Secure and Timely GPU Execution in Cyber-physical Systems

Jinwen Wang
Washington University in St. Louis
St. Louis, USA
jinwen.wang@wustl.edu

Yujie Wang
Washington University in St. Louis
St. Louis, USA
jay.wang@wustl.edu

Ning Zhang
Washington University in St. Louis
St. Louis, USA
zhang.ning@wustl.edu

ABSTRACT

Graphics Processing Units (GPU) are increasingly deployed on Cyber-physical Systems (CPSs), frequently used to perform real-time safety-critical functions, such as object detection on autonomous vehicles. As a result, availability is important for GPU tasks in CPS platforms. However, existing Trusted Execution Environments (TEE) solutions with availability guarantees focus only on CPU computing.

To bridge this gap, we propose AvaGPU, a TEE that guarantees real-time availability for CPU tasks involving GPU execution under compromised OS. There are three technical challenges. First, to prevent malicious resource contention due to separate scheduling of CPU and GPU tasks, we proposed a CPU-GPU co-scheduling framework that couples the priority of CPU and GPU tasks. Second, we propose software-based secure preemption on GPU tasks to bound the degree of priority inversion on GPU. Third, we propose a new split design of GPU driver with minimized Trusted Computing Base (TCB) to achieve secure and efficient GPU management for CPS. We implement a prototype of AvaGPU on the Jetson AGX Orin platform. The system is evaluated on benchmark, synthetic tasks, and real-world applications with 15.87% runtime overhead on average.

CCS CONCEPTS

• **Security and privacy → Trusted computing**.

KEYWORDS

GPU; Cyber-physical System; System Security; Availability

ACM Reference Format:
Jinwen Wang, Yujie Wang, and Ning Zhang. 2023. Secure and Timely GPU Execution in Cyber-physical Systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623197

1 INTRODUCTION

GPU plays an increasingly important role in real-time CPSs [1, 3, 15] as more and more AI components are integrated into safety-critical CPS, such as self-driving [4, 8, 14]. Under the increasingly rich features, the software system becomes incredibly complex, making it extremely challenging, if not impossible, to be vulnerable-free [9]. The exploitation of these vulnerabilities allows attackers to tamper with or simply deny key safety-critical functionalities in CPS, such as pedestrian detection.

GPU Execution Protection in CPS: Existing secure execution solutions [27, 32, 33, 42, 43, 48, 59, 67] for GPU mostly focus on the assurance of confidentiality and integrity. These GPU TEE solutions can be categorized into two approaches: leveraging CPU-TEE to secure GPU access, as demonstrated in [27, 33, 42, 43, 48], or directly instantiating a TEE on GPU [32, 59, 67]. CPU-based TEEs often result in larger TCB, while GPU-TEE typically require hardware modifications. However, for GPU protection in CPS, the availability (timeliness) is also an essential aspect.

While there has been some recent work on TEE with availability assurance [22, 61], they primarily considered systems with CPU executions only and cannot be directly applied to GPU. Accelerators, such as GPU, has its own resource management mechanisms and structures as an additional computational unit. As a result, availability of GPU execution requires a holistic consideration of both CPU and GPU resource management. Given the prevalence of AI in modern CPS for safety critical functions, such as perception and control, it is essential for availability assurance solutions to also support accelerators, such as GPU.

Secure and Timely GPU Execution with AvaGPU: To bridge this gap, we introduce AvaGPU, a TEE designed to provide real-time availability guarantees for CPU tasks involving GPU execution on CPSs in the presence of untrusted OS. To achieve this objective, AvaGPU needs to address several challenges unique to GPU and beyond existing availability solutions [22, 61]:

C1. CPU-GPU Task Priority Coupling: As an independent computing unit, the GPU contains its own task scheduler in the driver. The separation between CPU and GPU schedulers introduces priority decoupling between CPU and GPU tasks. Specifically, when the GPU scheduler allocates computational resources, like GPU processing units, to the submitted GPU tasks, it is unaware of the priority of the CPU tasks associated with the GPU tasks. Thus, GPU tasks from high priority CPU tasks, like safety-critical secure tasks, can be delayed by other GPU tasks submitted by lower priority CPU tasks [35, 46], resulting in priority inversion. Therefore, coupling the priorities of both CPU and GPU tasks is necessary to ensure timely completion of GPU-involved secure tasks. However, neither existing GPU secure execution solutions [27, 33, 59] nor CPU availability solutions [22, 61] enforces priority coupling between CPU and GPU tasks. To solve this problem, AvaGPU introduces a real-time CPU-GPU co-scheduling framework in TEE. This framework couples the priority of CPU and GPU tasks by prioritizing secure GPU tasks during the execution period of corresponding secure CPU tasks.

C2. Secure Preemptive Scheduling: Preemptive scheduling is crucial in mitigating priority inversion in real-time systems, by enabling higher priority tasks to interrupt lower priority tasks. However, mainstream GPUs lack hardware-level support or public APIs for task preemption [30]. Existing software-based GPU task preemption solutions can be categorized as two approaches. 1) Wait-based inter-thread preemption [24, 25, 63, 66] often incurs long preemption delay, hindering the real-time responsiveness of the system. 2) Re-execution-based methods are efficient but only work for idempotent workload [30, 37], cannot be applied on CPS workloads where states are crucial. To tackle this challenge, AvaGPU proposes to statically instrument both secure and non-secure GPU task codes to add self-suspending capability according to preemption signal (a software-based flag) from GPU scheduler.

However, non-secure GPU tasks submitted from the untrusted Rich Execution Environment (REE) can have their instrumentation-based self-suspending capability disabled in two ways. First, REE attackers can remove the self-suspending instrumentation. Second, it's difficult to verify that a non-secure GPU task is free of vulnerability. Thus, a REE attacker can hijack control flow [39, 44] in non-secure GPU tasks to bypass self-suspending instrumentation at runtime. Both approaches lead to non-secure GPU tasks executing without any suspensions, introducing delay on secure GPU tasks execution by contenting computing resources. AvaGPU proposes two defense mechanisms to prevent/eliminate these attacks as early as possible. First, AvaGPU only executes non-secure GPU tasks with correctly verified cryptographic signatures, preventing the removal of self-suspending instrumentation before execution. Second, AvaGPU detects and eliminates the self-suspending instrumentation bypassing at runtime. Specifically, AvaGPU monitors the secure GPU progress. If secure GPU tasks don't make expected progress in a limited time period. AvaGPU proactively kills unresponsive GPU tasks. However, the progress checkpoints must be selected carefully to defend against instrumentation bypassing attack without introducing high runtime overhead. While more frequent secure GPU tasks progress monitoring can detect attacks quicker, it also introduces higher runtime overhead. To minimize the attack detection runtime overhead while eliminating self-suspending bypassing attack in time for the guarantee of application's real-time performance, AvaGPU formulates the trade-off between security and runtime overhead as a constraint optimization problem to solve it.

C3. Secure GPU Management: In order to guarantee availability for GPU tasks, it is necessary to leverage hardware resource isolation to control the access of GPU resource from untrusted domains. This is usually accomplished by assigning the GPU to secure domain as a secure device. However, this poses two unique challenges. The first challenge is due to the execution of non-secure GPU code on the secure device (GPU). Existing hardware-enforced memory access control in ARM TrustZone works on the granularity of hardware device, and as a result, non-secure GPU tasks on the secure GPU can access all the secure resources. AvaGPU prevents this attack by deploying a DMA reference monitor in TEE to validate DMA memory access in GPU commands, preventing malicious modification of secure memory. The second challenge is due to the sharing of GPU between secure and non-secure CPU tasks, requiring mechanisms to harmonize the non-secure GPU driver in the REE and the secure GPU driver in the TEE. To minimize the impact on TCB, AvaGPU leverages CPS predictability to create a template driver [61]. However, different from regular I/O devices, there is complex resource management to enable GPU task execution, and trapping to TEE for every management operation significantly slows down the system, violating the real-time requirement for the CPS. To minimize this, AvaGPU proposes GPU management delegation, separate command buffers, and batch command buffer synchronization mechanisms based on Stage-2 memory access control.

Prototype and Evaluation: We implemented the prototype of AvaGPU on the Jetson AGX Orin platform. Two cases are used to demonstrate the effectiveness of defense against availability attacks, including malicious GPU frequency reduction attack and preemption bypassing attack. To evaluate the system performance of AvaGPU, we measure the performance overhead on Rodinia [19] GPU and SPECrate 2017 [20] CPU benchmark suite. The real-time performance of AvaGPU is evaluated on both synthetic real-time GPU tasks and real-world applications. In summary, we make following contributions:

- We design and implement a software-based real-time TEE solution for tasks involving both CPU and GPU execution, ensuring real-time secure GPU tasks finish correctly and timely in the presence of a compromised OS.
- To address priority inversion, we propose a secure real-time CPU-GPU co-scheduling mechanism and a fine-grained GPU task preemption mechanism to ensure real-time responsiveness. A secure GPU management system is also developed to take advantage of CPS predictability to isolate GPU resources with minimized overhead in system runtime and TCB.
- We implement a prototype of AvaGPU and show the proposed system can defend against availability attacks with case studies. We also evaluate the system performance on Rodinia and SPECrate 2017 benchmarks, synthetic real-time tasks, as well as real-world applications.

2 BACKGROUND

2.1 Graphics Processing Unit (GPU)

Hardware: GPUs, either dedicated or integrated, are classified based on whether they share physical memory with the CPU. CPSs typically employ integrated GPUs that share memory with the CPU because of Size, Weight, and Power (SWaP) limitations. Thus, AvaGPU focuses on integrated GPU. We use Nvidia GPU as an example. The CPU communicates with the GPU through access to the GPU-exposed MMIO memory space. A GPU primarily consists of copy (DMA) engine, command processor, computation unit, and memory controller. The copy engine transfers data between host device and GPU memory spaces, while the command processor receives commands from the GPU driver and dispatches them to the GPU computation unit. This unit features Graph Processing Clusters (GPCs) sharing an L2 cache, with each GPC containing multiple Streaming Multiprocessors (SMs). Each SM includes several cores that share an L1 cache. GPU uses different mechanisms to isolate cache between tasks. Specifically, L2 cache memory is indexed using physical addresses, while L1 cache uses virtual addresses for indexing. Thus, L1 caches are flushed during a context switch. A GPU task determines the number of threads used, organizing them as thread blocks divided into warps. The hardware scheduler utilizes warps as the scheduling unit for each SM. Modern mainstream GPUs from leading manufacturers, such as Nvidia [59], AMD [51], Intel [57], and Arm [27], utilize virtual memory to isolate memory space among various GPU tasks. This is accomplished through a separate Memory Management Unit (MMU) that employs page table walkers for address translation and a hierarchy Translation Lookaside Buffers (TLBs).

Software Stack: The GPU software stack primarily consists of a user-space runtime library (e.g., CUDA and OpenCL) and a kernel-space GPU driver. The user-space runtime library offers APIs for user-space applications (i.e., GPU tasks) to program the GPU execution unit with codes and to transfer data between the host device buffer and GPU buffer. These API calls are converted into GPU commands, which configure the GPU and control data transfers and task launches. The kernel-space GPU driver mainly responds to the GPU management, like memory management and GPU command submission. Each GPU task executes in a separate virtual memory space, ensuring memory isolation between tasks. This is achieved by allocating multi-level page tables in GPU driver for each GPU task before task loading. A command processor in the GPU fetches commands from host devices, with the GPU driver managing two buffers: a command buffer and a ring buffer. The runtime library places commands into the command buffer, which is memory-mapped to the user space. The GPU's command processor utilizes a ring buffer to fetch commands. Specifically, when the runtime library pushes commands into the command buffer, the command group's location and size are added to the ring buffer. Simultaneously, a PUT register is updated with a pointer to the command group. The command processor retrieves the command group each time the PUT registers are updated and uses the GET register to notify the host device of fetch completion. Once launched on the GPU, a task cannot be preempted until completion. This is due to the GPU's lack of exposed software interface for task execution suspension mechanisms [30], unlike an interruptible CPU. Consequently, the GPU cannot preserve the task context in the same manner as a CPU. However, mainstream GPUs support concurrently executing GPU tasks from mutually untrusted processes [23, 46], like Nvida MPS [18] and AMD ROCm [2], making performance interference through GPU computation resource contention possible. Existing GPU support tasks killing mechanisms [16, 30] which can stop specified ongoing tasks on GPU at process-granularity without modifying any GPU configurations.

Work Flow: The entire workflow of a GPU task execution comprises four phases: memory allocation, code/data transfer between host buffer and GPU buffer, task dispatching and computation, and data transfer from GPU to host device. During the memory allocation phase, the GPU driver allocates memory space for GPU task execution and creates PTEs. In the second phase, the host device specifies the source and destination for the data transfer, transferring input data and loading codes from the host buffer to the GPU buffer. Once the data is transferred to the GPU buffer, the task is dispatched to the computing unit to execute. Finally, the results are transferred from the GPU buffer back to the host buffer.

2.2 Arm TrustZone and Stage-2 Translation

Arm TrustZone: Arm TrustZone is a hardware security mechanism that divides computational resources into two domains: the normal world and the secure world. The normal world cannot access the resources of the secure world, while the secure world can access all resources. This asymmetrical permission arrangement allows the normal world to run a feature-rich, large-size commodity OS, whereas the secure world runs a smaller, secure OS with fewer features. The two OSes cannot execute concurrently on a CPU core at the same time, thereby ensuring CPU usage isolation. The transition between the two worlds is supervised by the highest privilege level, known as the secure monitor, which employs a specific instruction called a secure monitor call (smc). Arm TrustZone also enforces memory isolation at the bus level, preventing the normal world from accessing the secure world's memory.

Stage-2 Translation: The two-stage memory translation mechanism is commonly employed in high-end series, such as Arm Cortex-A, to support virtualization. This mechanism is responsible for mapping the Virtual Address (VA) in applications and the OS to the Physical Address (PA). Specifically, Stage-1 translates the VA into an Intermediate Physical Address (IPA), and then Stage-2 maps the IPA to the PA. When virtualization is not enabled, the VA is directly translated into the PA.

3 THREAT MODEL AND SECURITY GOALS

Threat Model: AvaGPU is designed to protect against privileged attackers capable of executing arbitrary code and reading/writing any memory in the REE on CPS platforms where GPU is shared between TEE and REE. The adversarial goal is to compromise the availability of secure safety-critical GPU task execution, such as object detection in autonomous vehicles, through Denial of Service (DoS) attacks. It's important to clarify that AvaGPU focuses on computational availability guarantee of GPU tasks, and is complementary to existing work [61] that provides system availability guarantee with only CPU as the computational unit. Concretely, there are six unique attack vectors. From the perspective of GPU task execution, (1) attackers can modify the GPU code and data stored in host buffers. (2) Attackers can manipulate the DMA controller to prevent GPU data transfer from completing correctly and promptly. (3) Attackers can manipulate CPU task and GPU task scheduling, which includes blocking GPU task command commits or submitting multiple GPU tasks to compete for shared resources, such as cache or computation units, on the GPU. In terms of GPU management, (4) attackers may either deny access to or maliciously configure the GPU, such as modifying the GPU frequency through the MMIO interface. (5) They can manipulate GPU memory address translation by modifying the GPU Page Table Entry (PTE) and Page Table Directory (PD) that refers to the table pointing to page tables for more granular address translation. (6) Attackers can exploit vulnerabilities of untrusted GPU tasks to arbitrarily read, write, and execute in their memory space.

Assumptions: AvaGPU targets real-time CPS where the worst case execution time of safety-critical tasks are well understood for schedulability analysis. We also assume the designer of the CPS system has the access to GPU tasks' source code. These GPU tasks

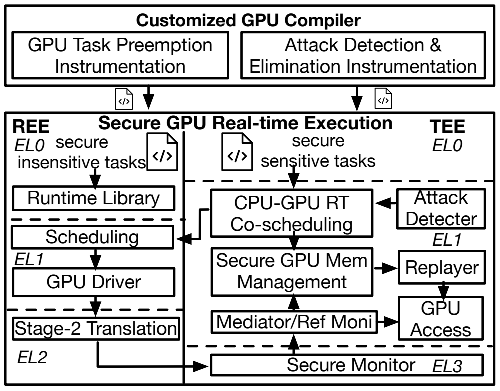


Figure 1: AvaGPU System Overview

are compiled on a trusted machine with our customized compiler. The AvaGPU's software components, including Stage-2 memory translation, the software stack in secure world, and the customized GPU compiler are trusted and presumed vulnerability-free. Any software components in the TCB on the CPU are verified through secure boot or remote attestation. Device initialization in REE is also part of the secure boot process. We also trust the hardware, including the CPU and integrated GPU, secure timer, as well as the corresponding supporting firmware. The physical attacks [29], cryptographic-based attacks, algorithm complexity attacks [41, 60], and side-channel attacks [45] are out of the scope of this paper.

System Goals: AvaGPU aims to ensure that secure CPU and GPU tasks can complete correctly and timely.

(R1) Integrity of GPU Task's Code and Data: If a GPU task's code and data integrity aren't protected, it yields incorrect results. Thus, the first goal is to protect GPU task data and code integrity.

(R2) Secure GPU Task Input/Output: GPUs use DMA to move data to the GPU buffer. Attackers could corrupt memory content of secure GPU tasks with malicious DMA transactions or delay them with lengthy data transfers. Therefore, AvaGPU should prevent malicious DMA write, and preemptive GPU data transfer operations.

(R3) Availability of GPU Computation Resources: Without access to properly configured GPU hardware, a GPU task cannot execute. Therefore, AvaGPU needs to ensure that secure tasks have access to correctly configured GPU computation resources.

(R4) Real-time Protection for Secure GPU Tasks: Some CPS tasks requiring GPU computation are inherently real-time in nature. Latency in computation results could lead to accidents. Thus, all secure real-time tasks utilizing GPU should have timely completions.

(R5) Isolation of GPU Resources: A privileged attacker can maliciously manipulate GPU resource management, including GPU memory translation and command submission, to tamper with secure GPU tasks. Thus, AvaGPU must ensure the isolation of GPU resource management for secure and non-secure GPU tasks.

(R6) Maintaining a Minimized TCB: Large TCB may introduce potential vulnerabilities. Therefore, AvaGPU aims to guarantee GPU availability while maintaining a minimal TCB.

4 DESIGN

An overview of AvaGPU is shown in Fig. 1. There are three key components. First, a secure CPU-GPU co-scheduling framework couples priority of CPU and GPU tasks, making GPU scheduler recognize GPU tasks' priority to guarantee their enough computation resources timely. Second, a software instrumentation-based secure and fine-grained GPU task preemption mechanism, allows the GPU scheduler to reliably and effectively preempt GPU tasks, thereby effectively mitigating priority inversion. Third, a trusted GPU setup and management mechanism isolates secure tasks and essential GPU management functions from untrusted software to guarantee trusted GPU hardware configuration, dynamic resource management and GPU task execution environment.

4.1 CPU-GPU Co-scheduling Infrastructure

The goal of the real-time CPU-GPU co-scheduler is to couple the priority for both CPU and GPU tasks, ensuring timely GPU-enabled task completion. A naive approach to construct such a scheduler to move the combined infrastructure to the secure world. However, this significantly increases the complexity of the TCB. To address this, recent work [61] adapts hierarchical scheduling to decouple the real-time scheduler of REE from the trusted scheduler in TEE, and rely on real-time scheduling theory based on the world scheduler to ensure the allocation of processor resources is adequate for both the secure world and the non-secure world to complete the workload. However, directly adapting this paradigm for CPU-GPU co-scheduling poses two new challenges. First, GPU lacks the individual world abstraction provided by ARM TrustZone, unlike the processor counterpart. This necessitates a flatten design on the scheduler. Second, the new co-scheduler has to ensure coherency of priority between a hierarchical CPU scheduling and flatten GPU scheduler.

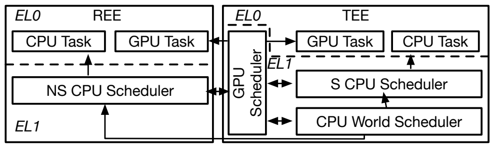


Figure 2: AvaGPU Real-time CPU-GPU Co-scheduling

To tackle this challenge, we propose to decouple the scheduler infrastructure of CPU scheduling and GPU scheduling while maintaining coherency on priority inheritance. The high level design is shown in Fig. 2, where the CPU scheduler is adapting the hierarchical scheduling while the GPU scheduler is flatten. To maintain coherency between the two, GPU execution strictly follows the world switching, where the accelerators prioritize on GPU tasks from the secure world when the processor is in secure state, and vice versa. Furthermore, within a single secure domain, GPU tasks inherit the priorities of the CPU tasks that they are associating with. However, it is important to note that such tightly coupled scheduling infrastructure may not provide the best performance. Since the focus of this work is not on real-time scheduling, we will leave such exploration as future work. Yet, it is important to note that AvaGPU is designed to be extensible to different types of CPU and GPU co-scheduling algorithms. The findings and designs will also generalize to or inform new co-scheduling designs.

4.2 Secure GPU Task Preemption

Preemption is one of the important features in real-time systems. However, existing GPU scheduling infrastructure does not allow task preemption. To tackle this challenge, there are two key design components to enable secure GPU task preemption. First, to enable preemption on GPU tasks (kernels), AvaGPU instruments both the secure and non-secure GPU tasks to enable software-based self-suspension and resumption. Second, to ensure that the instrumentation bypass can be detected, when the tasks are compromised or intentionally modified by the REE, AvaGPU adapts a two-stage defense mechanism that prevents/eliminates suspension instrumentation bypassing attack as early as possible.

GPU Task Execution Suspension and Resumption: AvaGPU supports GPU task suspension by instrumenting the GPU task, enabling it to self-terminate at specified locations after execution context saving. When a suspended GPU task is resumed, it restores the execution context and continuously executes from previous suspended point. The GPU task suspension mechanism consists of two stages, preventing the GPU hardware from queuing threads and intra-thread GPU task self-suspension. First, AvaGPU prevents the GPU hardware scheduler from queuing task threads. Each GPU thread always processes one input unit. However, GPUs allow applications to submit tasks requiring more threads than a GPU's total core count. Consequently, excess threads execution are queued in hardware and scheduled by the GPU hardware scheduler. However, hardware queued threads cannot be terminated until execution, causing preemption delays. AvaGPU solves this problem by transforming the GPU tasks to only apply the maximum concurrent threads a GPU supports, with each thread processing multiple thread workloads (i.e., multiple input units) sequentially. For example, Fig.3 illustrates the transferred GPU task *calculate_temp*. The variables *tx* and *ty* are used in line 14 to index the data processed by each thread. Rather than deriving *tx* and *ty* from the internal variable *threadIdx* only once in each thread, AvaGPU instruments the GPU task to obtain multiple ones from a software queue, as shown in lines 4 to 7. Consequently, all GPU task threads can be preempted instantly, as no threads remain queued on GPU hardware.

Second, AvaGPU proposes an event-based GPU task preemption approach, utilizing customized compiler to instrument the GPU task. The instrumentation allows GPU tasks to self-monitor software preemption signals (i.e., share variables) updated by trusted GPU scheduler. The memory region containing these signals is set to read-only for the untrusted OS via Stage-2 translation to prevent tampering. When a preemption signal is received, the GPU task saves its current execution context, and suspends execution. The suspended execution resumes by restoring the saved context when the scheduler resumes the suspended task. The saved context includes GPU register values, while the memory content remains unaltered for suspended tasks. As shown in Fig. 3, function *check_preemption* in line 12 examines a preemption signal and invokes *context_save* in line 12 if set. When a new thread is launched, *context_restore* in line 2 restores the context if suspended. Leveraging the predictability of real-time CPS, AvaGPU's event-based

```
01.__global__ void calculate_temp(...){  //GPU Code:
02.if(suspended()){context_restore()}  //context restoring
03.while(true){
04.if(queue_empty()){return;}
05.int curr_thread_idx = trd_dequeue();
06.int tx = get_bx(curr_thread_idx);
07.int ty = get_by(curr_thread_idx);
   ...
08.for (int i=0; i<cnt_preempt; i++){
09.   temp_cal(bx,by,tx,ty);
10.   checkpoint1 = 1;      //preemption checkpoint update
11.}}}
12.if(check_preemption()) context_save();  //context saving
13.for (; i<iteration ; i++){
14.   temp_cal(bx,by,tx,ty);}}}
15.calculate_temp<<<dimGrid, dimBlock>>> //CPU Code
```

Figure 3: GPU Task Transformation in AvaGPU

preemption approach minimizes unnecessary checks by examining preemption signals at submission time for GPU tasks. Consequently, preemption checking intervals align with the largest common factor of all GPU task release periods. Note that the long data transactions are also splitted into multiple ones to support preemption.

Instrumentation Modification Defense: AvaGPU verifies the signature of instrumented non-secure GPU tasks' hash prior to loading, with the aim of identifying any instrumentation modification from REE. Specifically, each GPU task has to be signed by the developer, and the signature for the task has to be checked by AvaGPU before loading it into the GPU for execution. Subsequent loads of the same GPU task can just reuse the previous hash checksum for verification without the need for public crypto operations. However, the delay between command queue commitment and GPU code execution could potentially allow the REE to execute a Time-of-Check-Time-of-Use (TOCTOU) attack by modifying GPU task code after signature check. To defend against this attack, AvaGPU sets memory space storing the GPU task codes as read-only in REE via stage-2 memory translation before verifying the GPU task signature.

Runtime Preemption Instrumentation Bypassing Defense: AvaGPU defends against runtime preemption bypassing using an attack detection and elimination mechanism instead of an attack prevention mechanism, such as memory safety [55] that typically introduces high runtime overhead. Specifically, AvaGPU identifies runtime preemption instrumentation bypassing by monitoring corresponding consequences, i.e., progress delay on secure tasks. AvaGPU detects progress delay by verifying whether the expected amount of computation is finished in a fixed amount of time. Specifically, AvaGPU first uses a customized compiler to insert progress delay checkpoint into secure GPU tasks. When control flow of secure GPU task reaches a delay checkpoint, the value of corresponding delay checkpoint pass variable will be set, as shown in line 10 in Fig. 3. To verify the progress between two checkpoints at runtime, AvaGPU triggers the secure timer after a period from the previous delay checkpoint, checking whether the next checkpoint pass variable is updated. If the next expected delay checkpoint pass variable value is not set, malicious GPU task contends shared resources. The preemption instrumentation is bypassed. As shown in Fig. 4, the system is expected to pass through *checkpoint2* at time point *t2*. If *checkpoint2* is not passed at *t2*, an attack is occurring.

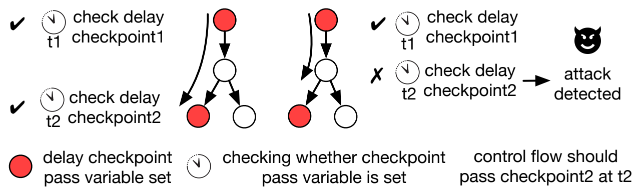

check delay t1 checkpoint1
check delay t2 checkpoint2
check delay t1 checkpoint1
check delay t2 checkpoint2
attack detected
delay checkpoint pass variable set
checking whether checkpoint pass variable is set
control flow should pass checkpoint2 at t2

Figure 4: Preemption Bypassing Detection

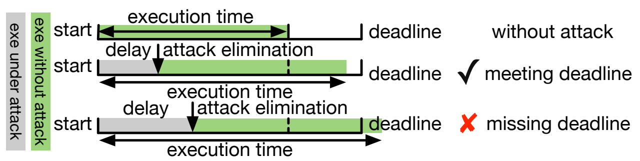


Figure 5: Preemption Bypassing Defense Strategy

When an attack is identified, AvaGPU kills the non-responsive GPU task [16, 30]. Resetting a specific hardware channel associated with a GPU task will stop corresponding ongoing computation on the Nvidia GPU. This action is specific to individual GPU tasks, only erasing the computational context of the target, thus leaving other tasks' memory untouched and preserving their accurate execution. Yet, only killing the non-responsive GPU task cannot prevent an attacker who turns on the malicious logic one by one. To be conservative, AvaGPU suspends other non-secure GPU tasks by sending software signals via secure shared memory until the delayed GPU task is complete. The attack detection strategy, including the number and the position of delay checkpoints in a GPU task, is decided automatically by AvaGPU which will be described in the next section. The delay checkpoint pass variables are located in TEE. Thus, neither an untrusted CPU process nor an untrusted GPU task can update delay checkpoint pass variables arbitrarily. Additional discussion on different strategies is in Section 9.

Defense Strategy Generation: As shown in Fig. 5, the time slack between real-time task execution time and the deadline is limited. Detecting progress delay at the end of task execution may lead the task to miss its deadline. Furthermore, real-time systems often maintain a system utilization upper bound to make sure the system is schedulable [40], making AvaGPU has less time to eliminate progress delay. To solve this problem, AvaGPU inserts multiple delay checkpoints in a secure GPU task. In this way, the longest delay time of a GPU task will be the duration between two checkpoints. However, inserting more checkpoints will introduce more runtime overhead. With all these system constraints considered, AvaGPU generates the progress delay checkpoint strategy given a user-provided real-time system utilization upper bound. The generated strategy guarantees that real-time tasks can complete in time after detecting GPU progress delay under the system utilization upper bound while maintaining a minimized runtime overhead.

Defense Strategy Optimization Formulation: In CPSs, CPU execution often relies on the output of secure GPU task execution. As a result, GPU execution often synchronizes with CPU task execution. AvaGPU addresses the challenge of generating defense strategies by formulating the process as an optimization problem. AvaGPU assumes that there is a set of m CPU tasks, denoted as $\tau_0, \tau_1, \ldots, \tau_m$. Each task τ_i has a deadline d_i, execution duration of e_i (where e_i represents the total execution delay for both CPU and GPU execution, excluding the runtime overhead of the progress delay detection in GPU tasks), and total GPU execution duration g_i. The developer provides the expected system utilization upper bound, $U_{up} \in [0, 1]$, as input to the model. The U_{up} is determined by the scheduling algorithm deployed in the system, which in turn determines the schedulability of the system. The output defense strategy consists of: the number of delay checkpoints for each CPU task $\mathbf{N} = \{n_i | 0 \leq i < m\}$, and the progress delay checkpoint positions in GPU tasks invoked from each CPU task $\mathbf{C} = \{c_{ij} | 0 \leq i < m,\ 0 \leq j < n_i\ 0 \leq c_{ij} \leq g_i\}$. To adhere to the developer's expected system utilization, the total system utilization resulting from the generated strategy must not exceed the developer's specified upper bound for system utilization:

$$\sum_{i=0}^{m-1} \frac{e_i + n_i \Delta c}{d_i} \leq U_{up}. \tag{1}$$

To guarantee every task can complete before the deadline, the total task execution time for each task τ_i after detecting the progress delay should be less than its deadline

$$max\{e_i + c_{ij+1} - c_{ij}\} + n_i \Delta c \leq d_i,\ 0 \leq j < n_i. \tag{2}$$

The optimization objective is to minimize the runtime overhead introduced by attack detection, which can be formulated as:

$$min(\Delta c \sum_{i=0}^{m-1} n_i), \tag{3}$$

where Δc represents the overhead of checking delay checkpoint pass variables, which is measured through program execution profiling. We adopt the genetic algorithm [31] to solve this optimization problem. The details of the algorithm are presented in Appendix. A.

4.3 Trusted GPU Setup and Management

The GPU driver manages GPU, such as initializing GPU, managing power and memory, etc. Secure GPU tasks need trusted driver to have availability guarantee. However, directly migrating the GPU driver into the TEE significantly increases the TCB. Recent studies [47, 61] have replayed pre-recorded MMIO read/write sequences and values to operate I/O devices or accelerators (i.e., GPU), offering a trusted driver without substantially increasing the TCB. However, they cannot be directly applied to guarantee GPU availability.

Shown in in Fig. 6 (a), [47] ensures confidentiality and integrity by isolating the trusted GPU replayer from the untrusted GPU stack. However, this approach doesn't prevent attacks from operating the GPU in the REE, leaving GPU availability unprotected. Similarly, as shown in Fig. 6 (b), [61] protects availability of I/O devices using a global replayer in TEE, followed by an I/O reference monitor to verify the validity of all MMIO operations. However, existing design in [61] does not have dynamic memory management needed by GPU, including runtime-allocated memory addresses and ring buffer pointer positions. Thus, as shown in Fig. 6 (c), AvaGPU offers secure dynamic resource management prior to the replayer, providing the runtime-determined resource metadata for the replayer.

Trust GPU Management Overview: To prevents REE from arbitrary operating GPU, GPU MMIO memory space is configured

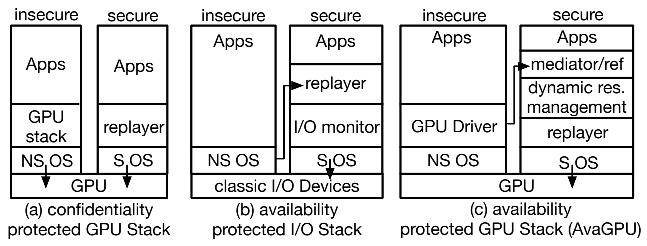


Figure 6: Solutions about Secure Peripheral Driver

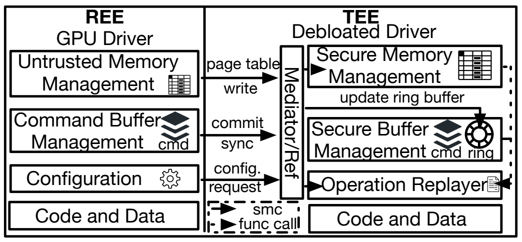


Figure 7: AvaGPU Secure GPU Management

as inaccessible for REE using Stage-2 translation. Any GPU configurations from REE should send a GPU configuration request to TEE. A GPU configuration mediator in TEE validates configurations from REE based on OEM policies. As shown in Fig. 7, to provide a trusted GPU driver while minimizing TCB, AvaGPU provides a minimalist dynamic resource management (i.e., memory and command buffer management) of the GPU driver in the TEE. Dynamic memory management and GPU task commitment requests in the REE driver are trapped and verified in TEE before GPU task execution. Once resource management is completed in TEE, the GPU replayer operates GPU with the result metadata from resource management and recorded I/O transaction message. However, trapping resource management from the REE into the TEE increases runtime overhead significantly. To solve this problem, AvaGPU introduces GPU PTE management delegation, separated command buffers, and batch command buffer synchronization mechanisms based on TEE and Stage-2 memory translation.

Trusted GPU Memory Management: The GPU driver manages GPU memory to provide isolation among GPU tasks. An untrusted REE OS can manipulate GPU memory management to compromise the integrity of secure GPU task execution, such as by mapping the virtual address of secure GPU task code to the physical address of malicious GPU task code. Thus, to support trusted GPU memory management for secure GPU tasks, AvaGPU manages GPU memory in TEE. However, naively trapping every GPU memory management function from REE to TEE introduces significant runtime overhead due to the context switch between the REE and TEE. To tackle this challenge, we build on top of the observation that most of the page table operations are read-only, AvaGPU therefore only traps the security sensitive write operations to TEE. To realize this design with minimal overhead, AvaGPU leverages the stage-2 translation to enforce the access control for REE. For each write request that is trapped into TEE, the memory management mediator refers back to the physical memory range list. Requests for memory outside of the range will be rejected. The access control is similar to Enclave Page Cache Metadata (EPCM) in Intel's Software Guard Extensions (SGX) and Reverse Map Table (RMP) in AMD Secure Encrypted Virtualization (SEV). However, different from these TEE designs, AvaGPU maintains a minimal resource management to ensure availability guarantee for secure GPU tasks.

Trusted GPU Buffer Management: The GPU runtime library generates GPU commands that are submitted to the command buffer. The location and size of each command group are stored in the ring buffer by command buffer management. The GPU continuously fetches commands from the ring buffer to execute tasks. An untrusted OS can maliciously modify commands of secure tasks, corrupting the execution integrity and availability of secure tasks. To address this issue, the TEE needs to take control of the command buffer and ring buffer management. AvaGPU migrates ring buffer and replicates command buffer management within the TEE. However, switching from REE to TEE when submitting every GPU command introduces significant runtime overhead. To address this issue, AvaGPU proposes the separated command buffer mechanism. Specifically, REE GPU driver submits GPU commands in the command buffer located in the REE. AvaGPU synchronizes the GPU command buffer in the REE with the one in the TEE, from which the GPU fetches commands. However, actively synchronizing two command buffers frequently introduces non-negligible runtime overhead. Thus, AvaGPU proposes a batch command buffer synchronization mechanism. Specifically, the two buffers are synchronized after the REE sends a request upon completion of a group of command commitments by a task. This strategy is active only when the GPU is busy, that is, when the ring buffer is not empty, thus avoiding delays caused by waiting for commands.

GPU Task's Code/Data Isolation and Transfer Protection: Code and data integrity of secure tasks should be protected from untrusted OS as they assure correct GPU task execution. Besides the page table in GPU, GPU tasks can also use DMA to transfer large trunks of data between the GPU memory space and host memory space [27, 59]. To prevent non-secure GPU tasks from sending DMA request on GPU to corrupt secure memory by making source and destination address in DMA commands fall into secure GPU tasks' memory space. AvaGPU employs a DMA reference monitor in TEE to validate DMA memory access in GPU commands before GPU task execution. Specifically, the DMA reference monitor validates that the source and destination addresses of DMA transfer commands from non-secure GPU tasks do not fall into the memory address space of secure GPU tasks before these commands are pushed into the command queue.

5 IMPLEMENTATION

We implemented a prototype of AvaGPU on the Jetson AGX Orin platform. This platform has 12 Arm Cortex-A78AE CPU cores, an GPU with 2048 CUDA cores, and 32GB memory. The software stack includes LLVM 17.0.0 compiler, the CUDA-11 runtime library, the Linux 5.10.65-tegra operating system, and the OP-TEE secure OS based on Arm TrustZone.

CPU-GPU Co-scheduling Infrastructure: The end-to-end scheduling infrastructure in AvaGPU comprises a hierarchical CPU task scheduling infrastructure and a global GPU task scheduling system.

We utilize the secure physical timer on the Cortex-A78AE to trigger the hierarchical CPU scheduling process. Both the world scheduler and the secure world scheduler employ Rate Monotonic (RM) scheduling. In our prototype, global GPU task scheduling aligns with the decision of CPU schedulers, allowing the GPU task of the latest CPU process to preempt the current executing GPU tasks. Additionally, it maintains a resource consumption table to manage each GPU task's computational resources, including registers, shared memory, and the number of threads.

GPU Task Preemption Instrumentation: The GPU task preemption method comprises two key components: identifying where to check for preemption signals, and determining the data to save and restore. Given the predictability in real-time systems, the GPU task checks for preemption signals whenever a task is released. Thus, the checking period is set as the greatest common factor of all tasks release periods. To figure out the code location of checkpoints, we use LLVM front-end passes to record timestamps with *clock64* function for each basic block and the statements within blocks that include preemption checkpoints. AvaGPU ensures that at least one preemption checkpoint is included within a fixed number of loop cycles prior to generating the strategy. The specific number of cycles is determined by the task release period. When a GPU task is preempted, AvaGPU saves all registers into secure memory, restoring them when the task is resumed. Memory content are preserved during suspension to maintain the content throughout the GPU task's execution. Note that GPU has more registers than CPU, leading to noticeable runtime overhead during context saving in task preemption. Thus, when multiple GPU tasks of the same priority execute, AvaGPU use the preemption algorithm in Appendix A to preempt as fewest GPU tasks as possible, while still freeing enough computational resources for the next task to be scheduled.

Preemption Bypassing Detection and Elimination: The preemption bypassing detection and elimination process consists of two components: detection strategy generation and attack checkpoint instrumentation. We use a Python script to generate an optimized delay detection strategy with Algorithm 2. Then we inserted delay checkpoints at the specified points within the GPU task, using the same approach as when inserting GPU task preemption checkpoints. Each checkpoint updates a unique variable in secure memory, marking the execution progress. A secure timer is set according to the strategy. It is noted that if two adjacent checkpoints are located in two GPU tasks, two checks are implemented. If a new checkpoint variable isn't updated when triggered, all non-secure GPU tasks are suspended though setting preemption variables. At last, AvaGPU halts the non-terminated task by resetting its hardware channel, thus eliminating the attack.

Trusted GPU Access: AvaGPU reserves memory in secure world for GPU resource management and Stage-2 translation, providing trusted services. Specifically, a 16MB secure memory region is reserved for the GPU page table and Stage-2 translation table, with the Stage-2 configuration registers [21] appropriately configured. AvaGPU uses a flat memory model for the normal world's Stage-2 translation, reflecting the single memory space. When page access rights change, the access permission bits of a PTE are modified, and the MMIO addresses of the GPU are configured as readable-only in the normal world. Our prototype includes a GPU configuration mediator that restricts non-secure tasks from modifying GPU frequency during secure tasks execution. However, AvaGPU supports policy-based hardware configuration mediator. For example, OEMs can limit frequency adjustments to specific ranges.

Trusted GPU Resource Management: For secure GPU memory management, we migrated GPU virtual memory management and GPU MMU operations from nvgpu driver to OP-TEE secure OS kernel. We reserve a 4k-aligned memory region to store PD/PTE, configuring the access rights of this region in Stage-2 memory translation to be readable-only for normal world and both readable and writable for secure world. To trap the PD/PTE update from normal world to secure world, we substitute the normal world's PD/PTE operation functions (such as *nvgpu_pd_write*) with an SMC. The secure world's memory management mediator validates these write requests, ensuring the physical addresses in the PTE from the REE are outside the secure world's physical memory, and that the PD doesn't point to the secure GPU tasks' page tables. We implemented trusted ring buffer and command buffer management in secure world by migrating command/ring buffer management functions from the nvgpu driver to secure world. To synchronize command buffers, the command are copied between two worlds. During function migration, we identified and removed certain functions and data structures, like those used in the Linux kernel, debugging, and logging functions in nvgpu drivers, to reduce the TCB. Nvidia's use of virtual addresses in DMA transfers negates the need for our prototype's DMA reference monitor to check the command's source and destination. However, for vendors like Arm [27] that don't support virtual address DMA transfers, the DMA reference monitor must reject commands if the source and destination addresses in the control block fall into the secure memory space.

GPU Replay Message Generation: Generating GPU replay messages requires recording MMIO read/write operations and correlating them with runtime resource management decisions. We logged these operations by instrumenting user-space CUDA API calls and MMIO real/write nvgpu driver functions like *nvgpu_os_readl/writel*. Additionally, we differentiated user-space operations from interrupt handling by instrumenting all interrupt handlers in nvgpu driver, as messages tied to interrupt handling are replayed only after an interrupt is triggered. To correlate runtime resource management decisions with MMIO operations, we first obtain decisions like GPU page table base address, code/data address, and ring buffer address from the instrumented runtime management functions in the driver. Next, we located recorded MMIO operations containing these decisions. During runtime, AvaGPU feeds the dynamically generated decisions in the messages to control the GPU.

TCB Analysis: AvaGPU minimizes the TCB size through driver debloating. As shown in Table. 1, the increased system TCB in AvaGPU includes operations for Stage-2 memory address translation, secure GPU management, mediators, GPU message replayer, and a scheduling framework. The prototype adds a total of 7301 Lines of Code (LoC) to the TCB, with the trusted debloated GPU driver accounting for 5331 LoC. This includes 153 LoC for hardware configuration and memory management mediators, as well as DMA reference monitor framework, a significant reduction from the 46K LoC in the nvgpu driver. The TCB size may further vary depending on the quantity and size of secure tasks.

Table 1: Line of Code (LoC) of Components in AvaGPU

S2 Trans.	S GPU Mng	Medi	Repl	Schd	NS GPU Dri	Compiler	Record
413	4861	153	317	1557	321	4212	1268

Trans.: Translation, Mng: Manager, Medi: Mediator, Repl: Replayer, Schd: Scheduler, Dri: Driver

6 EVALUATION

In this section, we evaluate AvaGPU using the prototype described in Sec.5. We aim to answer the following questions: (1) Can AvaGPU effectively defend against availability attacks? (2) What is the system overhead of AvaGPU on GPU tasks? (3) What is the system overhead on CPU tasks? (4) What is the real-time performance of real-time CPU tasks that involve GPU executions? To address these questions, we (1) conduct defense case studies to examine AvaGPU's impact on two availability attacks, i.e., malicious GPU frequency reduction and preemption bypassing attacks, (2) measure AvaGPU's system overhead on the Rodinia benchmark suite[19], (3) measure AvaGPU's runtime overhead on SPECrate 2017 benchmark, and (4) evaluate the real-time performance of both synthetic real-time tasks and three real-world GPU tasks, as well as CPU hierarchical scheduling computation cost on micro benchmark. Additionally, we evaluate AvaGPU's efficiency in generating performance interference attack defense strategies. More details can be found in Appendix A.

6.1 Defense Case Study

To demonstrate AvaGPU's effectiveness in defending against availability attacks, we conduct two case studies on the prototype platform: using AvaGPU to defend against GPU working frequency reduction and preemption bypassing attacks.

GPU Working Frequency Reduction: In a GPU working frequency reduction attack, an attacker with kernel privileges delays GPU task execution by reducing the GPU's working frequency. To delay the victim task as much as possible, the attacker lowers the GPU frequency from its highest setting (1.3GHz) to the lowest (115MHz) during the execution of the GPU task, causing CPU tasks to miss deadlines. In this case study, we select three open-source applications commonly used on commercial drones: Object Detection (OD) using DetectNet [10], Image Classification (IC) with ImageNet [13], and Image Compression (IP)[12]. These envisioned CPS applications operate autonomously under complex environments. As such, they often require powerful platforms with GPUs to support timely execution of object recognition, object classifications or other complex algorithms. Each application has a 50ms deadline, corresponding to 20 fps camera operation on a drone camera[11]. Secure applications (OD and IC) operate in secure world, while the non-secure application (IP) operates in normal world. The upper bound of system utilization is 69%, aligned with the RM scheduling algorithm in our prototype. We continuously execute each task at least 1000 times. As indicated in Table. 2, the average execution time for the three applications increases under a GPU working frequency reduction attack, resulting in an 85% deadline miss rate. However, with AvaGPU deployed, tasks meet deadlines even under this attack, with 7.52% increase in system utilization. This is because the GPU configuration mediator denies requests to reduce GPU execution frequency during secure task execution.

Table 2: Effectiveness of Frequency Reduction Defense

	Exe Time w/o atk w/o Ava	Utilization w/o atk w/o Ava	Exe Time w atk w/o Ava	Miss Rate w atk w/o Ava	Exe Time w atk w Ava	Utilization w atk w Ava
ObjDetect	10.59ms		56.15ms		12.25ms	
ImgClassify	1.64ms	47.36%	38.93ms	85.00%	1.89ms	54.88%
ImgCompress	11.45ms		54.26ms		13.3ms	

Exe: Execution, w/o: without, w: with, atk: attack, Ava: AvaGPU

Preemption Bypassing Attack Defense: In this case study, we evaluate the effectiveness of defending against preemption bypassing attacks on Autoware [5], a widely-used open-source autonomous driving software that requires high-end platform [6]. Within Autoware, perception tasks are crucial for detecting, recognizing, and tracking objects, with 3D object detection and tracking [64] utilizing the GPU for accelerated processing. The correctness and availability of 3D object detection and tracking are the prerequisite of subsequent functions such as planning and controlling. Thus, we protect it with AvaGPU in secure world. Other tasks such as sensor simulation run in normal world. The mission and map we used in simulation is sample-rosbag [7] in Autoware. We have profiled the execution time of each task, setting the deadline equal to the period, which corresponds to the time interval between two consecutive task executions. We simulate the attack by exploiting the buffer overflow vulnerability in five synthetic non-secure GPU tasks that run arithmetic computation loop to bypass the preemption, contending computation resources with victim GPU task. The buffer overflow vulnerability in untrusted GPU tasks is introduced at the end of the loop to overwrite the loop condition, making loop execute without stopping. To show the effectiveness of defense mechanism, we measure the execution time of 3D object detection and tracking under three system deployment scenarios. As illustrated in Table. 3, under a preemption bypassing attack, the average GPU task execution delay is 6.99 times longer than the execution without an attack, exceeding the task deadline by 1.36 times. The Autoware loss object perception of surrounding objects can be observed in visualized panel, impacting the availability of the following functionalities. When AvaGPU is deployed, the task meets its deadline even under a preemption bypassing attack, but with a 17% average increase in runtime overhead.

Table 3: Effectiveness of Preeemption Bypassing Defense

	w/o Atk	w Atk w/o AvaGPU	w Atk w AvaGPU	Deadline
Min	43824 us	281123 us	51142 us	250000 us
Max	55399 us	385721 us	65094 us	250000 us
Avg	48470 us	338760 us	56710 us	250000 us

6.2 System Overhead on GPU Benchmark

To evaluate AvaGPU's runtime overhead on GPU tasks under different workload distributions, we measure the execution times of applications in Rodinia benchmark suite [19] on systems with and without AvaGPU under three workload (the sum of the execution time of each application in every world, divided by its period) distributions between secure and non-secure world, i.e., 25%/75%, 50%/50%, and 75%/25%. For each distribution, we evaluate the runtime overhead of applications in both worlds over 10 iterations.

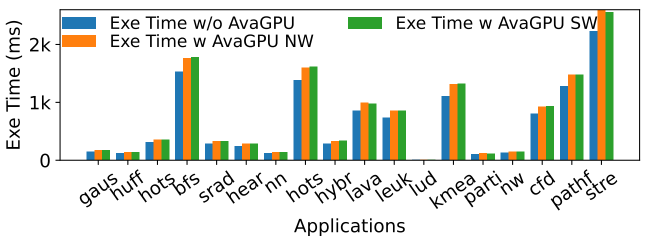


Figure 8: Runtime Overhead of Rodinia Benchmark

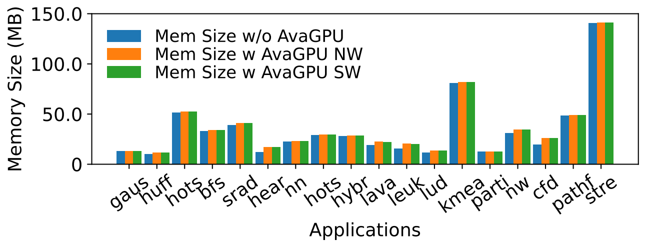


Figure 9: Memory Overhead of Rodinia Benchmark

Specifically, in each measurement iteration, under a given workload distribution, both the number and choice of applications in each world are randomly selected to prevent biased results. Since no real-time GPU task benchmarks exist, we treat Rodinia applications as real-time GPU tasks. Specifically, after selecting applications in each iteration, we assign deadlines to applications to ensure that the workload distribution in each world is satisfied and the entire system utilization reaches the upper bound, i.e., 69% under the RM scheduling algorithm. Assigning deadlines to tasks is reasonable here as we aim to measure the runtime/memory overhead of real-time applications with varied parameters, rather than assessing real-time performance that requires actual application deadlines. To measure the runtime overhead of each AvaGPU component, we record starting and ending timestamps to calculate execution delays.

Runtime Overhead Analysis: The maximum runtime overhead of each application when they runs in normal world and secure world is shown in Fig. 8. The highest runtime overhead is 17.12% for kmeans in normal world and 18.49% for kmeans in secure world. The runtime overhead for each component in AvaGPU in normal world and secure world are presented in Table. 4 and Table. 5. Preemption checking and context switching exhibit the highest average runtime overhead among all components. This overhead is mainly due to checking variables shared between the CPU and GPU, and saving and restoring the execution context. The data transfer overhead primarily arises from extra DMA configurations due to transaction splitting, which increases in proportion to the size of the data. The code verification overhead on the other hand, is solely caused by validating the GPU task code integrity in the normal world, and it increases with the size of the code. The scheduler's runtime overhead is introduced by CPU-GPU co-scheduling. It fluctuates depending on various real-time task parameters throughout the system. The overhead incurred by memory management and command buffer management is primarily attributable to page table and command/ring queue operations. The memory management overhead grows in proportion to the size of both the code and data of GPU tasks. This overhead also comprises the runtime overhead incurred by PTE checking in memory management mediator for non-secure GPU tasks. The configuration mediator overhead occurs due to the verification of the validity of GPU operation requests made from the normal world. Preemption bypassing attack detection runtime overhead is only introduced in secure tasks, mainly caused by checking the progress variables which are shared between CPU and GPU. The GPU context switching overhead is introduced by saving/restoring registers on GPU task preemption/resuming.

Table 4: Runtime Overhead Breakdown of AvaGPU (NW)

	preemp check	data transf	code verif	ctx swch	sched	mm mng	cmd buffer	config media	total
Avg	3.47%	0.77%	0.66%	5.40%	1.61%	1.93%	1.94%	0.63%	15.87%
Max	3.96%	3.95%	1.18%	6.16%	1.96%	2.18%	2.19%	0.81%	17.12%
Min	3.10%	0.21%	0.21%	4.77%	0.97%	1.70%	1.75%	0.47%	14.51%

preemp: preemption, tranf: tranfer, verif: verification, ctx: context, sched: scheduling, mm mng: memory management, cmd: command, config: configuration, media: mediator

Runtime Overhead Difference between NW and SW: Table. 4 and Table. 5 show the runtime overhead differences between the normal and secure world. These differences include (1) GPU task code verification, (2) preemption bypassing attack detection, (3) memory and command buffer management, and (4) configuration mediation. The overall runtime overhead of GPU tasks in both the normal and secure worlds is similar due to a combination of factors. (1) AvaGPU does not verify the integrity of GPU task code in the secure world. (2) Preemption bypassing attack defense is only applied to secure GPU tasks through instrumentation. (3) In the secure world, memory management must validate every PTE submitted from the normal world and configure the Stage-2 translation table to enforce read-only permission for page tables of non-secure tasks. Additionally, commands must be copied from normal world to secure world before non-secure GPU tasks execution. (4) The configuration mediator in the secure world must verify the validity of configuration values from normal world requests. Additionally, all GPU operation requests from the normal world result in world context switch runtime overhead.

Memory Overhead: Fig. 9 illustrates the maximum memory overhead of each application when they run in normal world and secure world under different system utilizations. The memory overhead is mainly introduced by context saving when a GPU task is suspended. Among all applications in the benchmark, two bioinformatics processing programs, namely *hear* and *leuk* have the highest memory overhead, i.e., 37.98%, 31.10% in secure world and 38.21%, 29.93% in normal world. These programs require a large number of registers for efficient computation, which are saved during context-switching, thus leading to significant memory overhead. However, the memory overhead introduced by context-saving is less than 10MB, which is less than 0.04% of the total memory on our platform. Even on lower-performance platforms, such as the Nvidia Jetson Nano [17] with 2GB memory, the memory overhead accounts for less than 0.5% of the entire memory.

6.3 System Overhead on CPU Tasks

To evaluate AvaGPU's runtime overhead on CPU tasks, we measure the execution time of programs in SPECrate 2017 benchmark running in normal/secure world on the system with/without AvaGPU.

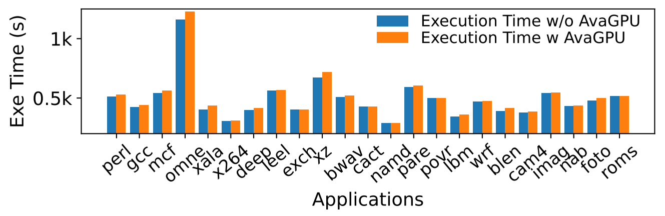
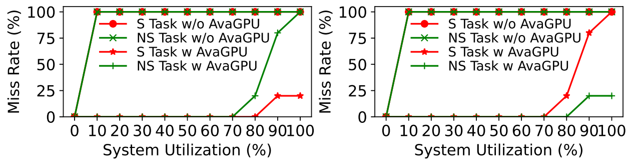


Figure 10: Runtime Overhead on CPU Tasks



(a) Secure Task Prioritized (b) Insecure Task Prioritized

Figure 11: Schedulability Analysis of AvaGPU

Table 5: Runtime Overhead Breakdown of AvaGPU (SW)

	preemp check	data transf	ctx swch	sched	mm mng	cmd buffer	attack detect	total
Avg	2.78%	0.53%	5.53%	1.17%	1.70%	1.48%	2.39%	15.58%
Max	3.53%	3.73%	6.13%	1.58%	2.16%	1.67%	3.21%	18.49%
Min	1.62%	0.21%	4.73%	1.07%	1.22%	1.21%	2.19%	13.14%

preemp: preemption, tranf: tranfer, ctx: context, sched: scheduling, mm mng: memory management, cmd: command, attack detect: preemption bypassing attack detect

Table 6: RT Performance of Real-world Applications

	preemp check	data transf	code verif	ctx swch	sched	mm mng	cmd buffer	attack detect	total
OD	2.38%	0.93%	N/A	5.18%	1.78%	1.67%	1.48%	2.18%	15.6%
IC	2.57%	0.81%	N/A	5.27%	1.57%	1.53%	1.44%	2.61%	15.8%
IP	3.13%	0.63%	0.83%	5.31%	2.03%	2.18%	2.09%	N/A	16.2%

preemp: preemption, tranf: tranfer, verif: verification, ctx: context, sched: scheduling, mm mng: memory management, cmd: command, attack detect: preemption bypassing attack detect

CPU Task Runtime Overhead: As shown in Fig. 10, the maximum, minimum, and average runtime overhead of AvaGPU on SPECrate 2017 benchmark in both worlds are 6.87%, 0.23%, and 2.25%, respectively. The runtime overhead on CPU tasks is introduced by Stage-2 memory translation mechanism that is used to enforce memory access control, preventing attackers from manipulating GPU in normal world without being trapped into secure world for verification. It needs additional page table walks, introducing runtime overhead for CPU tasks. AvaGPU guarantees the correctness of dynamic GPU resource management, GPU tasks commitment, and GPU configuration by taking over these operations from normal world GPU driver, i.e., relaying these operations from normal world GPU driver to secure world. However, these modifications are all located in the normal world GPU driver, thus having no impact on CPU tasks without GPU execution involved.

6.4 Real-time Performance of System

To evaluate the real-time performance of AvaGPU, we evaluate the real-time task miss rate for both synthetic real-time tasks under varying system workloads and real-world CPS GPU applications, demonstrating AvaGPU's feasibility for real-world use cases. We generate synthetic task sets by randomly creating ten benign CPU tasks that exclusively execute GPU tasks, with GPU tasks featuring different task execution times calculated through varying iterations of matrix multiplication. GPU execution durations are randomized between 10us and 20ms, including five secure tasks and five non-secure tasks. Task periods are set to yield total system utilization ranging from 0% to 100%. Each task executes 100 times. To evaluate the computation cost of hierarchical scheduling in AvaGPU, we also measure the execution time of each component in the hierarchical scheduler, including world scheduler, secure world scheduler, and normal world scheduler, during synthetic real-time tasks execution under different system utilization. To evaluate the influence of task priority on real-time performance, we alternate high priority between secure and non-secure benign tasks. We employ CARTS [50] to obtain root-level scheduling parameters. Real-time performance under an attacker in synthetic GPU tasks is shown by executing 10 malicious CPU tasks that have the lowest priority in normal world, running GPU tasks to achieve 100% CPU utilization for each group. We use the same settings as the GPU working frequency reduction case study (i.e., all tasks have a 50ms deadline, and the system utilization upper bound is 69%) as real-world workload.

Real-time Performance: Fig. 11 illustrates the real-time performance of synthetic tasks in AvaGPU. In a system without AvaGPU, all tasks miss their deadlines due to attackers submitting numerous GPU tasks in brief periods, contending for computational resources. However, with AvaGPU protection, benign tasks only begin to miss deadlines when system utilization of benign tasks exceeds 69%, in line with the theoretical result[40]. Prioritized tasks start missing deadlines at higher utilization levels since they are scheduled first when computational resources are limited. The real-time performance of synthetic tasks shows that AvaGPU can maintain reasonable real-time performance under malicious GPU resource contention. Table. 6 shows the real-time performance of a system running three real-world applications. The highest average runtime overhead, 16.2%, occurs on IP in the normal world. It includes 0.7% configuration mediator checking overhead. Nevertheless, all applications complete their tasks before the deadline, proving the feasibility of AvaGPU for systems running real-world applications.

CPU Hierarchical Scheduling Compuataion Cost: We instrumented AvaGPU hierarchical scheduler and normal world OS scheduler in Linux to record the scheduling event count and the total overhead over the execution of the same set of synthetic real-time tasks used in above real-time performance section. The total execution time of the tasks set is 15.79s. Table. 7 shows the computation cost of hierarchical scheduler when the system is schedulable (system utilization remains below 69%). The normal world scheduler (i.e., Linux scheduler) has significantly higher execution time compared to secure world scheduler and world scheduler because of complex data structure and process operations in Linux scheduler. Furthermore, the maximum scheduling events in normal world scheduler under different workloads is also higher than world scheduler and secure world scheduler. This is because Linux scheduler is jitter-based, scheduler works at every fixed interval. While world scheduler and secure scheduler is event-driven, working only when a new scheduling event arises, such as the arrival of a new task. The

Table 7: Hierarchical Scheduling Runtime Overhead

Scheduler	Max	Min	Avg	Max Events	Max Pct
World Scheduler	0.83 us	0.71us	0.72 us	2863	0.01%
NW Scheduler	36 us	23 us	28 us	4309	0.80%
SW Scheduler	0.86 us	0.76 us	0.79 us	2261	0.01%

Max: Maximum Execution Time, Min: Minimum Execution Time, Avg: Average Execution Time, Max Events: Maximum number of scheduling events, Max Pct: Maximum percentage of scheduler execution time in all real-time tasks execution time.

maximum percentage of execution time taken up by the hierarchical scheduling in AvaGPU across varying workloads is 0.82%.

7 SECURITY ANALYSIS

GPU Tasks' Code and Data Integrity: *GPU Task's Code and Data Corruption (R1):* Attackers may modify the code and data of secure GPU tasks to corrupt their executions. However, the codes and data of secure GPU tasks are located within the TEE memory, shielding them from direct modification by REE attackers.

Secure GPU Task Input/Output: *DMA Manipulation (R2):* An attacker could maliciously send commands to the GPU to deny or manipulate the DMA transfers between the host and the GPU buffer. AvaGPU prevents such attacks by checking the destination and length of DMA data transfer transactions with the DMA reference monitor. An attacker cannot deny the allocation of a DMA buffer because the DMA buffers used by secure tasks are pre-allocated within the TEE. The DMA attack transferring data from other untrusted devices to secure memory is prevented by Arm TrustZone.

GPU Access Availability: *Denial/Malicious GPU Access/Configuration (R3):* A REE attacker cannot prevent secure GPU tasks from accessing or configuring the GPU, as the GPU's MMIO memory space is accessed directly from the TEE and is typically predefined for embedded devices. AvaGPU prevents attackers from arbitrarily configuring the GPU by using Stage-2 translation-based memory control. All GPU configuration requests are trapped and sent to the TEE configuration mediator for verification before being written to GPU.

Real-time Execution Availability: *(1) Denial Scheduling GPU Tasks (R4):* All secure CPU tasks are invoked directly by the trusted scheduler within the TEE, preventing attackers from hindering their invocation. AvaGPU's GPU task scheduler, located within the TEE, is shielded from REE attackers' control flow manipulation. Attackers may delay GPU scheduling by triggering REE interrupts. To counter this, a REE interrupt handling task with a fixed periodic budget is introduced. CPU scheduling analysis guarantees that the CPU task will meet its deadline, even if the entire interrupt handling task's budget is consumed. *(2) Delaying GPU Tasks Finishing (R4):* An attacker can delay a secure GPU task by statically disabling the suspension instrumentation, runtime corrupting preemption signals or hijacking control flow. Firstly, AvaGPU prevents suspension instrumentation disabling by verifying the instrumented task's signature, using an encrypted hash signed by a trusted compiler with the public key in the TEE. Additionally, AvaGPU mitigates potential TOCTOU attacks, which may occur after signature verification but before execution, by making GPU task code pages read-only for normal world during Stage-2 translation prior to signature verification. Secondly, an attacker's attempts to modify preemption signals to bypass self-preemption checks are prevented by Stage-2 translation, as the variables are only readable in the REE. AvaGPU detects and eliminates any efforts to circumvent preemption through runtime exploiting vulnerabilities like memory safety bugs, but it doesn't prevent memory safety corruption due to the significant runtime overhead.

GPU Resources Isolation *Denial/Malicious GPU Management (R5):* AvaGPU utilizes a debloated secure driver accessible directly within the TEE, making secure GPU tasks independent of REE GPU management functions. Thus, REE cannot deny GPU management for these tasks. Memory isolation provided by TEE prevents attacks on resource management data structures. Memory management mediator prevents malicious attempts to corrupt trusted resource management in the TEE with invalid requests.

8 RELATED WORK

GPU Execution Environment Isolation: As shown in Table. 8, existing research on building isolated GPU execution environments falls into two categories: GPU virtualization and TEE on GPU.

Through the introduction of a unified additional layer of GPU memory and configuration management in the hypervisor, GPU virtualization [54, 57] enables the isolation of computation environments across VMs on a shared hardware platform, thus eliminating interference between individual VM memory and configuration management. GPU virtualization only provides OS-level isolation, AvaGPU complements this by providing a finer-grained GPU computation isolation, i.e., between trusted and untrusted processes.

Another research line studies how to build TEE for GPU computation. Hardware-based solutions [32, 33, 59, 67] utilize customized hardware to provide integrity and confidentiality for GPU worklaod. Graviton [59] uses a customized GPU command processor to enforce GPU physical memory isolation for different enclaves. Based on Graviton, Telekine[32] transforms GPU computations into a data-oblivious form to defend against side-channel attacks. HIX [33] leverages customized CPU MMU and enclave metadata to enforce the GPU usage isolation. HETEE [67] leverages centralized FPGA-based controller to isolate accelerators physically. AvaGPU complements these work by additionally guaranteeing availability for secure GPU tasks with software-based GPU resource management and access control solutions.

For software-based solutions, GPU separation Kernel (GSK) [65] provides trusted display by isolating trusted GPU drivers in a separated kernel to enforce GPU access control. AvaGPU complements GSK by minimizing TCB of trusted GPU driver. Strongbox [27] implements GPU TEE on Arm platform by leveraging Stage-2 memory translation and Arm Trustzone for isolating GPU task data and code. Different from Strongbox, AvaGPU uses Stage-2 memory translation to provide trusted and efficient GPU memory management. HoneyComb [43] implements a TEE by statically verifying the GPU task binary before loading to confine the behaviors of GPU tasks. CODY [48] and RT-TEE [61] adopt I/O message recording and replaying to provide a trusted peripheral driver. However, GPU access control and dynamic memory management is also necessary when guaranteeing GPU workload availability. Thus, AvaGPU extends

Table 8: GPU TEE Related Work Comparison Table

System	Conf./Inte.	CPU Avai.	GPU Avai.	OS Avai.	C.GPU T. A.	SW
GPUvm [54]	✓	✓	✓	✓		✓
gVirt[57]	✓	✓	✓	✓		✓
Graviton [59]	✓					
HIX [33]	✓					
HETEE [67]	✓					
Trusted Disp [65]	✓					✓
StrongBox [27]	✓					✓
CODY [48]	✓					✓
SecDeep [42]	✓					✓
RTTEE [61]	✓	✓				✓
AvaGPU	✓	✓	✓		✓	✓

Conf: GPU Data Confidentiality, Integrity: GPU Data/Code Integrity, OS Avai: OS-level GPU Compute Availability, C.GPU T. A.: Task-level CPU-GPU Compute Availability SW: Software Solution.

Table 9: Kernel Preemption Related Work Comparison Table

System	Split Kernel	Re-execution	Inter Block	Intra Thread
PKM, GEPS [24, 66]	✓			
Effisha, FLEP [25, 63]			✓	
REEF, Lee et al. [30, 37, 38]		✓		
AvaGPU				✓

message replaying-based drivers by providing a secure and efficient GPU access control and dynamic memory management. From security property perspectives, all above software-based GPU solutions focus on confidentiality and integrity of GPU tasks, AvaGPU supplements them by additionally guaranteeing the availability of secure GPU tasks. RT-TEE [61] presents a solution to ensure real-time availability for CPU tasks. AvaGPU complements RT-TEE by ensuring GPU tasks availability.

Real-time GPU Scheduling: As shown in Table. 9, GPU task scheduling mechanisms are divided into non-preemptive scheduling and preemptive scheduling. The non-preemptive GPU task scheduling solutions [28, 34–36, 52] only schedule GPU tasks after one GPU task finishing. AvaGPU proposes a preemptive-based GPU scheduling mechanism that can schedule the GPU tasks at each preemption checkpoint during a GPU task execution, reducing scheduling delay. GPU doesn't support a preemptive interface in the hardware. Thus, to support preemptive-based GPU scheduling either hardware or software needs to be modified. Customized hardware architectures [49, 56, 62] extend existing GPU hardware to support preemption. However, compatibility issues make it challenging for these solutions to be widely adopted. Software-based GPU preemption mechanisms have three categories. The first approach splits a long execution GPU tasks into multiple short ones and schedules at the end of splitted GPU tasks execution[24, 66]. However, launching sub-tasks introduces high latency. AvaGPU complements this approach by implementing GPU task preemption with software instrumentation without additional GPU task launching. The second approach, thread block-level preemption [25, 63], preempts GPU tasks at the end of a thread block. This method introduces high preemption delay if thread block execution time is long. AvaGPU complements these methods to support preemption at any expected code execution points no matter how long a thread is. The last approach kills and restores impotence workloads [30, 37, 38] without saving context. AvaGPU complements this approach to support any kind of workload. Additionally, AvaGPU complements all above work by securing the scheduling infrastructure and defense against compromised GPU tasks.

9 DISCUSSIONS AND LIMITATIONS

GPU Source Code Requirement: AvaGPU includes a GPU task compiling tool that requires the source code of the GPU task. For binary-only AI applications, it's possible to extend AvaGPU by instrumenting GPU binary code with GPU binary instrumentation tools [58]. However, significant reverse engineering efforts are required. Similar to other security mechanisms [26, 53], modifying the source code of an application can alter its performance. Consequently, secure applications require re-certification procedures and schedulability testing after source code modification.

Availability on Dedicated GPU: AvaGPU is primarily focused on protecting the availability of embedded real-time systems, which typically utilize integrated GPUs. The key distinction between integrated and dedicated GPUs is that integrated GPUs share the same physical memory as the CPU, while dedicated GPUs have their own dedicated physical memory. Though the design of AvaGPU is also applicable to platforms utilizing dedicated GPU, securing various communication channels presents additional challenges, such as addressing malicious PCIe channel configurations.

Suspending/killing Strategies: Current AvaGPU suspends (responsive ones) and kills (non-responsive ones) non-secure GPU tasks upon detection of delay of secure GPU tasks. Such aggressive strategy prevents sequential delays from multiple non-secure GPU tasks at the cost of non-secure world performance. However, depending on the level to tolerance of the control system, AvaGPU also supports other strategies with different trade-offs. For example, AvaGPU can only terminates the non-responsive GPU task without suspending responsive non-secure ones.

Extending to Mutually Untrusted Secure GPU Tasks: While AvaGPU assumes a simplified model of trusted GPU tasks, there are systems where secure GPU tasks may be mutually untrusted. Extending AvaGPU's support for availability protection to such setting requires addressing three new attack vectors. First, malicious secure tasks might disable/bypass preemption, by modifying GPU tasks or exploiting vulnerabilities, to monopolize computational resources. Second, malicious secure tasks can modify computation configurations when other secure GPU tasks are running. Last, they can compromise other tasks' memory via DMA requests targeting another task's memory [27]. To defend against these attacks, AvaGPU can be extended to additionally verify user space requests of GPU configurations and DMA transactions from secure GPU tasks with existing GPU configuration mediators and DMA reference monitors. Additionally, preemption bypassing attacks can be prevented by applying existing delay detection and attack elimination mechanism to secure tasks. To understand the cost of this extension, we implemented a prototype and measured the execution time of secure GPU tasks with the same setup in section 6.2 (i.e., System Overhead on GPU Benchmark). The maximum and average runtime overhead of secure GPU task is 19.32% and 16.40% responsively, 0.82% higher than the baseline AvaGPU.

Remote Attestation: The implementation of AvaGPU adapts the remote attestation mechanism from the embedded GPU TEE solution [27], where the keying materials are stored in the secure storage. Upon receiving a challenge from remote verifier, a signed measurement over the software TCB is returned for verification.

Though attestation is not the focus of this paper, it is one of most foundational techniques for TEE and requires additional investigation in the context of AvaGPU in the future.

Solution Generality: The key features leveraged in AvaGPU, include virtual memory system-based GPU memory isolation, two stage memory translation, and GPU task killing. Virtual memory systems and two stage memory translation are well supported by mainstream GPU vendors, including Nvidia, AMD, Intel and Arm, and CPU architectures, such as Arm, MIPS, and RISC-V. Although GPU task killing are not clearly documented, and may differ by GPU vendors. It's still possible to adapt a less efficient mechanism, such as terminating all GPU tasks and resuming only trusted ones to ensure availability without task level GPU process killing.

10 CONCLUSION

In this paper, we introduce AvaGPU, which ensures real-time availability guarantees for safety-critical GPU tasks. To couple the priority of secure CPU and GPU tasks, AvaGPU proposes a secure real-time CPU-GPU co-scheduling framework, effectively mitigating performance interference. To enable secure and efficient preemptive GPU scheduling, AvaGPU proposes a secure and fine-grained GPU task preemption mechanism, effectively bounding priority inversion. To provide an efficient and trusted GPU driver with minimized TCB, AvaGPU proposes a new splitted GPU driver. We developed a prototype on Jetson AGX Orin platform and evaluated the system with benchmarks, synthetic tasks, and real-world applications. The source code is available at our project repository [1].

ACKNOWLEDGMENT

We thank the reviewers for their valuable feedback. This work was partially supported by the NSF (CNS-1916926, CNS-2038995, CNS-2154930, CNS-2238635), ARO (W911NF2010141), and Intel.

REFERENCES

[1] 2023. AMD Embedded GPU. https://www.amd.com/en/products/embedded.
[2] 2023. AMD ROCm. https://rocm.docs.amd.com/en/latest/.
[3] 2023. Arm Embedded GPU. https://www.arm.com/markets/automotive/autonomous-vehicles.
[4] 2023. Artificial Intelligence & Autopilot. https://www.tesla.com/AI.
[5] 2023. Autoware. https://autoware.org/.
[6] 2023. Autoware Hardware Expectation. https://autowarefoundation.github.io/autoware-documentation/main/installation/.
[7] 2023. Autoware Sample-Rosbag. https://autowarefoundation.github.io/autoware-documentation/main/tutorials/ad-hoc-simulation/rosbag-replay-simulation/.
[8] 2023. BYD Self Driving. https://en.byd.com/news/byd-selects-nvidia-drive-hyperion-\for-next-generation-software-\defined-electric-vehicles/.
[9] 2023. CPS Kernel Vulnerability. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=cyber+physical+kernel+vulnerability.
[10] 2023. DetectNet. https://github.com/Pypearl/ObjectDetectionCuda.
[11] 2023. Drone Camera. https://www.bhphotovideo.com/c/product/802344079-USE/kolibri_xk6600_gy_hellfire_hd_camera_drone.html.
[12] 2023. Image Compression. https://github.com/adolfos94/Haar-Wavelet-Image-Compression.
[13] 2023. ImageNet. https://github.com/dusty-nv/jetson-inference/blob/master/docs/imagenet-console-2.md.
[14] 2023. NIO Self Driving. https://www.nio.com/nad.
[15] 2023. NVIDIA DRIVE. https://developer.nvidia.com/drive/hyperion.
[16] 2023. Nvidia GPU Chennel Reset. https://switchbrew.org/wiki/NV_services#NVGPU_IOCTL_CHANNEL_FORCE_RESET.
[17] 2023. Nvidia Jetson Nano. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/education-projects/.
[18] 2023. Nvidia Multiple Process Service. https://docs.nvidia.com/deploy/mps/index.html.
[19] 2023. Rodinia Benchmark. https://github.com/yuhc/gpu-rodinia.
[20] 2023. SPEC CPU2017. https://www.spec.org/cpu2017/.
[21] 2023. Stage2 Translation. https://developer.arm.com/documentation/102142/0100/Stage-2-translation.
[22] Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. 2021. Aion: Enabling open systems through strong availability guarantees for enclaves. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1357–1372.
[23] Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. 2017. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 104–115.
[24] Can Basaran and Kyoung-Don Kang. 2012. Supporting preemptive task executions and memory copies in GPGPUs. In *2012 24th Euromicro Conference on Real-Time Systems*. IEEE, 287–296.
[25] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. Effisha: A software framework for enabling efficient preemptive scheduling of gpu. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 3–16.
[26] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. ACES: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 18)*. 65–82.
[27] 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.
[28] Glenn A Elliott, Bryan C Ward, and James H Anderson. 2013. GPUSync: A framework for real-time GPU management. In *2013 IEEE 34th Real-Time Systems Symposium*. IEEE, 33–44.
[29] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. 2009. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM* 52, 5 (2009), 91–98.
[30] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *OSDI*. 539–558.
[31] John H Holland. 1992. Genetic algorithms. *Scientific american* 267, 1 (1992), 66–73.
[32] 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.
[33] 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.
[34] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. 2011. RGEM: A responsive GPGPU execution model for runtime engines. In *2011 IEEE 32nd RTSS*. IEEE, 57–66.
[35] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, Yutaka Ishikawa, et al. 2011. TimeGraph:GPU Scheduling for Real-Time Multi-Tasking Environments. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*.
[36] 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.
[37] 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.
[38] Hyeonsu Lee, Jaehun Roh, and Euiseong Seo. 2018. A GPU kernel transaction-ization scheme for preemptive priority scheduling. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 202–213.
[39] 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*. 27–41.
[40] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.
[41] Han Liu et al. 2023. SlowLiDAR: Increasing the Latency of LiDAR-Based Detection Using Adversarial Examples. In *IEEE/CVF CVPR*. 5146–5155.
[42] 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.
[43] Haohui Mai et al. 2023. Honeycomb: Secure and Efficient GPU Executions via Static Validation. In *USENIX OSDI*. 155–172.

[1]Source code is available at https://github.com/WUSTL-CSPL/AvaGPU

[44] Andrea Miele. 2016. Buffer overflow vulnerabilities in CUDA: a preliminary analysis. *Journal of Computer Virology and Hacking Techniques* 12 (2016), 113–120.

[45] Hoda Naghibijouybari et al. 2018. Rendered insecure: Gpu side channel attacks are practical. In *CCS*. 2139–2153.

[46] Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H Anderson, F Donelson Smith, Alex Berg, and Shige Wang. 2017. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 353–364.

[47] Heejin Park and Felix Xiaozhu Lin. 2022. GPUReplay: a 50-KB GPU stack for client ML. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 157–170.

[48] Heejin Park and Felix Xiaozhu Lin. 2023. Safe and Practical GPU Acceleration in TrustZone. *EuroSys* (2023).

[49] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative preemption for multitasking on a shared GPU. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 593–606.

[50] Linh TX Phan et al. 2011. CARTS: a tool for compositional analysis of real-time systems. In *SIGBED Review*. ACM.

[51] Bharath Pichai and other. 2014. Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 743–758.

[52] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 233–248.

[53] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. 2020. OAT: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1433–1449.

[54] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2014. GPUvm: Why Not Virtualizing GPUs at the Hypervisor?. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 109–120.

[55] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 48–62.

[56] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling preemptive multiprogramming on GPUs. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 193–204.

[57] Kun Tian, Yaozu Dong, and David Cowperthwaite. 2014. A Full GPU Virtualization Solution with Mediated Pass-Through. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 121–132.

[58] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. 2019. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 372–383.

[59] 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.

[60] Jinwen Wang et al. 2023. ARI: Attestation of Real-time Mission Execution Integrity. In *USENIX Security*. 2761–2778.

[61] 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 S&P*. IEEE Computer Society, 1573–1573.

[62] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 358–369.

[63] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. 2017. Flep: Enabling flexible and efficient preemption on gpus. *ACM SIGPLAN Notices* 52, 4 (2017), 483–496.

[64] Tianwei Yin, Xingyi Zhou, and Philipp Krahenbuhl. 2021. Center-based 3d object detection and tracking. In *IEEE/CVF CVPR*. 11784–11793.

[65] Miao Yu, Virgil D Gligor, and Zongwei Zhou. 2015. Trusted display on untrusted commodity platforms. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 989–1003.

[66] Husheng Zhou, Guangmo Tong, and Cong Liu. 2015. GPES: A preemptive execution system for GPGPU computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 87–97.

[67] 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.

A ADDITIONAL DESIGN DETAILS

GPU Preemption Algorithm: The algorithm, which utilizes a 4-dimensional DP table, calculates the minimum number of tasks to preempt to meet specific resource needs (including memory, registers, and threads) for the next GPU task. It takes into account the number of running tasks with the same priority and iterates through all task and resource combinations, determining the minimum number to preempt by including or excluding the current task. Once the table is built, it backtracks to identify the specific tasks preempted in the optimal solution.

Algorithm 1: Task Preemption Algorithm

```
Input  : Number of existing GPU tasks with the same priority $n$. Tasks with the same
         priority. $T = \{\tau_i = (m_i, r_i, t_i) | 0 \le i < n\}$, $m_i, r_i, t_i$ is the required memory,
         registers, and threads of $\tau_i$. A new task $\tau_w = (m_w, r_w, t_w)$ to be executed.
Output : Tasks to be preempted $T' = \{\tau_i' | 0 \le i < n\}$
for $i = 0$ to $n$ do
    $dp[i][0][0][0] \leftarrow 0$
for $i = 0$ to $n$, $m = 0$ to $m_w$, $r = 0$ to $r_w$, $t = 0$ to $t_w$ do
    $dp[i][m][r][t] = dp[i-1][m][r][t]$
    if $m >= m_{i-1}$ and $r >= r_{i-1}$ and $t >= t_{i-1}$ then
        $dp[i][m][r][t] =$
          $\min(dp[i][m][r][t], 1 + dp[i-1][m - m_{i-1}][r - r_{i-1}][t - t_{i-1}])$
$T' = \{\}$
for $i = n$ to 1 do
    if $dp[i][m_w][r_w][t_w]$ != $dp[i-1][m_w][r_w][t_w]$ then
        add $\tau_{i-1}$ to $T'$
        $m_w$ -= $m_{i-1}$, $r_w$ -= $r_{i-1}$, $t_w$ -= $t_{i-1}$
return $T'$;
```

Algorithm 2: Defense Strategy Optimization

```
Input: Deadline of CPU tasks $\{d_i\}$, execution time of CPU tasks $\{e_i\}$,
  system utilization upper bound $U_{up}$, maximum iteration number $c_{max}$,
  selected offspring number $K$, number of crossover rounds $N$, $p_m$ is
  mutation probability.
Output: Attack detection points $N$, attack detection position $C$
Randomly perturb $N$, $C$ into initialization groups $\mathrm{N}_0$, $\mathrm{C}_0$
for $c = 0$ to $c_{max} - 1$ do
    Calculate fitness score $S$ of $\mathrm{N}_c$ and $\mathrm{C}_c$, by Eq. 3 in the constraints of Eq.
      1 and Eq. 2;
    $\mathrm{N}_c, \mathrm{C}_c \leftarrow argsort(S)[:K]$, $S_c \leftarrow sort(S)[:K]$;
    for $i = 1$ to $N$ do
        $parent_1, parent_2 \leftarrow Sample(\mathrm{N}_c, \mathrm{C}_c, \mathrm{S}_c)$
        $child_i \leftarrow Crossover(parent_1, parent_2)$
        $N_i, C_i \leftarrow Mutate(child_i, p_m)$
        $\mathrm{N}_{c+1}, \mathrm{C}_{c+1} \leftarrow \{N_i\}, \{C_i\}$
return $\mathrm{N}_{c_{max}}, \mathrm{C}_{c_{max}}$;
```

Preemption Bypassing Defense Strategy Optimization: AvaGPU utilizes the genetic algorithm [31] to address the preemption bypassing attack defense strategy optimization. It begins by randomly selecting N strategies and iteratively refines them for reduced runtime overhead. The fitness score, representing the strategy's runtime overhead, is calculated using Equations 1, 2, and 3. In each iteration, the top K strategies are chosen as offspring based on their scores. New strategies are then produced via crossover, where elements in N and C from two parent strategies are exchanged, followed by potential mutations with a probability of p_m. This process continues until a satisfactory offspring emerges. Details can be found in Alg. 2. To assess the convergence of our optimization algorithm, we tested it on three task sets with system utilizations of 20%, 40%, and 60%. Each set had three tasks with random execution times. Results showed the algorithm converged in fewer than 2000 iterations for all sets, and reduced runtime overhead from 43.98% to under 3% of task execution time.