In *33rd USENIX Security Symposium (USENIX Security 24)*. 5627–5644.

[68] Xiaomi. 2024. Xiaomi Website. https://www.mi.com/global/

[69] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. 2025. CipherSteal: Stealing Input Data from TEE-Shielded Neural Networks with Ciphertext Side Channels . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 79–79. https://doi.org/10.1109/SP61157.2025.00079

[70] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. 2023. TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 960–974. https://doi.org/10.1145/3576915.3616672

[71] Zongwei Zhou, Miao Yu, and Virgil D Gligor. 2014. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *2014 IEEE symposium on security and privacy*. IEEE, 308–323.

[72] 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.