CRONUS: Fault-isolated, Secure and High-performance Heterogeneous Computing for Trusted Execution Environment

Jianyu Jiang[1], Ji Qi[1], Tianxiang Shen[1], Xusheng Chen[1], Shixiong Zhao[1],
Sen Wang[2], Li Chen[2], Gong Zhang[2], Xiapu Luo[3], Heming Cui[1,4,*]
[1]*The University of Hong Kong, Hong Kong, China, Email: {jyjiang, jqi, txshen, xschen, szhao, heming}@cs.hku.hk*
[2]*Huawei Technologies, Hong Kong, China, Email: {wangsen31, chen.li7, nicholas.zhang}@huawei.com*
[3]*The Hong Kong Polytechnic University, Hong Kong, China, Email: csxluo@comp.polyu.edu.hk*
[4]*Pujiang Lab, Shanghai, China*

***Abstract*—With the trend of processing a large volume of sensitive data on PaaS services (e.g., DNN training), a TEE architecture that supports general heterogeneous accelerators, enables spatial sharing on one accelerator, and enforces strong isolation across accelerators is highly desirable. However, none of the existing TEE solutions meet all three requirements.**

In this paper, we propose CRONUS, the first TEE architecture that achieves the three crucial requirements. The key idea of CRONUS is to partition heterogeneous computation into isolated TEE enclaves, where each enclave encapsulates only one kind of computation (e.g., GPU computation), and multiple enclaves can spatially share an accelerator. Then, CRONUS constructs heterogeneous computing using remote procedure calls (RPCs) among enclaves. With CRONUS, each accelerator's hardware and its software stack are strongly isolated from others', and each enclave trusts only its own hardware. To tackle the security challenge caused by inter-enclave interactions, we design a new *streaming remote procedure call* abstraction to enable secure RPCs with high performance. CRONUS is software-based, making it general to diverse accelerators. We implemented CRONUS on ARM TrustZone. Evaluation on diverse workloads with CPUs, GPUs and NPUs shows that, CRONUS achieves less than 7.1% extra computation time compared to native (unprotected) executions.

***Keywords*-ARM TrustZone; Trusted Execution Environment; Accelerator; GPU; Fault isolation; Security isolation**

I. INTRODUCTION

The emerging Platform as a Service (PaaS) paradigm enables users to run diverse cloud applications (e.g., DNN training and inference [88], [34]) without knowing the details of software and hardware stacks. For high performance, PaaS typically embraces diverse accelerators (e.g., GPUs [82] and NPUs [42], [28]) to accelerate heterogeneous computation.

Recently, to protect sensitive user data in clouds, Trusted Execution Environment (TEE) is becoming the de-facto technique. TEE (e.g., TrustZone [13] and Intel SGX [50]) provides confidentiality and integrity for user data with a secure execution environment called enclave (or Trusted App[1]), which cannot be accessed or tampered with even by privileged attackers such as operating systems. TEE maintains a low computation overhead compared to native (unprotected) computation, making TEE attractive for protecting data [98], [99], [125], [56], [6] in PaaS (e.g., Azure Confidential Computing [18]).

In this paper, we summarize three crucial requirements for enabling TEE in PaaS to securely process sensitive user data. First (**R1**), the growing number of accelerators designed and integrated into a cloud server (e.g., Amazon F1 [9] and NVIDIA DGX [83]) provide holistic performance gains (e.g., low latency and high throughput), making it critical for supporting *general* accelerators in TEE without hardware customization. This is because hardware customization is difficult for verifying correctness (§II-B) and is not generic to accommodate new accelerators. Second (**R2**), it is essential to enable *spatial sharing*, where different users share the same accelerator for low costs (i.e., high utilization and profits for clouds). In contrast, preserving a dedicated accelerator for a single tenant often results in extremely low utilization (e.g., 10%) on every accelerator [118], [117].

Third (**R3**), it is extremely important to enable *strong isolation* between accelerators. First, long-term analysis [45], [111] on server failures shows that accelerator-supported servers are susceptible to generating 10X more faults than CPU-only servers. Hence, it is crucial to ensure *fault isolation* (**R3.1**): any faults of the software and hardware stack in an accelerator should be isolated from other accelerators. Second, accelerators' flexible programmability (e.g., FPGA) makes it possible for a malicious user injecting code to compromise other accelerators co-located in the same server [128], [62], [7], [41], [16], [75]. Therefore, it is extremely important to gain *security isolation* (**R3.2**): the software and hardware stack of one accelerator shall be securely isolated from others', and users trust only software and hardware stack of used accelerators.

Unfortunately, despite tremendous efforts for integrating accelerators into TEE, none of the existing approaches (classified as *hardware-based* and *software-based* approach, see Figure 1) meets all three requirements. Specifically, the hardware-based approach (e.g., CURE [20] and Gravi-

* Heming Cui is the corresponding author.
[1]We use enclave and trusted app interchangeably in this paper.

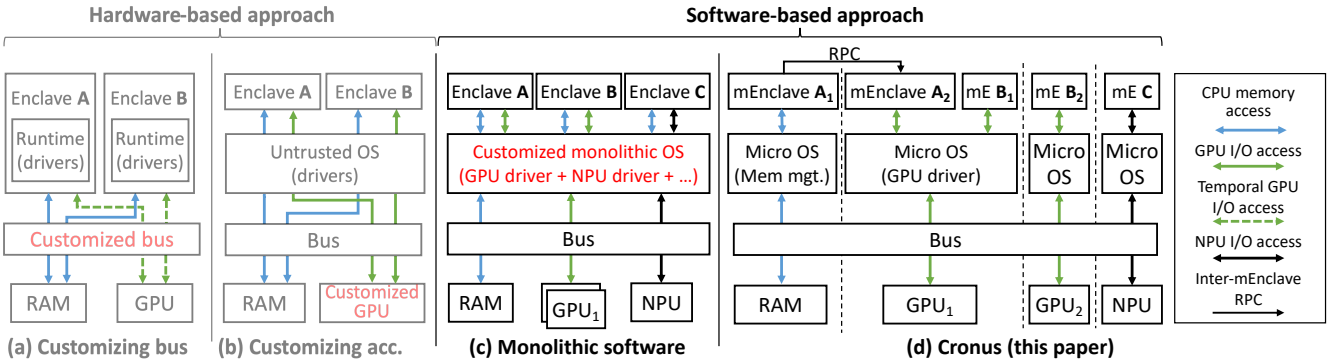

Figure 1: Comparison of existing approaches for supporting accelerators in TEE. Hardware-based approaches include (a) customizing bus (e.g., CURE [20], HETEE [127] and HECTOR-V [81]) and (b) customizing accelerators (e.g., Graviton [109] and SGX-FPGA [116]). (c) Software-based approaches can support multiple accelerators using a trusted monolithic kernel (e.g., TrustZone [13] and SeCloak [71]). The normal OS (untrusted) in (a), (c) and (d) is omitted for brevity. Components in red require manual customization.

ton [109]) customizes I/O bus (Figure 1a) or hardware accelerators (Figure 1b) to enforce isolation among enclaves, yet doing so requires extensive TEE [54] or accelerator specific [109] hardware customizations (violating **R1**). On the other hand, the software-based approach typically leverages the unmodified hardware and customizes only the software runtime [13] to form a monolithic design, which runs all accelerator software in the same address space without isolation (violating **R3**, see Figure 1c).

In this paper, we argue that the root impediment to meeting the three requirements lies in the monolithic design adopted by existing hardware- and software-based systems (§II-B). Inspired by the microkernel paradigm [61], [22], [79], we eliminate the monolithic design and present MicroTEE (Figure 1d), a new TEE architecture that meets the three crucial requirements to enable TEE-compatible secure heterogeneous computing on these accelerators.

MicroTEE partitions a monolithic enclave running heterogeneous computing tasks on different hardware (e.g., CPU-GPU coordinated DNN training) into several micro enclaves (mEnclaves) and partitions the monolithic OS into several micro OSes (mOSes). Specifically, each mEnclave encapsulates one type of computation (e.g., GPU computation) of the heterogeneous computing task, and each mOS is a shim trusted runtime that manages only one accelerator; each mOS includes only enclave and hardware management logic (e.g., drivers) for sharing hardware resources among mEnclaves. MicroTEE constructs a heterogeneous computing task using remote procedure calls (RPCs) between mEnclaves.

In order to strongly isolate each mOS, MicroTEE contains two aspects. First, MicroTEE isolates different mOSes (and its managing accelerator) by leveraging existing hardware isolation techniques (e.g., ARM S-EL2 [105]). Second, MicroTEE eliminates a core OS kernel (used in microkernels) that coordinates different mOSes by replicating basic OS functionalities (e.g., RPC) in all mOSes; MicroTEE offloads OS coordinations (e.g., enclave scheduling) out of the TEE.

The MicroTEE architecture meets the three crucial requirements simultaneously. First, MicroTEE meets **R1** by its software-based architecture, which is applicable to diverse accelerators. Second, MicroTEE meets **R2** with the help of the mOS applying intra-accelerator isolation (e.g., NVIDIA MIG [84]) to spatially share an accelerator across mEnclaves. Third, more importantly, MicroTEE enables **R3** with its microkernel-inspired design. Specifically, any faults in an mEnclave or mOS are constrained within its security boundary: after a fault happens in one mOS or one mEnclave running on an accelerator, MicroTEE can simply restart only this accelerator (**R3.1**). Furthermore, a heterogeneous computing task trusts only the hardware (e.g., CPU and GPU) and mOSes being used, instead of all accelerators with their mOSes running on the same server (**R3.2**). For example, a task running on a CPU and GPU does not need to trust the NPU hardware and mOSes managing NPUs.

One main challenge arises when implementing the MicroTEE architecture. Specifically, frequent RPCs between mEnclaves can significantly downgrade performance during execution and violate **R3** during failure. Specifically, RPC usually requires context switches to make the remote mEnclave active, and recent work [72] shows that context switches within TEE can result in prohibited performance overhead. Worse, an attacker can actively reorder and replay RPCs to compromise the integrity of heterogeneous computing [101], [54] (§II-C) or maliciously generate failures to an mOS/mEnclave to substitute a trusted mOS/mEnclave to a malicious one, gaining sensitive user data sent through RPC (§IV-D).

We tackle the challenge and present CRONUS[2], the first TEE system that realizes the MicroTEE architecture. CRONUS tackles the challenge with a new *streaming remote procedure call (sRPC)* protocol that mitigates frequent context switches and does RPC over shared trusted TEE memory to resist reorder and replay attacks. To tackle malicious attacks during failures, sRPC carries a *proceed-trap*

[2]Cronus is the name of an ancient Greek god.

failure recovery procedure that proceeds execution without downtime upon a failure and traps for access with potential data leakages to prevent leakages through failed mOSes/-mEnclaves (§IV-D). CRONUS's implementation includes a special attestation protocol for users attesting TEE with different accelerators and enables direct integration of off-the-shelf accelerator drivers with a shim mOS library that provides basic OS functions (e.g., memory mapping).

We evaluated CRONUS on CPU, a state-of-the-art NVIDIA GPU (GTX 2080) and a VTA NPU [107]. We evaluated CRONUS on two notable microbenchmarks `rodinia` [96] and `vta-bench` [110] for GPU and NPU, and also evaluated CRONUS on real-world DNN training using PyTorch on GPU, and DNN inference using TVM [107] on NPU [107]. We trained four state-of-art DNNs (i.e., LeNet [66], RestNet [47], DenseNet [48], and VGG [88]) on three popular datasets (i.e., MNIST [66], Cifar [63] and ImageNet [31]) using PyTorch, respectively. Our evaluation and analysis show that:

- **R1:** CRONUS is fast on general accelerators. CRONUS incurs less than 7.1% extra computation time on diverse workloads computed on CPU, GPU and NPU.
- **R2:** CRONUS enables spatial sharing of accelerators. Compared with an accelerator used by a dedicated mEnclave, an accelerator spatially shared by multiple mEnclaves has an up to 63.4% higher throughput.
- **R3.1:** CRONUS is fault-isolated across accelerators. CRONUS recovers from an accelerator failure by restarting only the fault-inducing accelerator's mOS (in hundreds of milliseconds), instead of rebooting the whole machine (in minutes), including all non-faulting accelerators.
- **R3.2:** CRONUS is security-isolated. A PaaS application needs only to trust the code of its mEnclaves being used by itself and the mOSes that manage these mEnclaves; the application does not need to trust other mOSes and mEnclaves running other applications on the same server.

The main contribution of the paper is MicroTEE, the first TEE architecture design that enables fault-isolated and security-isolated heterogeneous computing on general, spatially shared accelerators using existing hardware primitives. MicroTEE's design can be applied to diverse TEE hardware (e.g., ARM TrustZone S-EL2 and RISC-V [68]) with secure I/O (§VII-A). Other contributions include the sRPC protocol that reduces the cost of secure inter-mEnclave RPCs, and enables fast, secure and independent failure recovery. The strong isolation (**R3**) properties of CRONUS make it extremely suitable for supporting diverse accelerators (e.g., TPU [42]) and diverse applications [121], [94].

II. BACKGROUND

A. Trusted Execution Environment (TEE)

TEE [23], [39], [30], [113], [26], [68], [123], [124], [56], [100] typically provides an isolated execution environment called enclave, which cannot be seen or tampered with by malicious attackers [93]. Enclaves provide a new attestation protocol for the user to verify the code running within the enclave. The enclave abstraction can be provided by Hardware (e.g., Intel SGX) or by the co-design of hardware-software co-design (e.g., TrustZone [13], Komodo [39] and CURE [20]). An application using TEE is usually partitioned into two parts: a trusted part that processes sensitive user data and runs within an enclave, and an untrusted part that runs outside the enclave.

ARM TrustZone provides two worlds: a normal world (untrusted) that runs an off-the-shelf untrusted operating system (e.g., Linux) and applications, and a secure world (trusted) that is fully isolated. TrustZone supports four secure privilege levels (i.e., EL0 $\sim$ EL3). At EL3, a secure monitor is running for secure context switching, interrupt handling and secure boot. At EL1, a trusted OS is running to provide the enclave abstraction with isolation and provides basic OS functionalities such as memory mapping and signal handling.

TZASC/TZPC (ARM TrustZone). To isolate memory access between the secure world and normal world, TrustZone makes use of a chip called TrustZone Address Space Controller (TZASC) connected to a DRAM, which configures if a memory region is *secure*. Then, access from the normal world to a secure memory region is filtered. Similarly, for IO access, TrustZone makes use of an APB TrustZone Protection Controller (TZPC) chip for configuring if the normal world can access a given I/O device.

ARM Secure EL2 (S-EL2) Extension. From ARMv8.4, TrustZone provides virtualization supports for the secure world, implemented using a stage-2 page table. S-EL2 runs a trusted OS and applications within a *partition*, and partitions are managed by a secure partition manager (SPM) that runs as the hypervisor in the secure world and isolates physical resources (e.g., memory and devices) into different partitions.

B. The Motivation of CRONUS

Table I summarizes existing work for supporting accelerators in TEE, which is classified into two categories: *Hardware-based* approach and *Software-based* approach.

The hardware-based approach [20], [127], [54], [109], [116] customizes either the I/O buses or the accelerator hardware to ensure secure access between an enclave and an accelerator. On the one hand, several systems [20], [127], [81] implement access control logic in the I/O bus to ensure dedicated accelerator access for enclaves. These systems implement course-grained enclave-accelerator mappings with strong isolation among accelerators (**R3**) and support only specific types of accelerators (e.g., AXI-based accelerators). However, customizing buses requires excessive efforts for modifying the hardware and ensuring backward compatibilities (violating **R1**), which requires collaborations from diverse hardware vendors (e.g., accelerators and PCs) and software developers. Worse, since access control is conducted

System	R1: Supporting Accelerators	R2: Spatial Sharing of Acc.	R3: Strong Isolation					RPC
			R3.1: Fault Isolation (FI)			R3.2: Security Isolation (SI)		
			FI:Enclave	FI:Kernel/Acc.	FI:Fault Recovery	SI:Enclave	SI:Kernel/Acc.	
HARDWARE-BASED								
HETEE [127]	PCIe Acc.	✗[1]	✓	✓	Slow[2]	✓	✓	✗
CURE [20]	AXI Acc.	✗[1]	✓	✓	Slow[2]	✓	✓	✗
HIX [54]	GPU	✓	✓	N/A[3]	Slow[2]	✓	N/A[3]	Slow[5]
Graviton [109]	GPU	✓	✓	N/A[3]	Fast[4]	✓	N/A[3]	✗
SGX-FPGA [116]	FPGA	✓	✓	N/A[3]	Fast[4]	✓	N/A[3]	✗
SOFTWARE-BASED								
Panoply [101]	✗	N/A	✓	N/A[3]	Fast	✓	N/A[3]	Slow[5]
EnclaveDB [91]	✗	N/A	✓	N/A[3]	Fast	✓	N/A[3]	Fast[6]
TrustZone [13]	Generic	✓	✓	✗	Slow[2]	✓	✗	Fast[7]
Ji et. al. [55]	✗	N/A	✓	✓	Fast	✓	✗	Fast[7]
CRONUS (ours)	**Generic**	✓	✓	✓	**Fast**	✓	✓	**Fast**

Remarks:
[1] Support temporal sharing but require cold-reboots of the accelerator when switching among enclaves.
[2] Require cold-reboot to clear accelerator states upon failures.
[3] FI and SI are not applicable when supporting one or no accelerators.
[4] Require implementing failure recovery at hardware.
[5] Uses encryption and acknowledgements for RPC, executing in lock-step.
[6] Defend against attacks at the application level.
[7] Use monolithic OS for managing sharing memory.

Table I: Comparison between CRONUS and related work.

at the bus level, they are not aware of the accelerators' internal states to support spatial sharing and support only temporal sharing of accelerators between enclaves (violating **R2**): each enclave take turns to gain dedicated access to an accelerator, even though each of them takes up only few resources (e.g., 5GB out of 32GB GPU memory).

On the other hand, other systems (e.g., Graviton [109]) customize accelerator hardware with enclave management logic for secure access from an enclave to an accelerator. Customizing accelerators enables spatial sharing (**R2**) within the accelerator and fault isolation (**R3.1**) across accelerators, without trusting the driver for managing accelerators (**R3.2**). However, customizing accelerator hardware adds significant complexities (some may be vulnerabilities [21]) to the (monolithic) hardware and usually requires TEE-accelerator co-design (violating **R1**), making it time-consuming to verify the correctness of hardware [21], [78].

The software-based approach (e.g., TrustZone [13]) enforces resource isolation for enclaves by exploiting existing unmodified hardware isolation primitives and customizing software runtime with spatial sharing of hardware resources (**R2**). By integrating all existing software customizations into a monolithic OS, this approach can easily support diverse accelerators (e.g., GPU and NPU) in TEE (**R1**). However, because of the monolithic design, this approach is neither fault-isolated nor security-isolated (violating **R3**).

Ji et al. [55] apply the microkernel [61], [22], [79] paradigm to TEE to isolate faults among OS kernel components.CRONUS is different from microkernel-based approaches in two aspects. First, a fault in the core kernel that coordinates different kernel components still cause failures of the whole OS (violating **R3.1**). More importantly, the OS components are mutually trusted, so a malicious OS component is capable of acquiring sensitive information stored in other OS components; hence an enclave still has to trust the core OS kernel and all its components (violating **R3.2**). Nevertheless, it is not designed for supporting accelerators within TEE.

Recently, some TEE work [72], [10], [15], [53] provides confidential virtual machines by running a VM directly in a TEE. We believe that VM-based is not suitable for PaaS as a VM can hardly enable spatial sharing of accelerators (violating **R2**), because this requires expensive accelerator virtualization at the hypervisor [120], [104], and virtualizing multiple accelerators at the hypervisor results in monolithic design (violating **R3**). Worse, the bootup time of a VM is too long for short-duration tasks (e.g., DNN inference) in PaaS.

The motivation of CRONUS. So far, no existing work meets all three crucial requirements simultaneously, and we owe it to the conundrum in determining the software/hardware that runs hardware managing logic (e.g., access control between enclaves and accelerators). First, **R1** demands no hardware customization and implies that the managing logic should not be at hardware, excluding accelerator customization in the hardware-based approach. Second, **R2** demands the managing logic aware of accelerator details (as spatial sharing mechanisms [84], [85] are accelerator-specific), so it should be at either OS software or at accelerator hardware, excluding bus customization in the hardware-based approach. Third, **R3** demands inter-accelerator isolation (i.e., no accelerator details), indicating that the managing logic should ignore accelerator details and excluding software-based monolithic approach.

From this point of view, **R2** and **R3** seem to be fundamentally conflicting as one demands accelerator details while the other does not. Yet, this is true only when all hardware managing logic can only be run monolithically. CRONUS resolves the conflict by splitting the managing logic into

two parts: it firstly enforces inter-accelerator isolation (for **R3**) by separating and isolating each accelerator's managing logic into mOS using strong isolation techniques (e.g., ARM S-EL2) without knowing the accelerator details; it then enforces intra-accelerator isolation (for **R2**) at each mOS aware of the accelerator details, without hardware customization (for **R1**). Separating managing logic in CRONUS requires tackling challenges of secure and efficient coordination between the two parts and among mOSes during execution and failover (§II-C, §IV).

C. Preserving Inter-enclave Integrity

Previous work [77], [101], [11] shows that interaction among enclaves can be easily manipulated by malicious OS attackers to compromise the integrity of enclave computation. For example, an attacker can reorder, replay and drop RPC among two enclaves to cause incorrect computation results (i.e., compromising integrity). To prevent these attacks, existing work can be classified into two categories: the synchronous approach [11], [101] and the asynchronous approach [77].

The synchronous approach [101] typically adopts encryption (e.g., SSL) for RPC requests (sent via untrusted memory) to resist reorder and replay and uses acknowledgements to defend against dropping, resulting in lock-step RPC execution. Yet, this approach is costly as it requires synchronous and serial execution of all RPC, unsuitable for frequent or asynchronous inter-enclave RPCs. Note that TrustZone [13] can enable secure synchronous RPC via trusted share memory, but its monolithic design gives up fault isolation (**R3.1**).

On the other hand, the asynchronous approach [77], [91] does not enforce any execution order in the RPC protocol and defends reply, reorder and dropping attacks at application logic. Specifically, this approach typically uses monotonic counters and merkel trees to keep globally consistent application states. This approach is efficient as it does not need lock-step RPC execution, yet it requires a tremendous amount of development efforts from application developers.

Different from previous work, CRONUS enables secure and fast RPC without lock-step RPC execution and requires no application development efforts by constructing trusted shared memory between mEnclaves (§IV-C). CRONUS mitigates context switches for RPCs during execution (§IV-C) and defends against diverse active attacks during failover (§IV-D).

III. OVERVIEW

A. Architecture

Figure 2 shows the architecture of CRONUS. In CRONUS, there are two worlds: a secure world (trusted) and a normal world (untrusted). CRONUS's secure world has multiple partitions, and each partition runs as a strongly isolated ARM S-EL2 partition (**R3**) managed by the *Secure Partition Manager* (§V). Each partition runs a *MicroOS* image provided

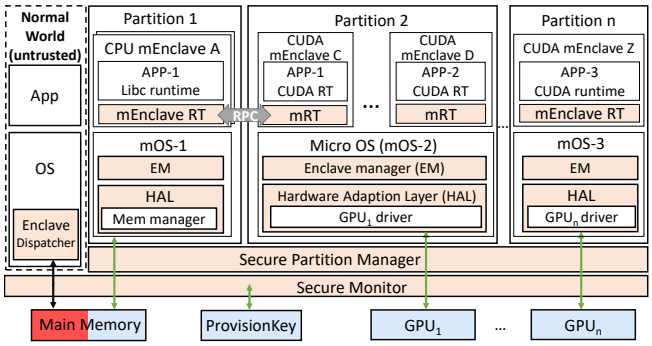

Figure 2: Architecture of CRONUS. CRONUS's components are in orange. The normal world components are untrusted and have access to a subset of non-secure main memory (red). All other components are in the secure world and are trusted. Hardware in blue can only be accessed from the secure world.

by the normal world's OS and runs multiple *MicroEnclaves* on top of a device. The device can be either a CPU or an accelerator (e.g., a GPU or an NPU). Note that each device is managed by only one partition, and vice versa.

CRONUS's normal world runs a full-fledged OS (e.g., Linux), applications and *Enclave Dispatcher*. Enclave Dispatcher determines which partition is used to handle an mEnclave request from an application. Moreover, the Enclave Dispatcher records the device type and configurations, mOS images, and usable resources in each partition. In the secure world, each partition runs two layers of components as follows.

MicroEnclave (mEnclave) runs a user-level (typically `EL0`) application code processing sensitive data (§IV-A). CRONUS carries an mEnclave abstraction that unifies executions for different devices. Each mEnclave contains a runtime that defines the mEnclave execution logic and loads mEnclave images (if any). mEnclave executes code on a device (e.g., CPU and GPU), and the device may be shared by multiple mEnclaves in the same partition (e.g., Partition 2 in Figure 2, meeting **R2**). mEnclave uses inter-enclave RPC to construct heterogeneous computing, and CRONUS's sRPC protocol ensures strong isolation (**R3**) during execution (§IV-C) and failover (§IV-D). Each mEnclave has a static list of `mECalls` that can be called by another mEnclave or the application in the normal world.

MicroOS (mOS) runs an Enclave Manager (§IV-A) and a Hardware Adaptation Layer (HAL, see §IV-B). The Enclave Manager is responsible for loading and initializing an mEnclave, providing device-independent functionalities such as attestation and inter-enclave RPC. HAL is responsible for configuring, accessing, attesting and virtualizing hardware resources (e.g., GPU Memory) for different mEnclaves. HAL makes use of intra-accelerator isolation techniques (e.g., NVIDIA's MPS [85]) to efficiently and securely share an accelerator among mEnclave (**R2**). HAL includes a shim core library that implements basic kernel functions (e.g., memory mapping and locks), so HAL can integrate off-the-

shelf device drivers in Linux to support general accelerators (**R1**). Note that mOSes are run upon system startup, so mEnclaves do not need to wait for their bootups.

B. Threat and Failure Model

Threat model. CRONUS shares the same threat model used in other commonly used TEE architecture [20], [30], [72], [13], [68]. Specifically, CRONUS trusts the TEE hardware (i.e., TrustZone), bus implementation, and CRONUS's software components in the secure world. An S-EL2 partition (mEnclaves, the mOS and the accelerator) may be malicious and tries to attack other S-EL2 partitions by running malicious code; the partition may be faulty because of buggy code/hardware. Since all partitions are isolated from each other (Figure 2), a malicious/buggy partition (e.g., a maliciously configured accelerator) cannot compromise any other partitions. Therefore, a PaaS service involving multiple partitions trusts only its used partitions and attests the integrity of these partition's code and hardware (§IV-A). Note that different PaaS services may use different mOS (even for the same type of accelerators) for optimizing performance [74], and each service trusts only its used version of mOS in CRONUS, instead of trusting all versions of mOSes running at same machine in the monolithic OS.

Failure model. Similar to previous TEE systems [127], [54], we assume crash safety [27], i.e., the security properties of the system are still maintained when the system is crashed due to software bugs, hardware errors, or malicious attacks (by the malicious normal OS). Hence, an mOS can fail arbitrarily at any time. Further, a fault in a partition (e.g., a GPU mOS) is isolated within the fault-inducing partition because of S-EL2, so it will not crash the whole system (**R3.1**). After crashes, the system recovers (e.g., software recovery) and continues serving new requests without compromising safety. CRONUS does not recover application data and can integrate techniques for recovering application data [29], [1], [40]) for this purpose.

In-scope attacks. Same as TrustZone [13], software running in the normal world such as hypervisors, privileged OS and malicious software are not trusted. Attackers can try to modify the system configurations by privileged OS or software. They can also manipulate network packets, hijack system calls, and invoke a `mECall` with arbitrary parameters. An mEnclave may contain malicious code trying to attack the normal OS, or other mEnclaves to sabotage CRONUS's guarantee.

We identify all possible attacks present in CRONUS's new TEE architecture under CRONUS's threat model. The untrusted normal OS may maliciously dispatch an mEnclave request (e.g., `mECall`) to an incorrect partition (tackled by CRONUS's ownership assurance in §IV-A); it can also misconfigure an accelerator or configure a fabricated accelerator into the secure world (tackled by CRONUS's attestation protocol in §IV-A); it can reorder and replay RPCs between mEnclaves (tackled by CRONUS's sRPC protocol in §IV-C); it can also interrupt or terminate the execution of the mOS or mEnclaves running in a partition, or even substitute the mOS with a malicious one to compromise isolation across accelerators (tackled by CRONUS's sRPC failover procedure in §IV-D).

Out-of-scope attacks. Same as previous TEE work [20], [30], [13], CRONUS does not handle denial of Service (DoS) attacks, timing attacks [17] of a `mECall`, rollback attacks of sealed data [77], architecture side-channel attacks [80], [112], [70], [36] and physical attacks (e.g., bus snooping [67]). In the future, these attacks can be tackled by integrating existing defenses [8], [77], [51], [14], [32], [25], [114], [44] into CRONUS; CRONUS does not handle buggy/malicious code (e.g., malicious libraries in the open-sourced software supply chain [115]) within a PaaS service's trusted partitions (buggy/malicious code within the service's untrusted partitions is isolated, see §III-B), CRONUS can tackle them by isolating and verifying potentially buggy/-malicious code [102], [87], [73], [60], [57].

C. Hardware Assumptions

CRONUS leverages four widely used hardware primitives in TEEs (e.g., TrustZone) and co-designs them with the mTEE architecture (§I) for the three crucial requirements (§I). We will discuss how to adapt CRONUS to other TEEs (§VII-A).

- Isolation. CRONUS leverages a common TEE primitive [50], [13], [68] that provides an isolated execution environment (e.g., secure world), whose resources (e.g., memory region and I/O devices) cannot be directly accessed by the normal execution environment (e.g., privileged OS and hypervisors). CRONUS also utilizes widely existing secure world isolation techniques (e.g., TrustZone S-EL2 [13] and RISC-V PMP [68]) for isolating partitions in the secure world.
- Hardware Root of Trust (RoT). CRONUS leverages hardware RoT in TEE and accelerators for attestation (i.e., running on a real TEE platform and accelerators, see §IV-A). Specifically, the TEE and accelerator vendors utilize a hard-coded private key (e.g., a secret stored in ROM) for signing a measurement report (§IV-A). This feature exists in cloud-based TEEs (e.g., SGX [50], TrustZone [13] and TDX [53]) and many hardware accelerators (e.g., GPU [19] and NPU [119]).
- SecureIO. CRONUS leverages a secure I/O bus for TEE to ensure the trusted world's dedicated access to accelerators connected to the I/O bus, same as previous systems using I/O peripherals [20], [81], [71]. This can be realized using ARM TrustZone's TZPC [13], RISC-V's PMP [68] (§VII), or a customized secure I/O bus such as HIX [54].
- Shared TEE memory. CRONUS leverages shared TEE memory for fast RPCs between mEnclaves. This primitive can be enabled in TrustZone [13], RISC-V-based TEE [81],

[68], [38] and Intel TDX [53] (see §VII). Nevertheless, we found that naively using the primitive for RPC is neither secure nor efficient (§IV), and Cronus carries an sRPC protocol for fast and secure RPC during execution (§IV-C) and failover (§IV-D).

D. An Application's Workflow

Here we would give a brief introduction of the lifecycle of an application (`App-1`) using TEE for protecting sensitive data, as shown in Figure 2. First, a user submits `App-1` built with a manifest (with all mEnclaves' descriptions, see Figure 3) for processing sensitive user data. `App-1` is started and then creates a CPU mEnclave (`mEnclave A`) in partition 1 and initializes it. After remote attestation to `mEnclave A` and the manifest, the user can provide encrypted sensitive user data to `App-1` (untrusted), and `App-1` first calls the `mEnclave A`'s `mECall` with the encrypted data, and `mEnclave A` decrypts and then starts processing on the decrypted data. During execution, the `mEnclave A` creates and initializes a CUDA mEnclave (`mEnclave C`) in partition 2, using the mEnclave interface. `mEnclave A` then communicates with `mEnclave B` via RPC to accelerate data processing.

IV. Cronus

A. MicroEnclave

The MicroEnclave Model. Cronus partitions a monolithic enclave computed on heterogeneous hardware into multiple MicroEnclaves (mEnclaves), each running a type of computation (e.g., CUDA computing). This partition design is feasible because enclave execution shares the same execution model (i.e., remote execution [103]) with accelerators' execution. For instance, both launching a CUDA kernel (in GPU) and doing ECalls in a CPU enclave offload the computation of a function (`CUDAMatAdd` or `ECallMatADD`) to a device (e.g., GPU) or the TEE.

At a high level, we model mEnclave as a black box executor $\langle mECall, state \rangle$ that executes a fixed set of functions $mECall$s on the internal $state$ (e.g., memory and GPU state), without revealing these states to untrusted parts outside an mEnclave. We define only the specification of the executor, yet the implementation can vary. For example, an executor can execute a dynamic library (e.g., OpenEnclave [86], Intel SGX SDK [52]) and a CUDA executable file (i.e., CUDA ELF). We call an implementation of a black box executor as an execution model. Specifically, the execution model defines several functions in the lifecycle of an mEnclave (e.g., `me_create`). For example, a CUDA mEnclave's `me_create` parses a CUDA ELF file and loads it into the secure memory.

API of an mEnclave. Similar to the standard enclave API [50], mEnclave encapsulates the same set of API (e.g., `create` and `ecall`) for managing and executing code within TEE. The main difference is that standard enclave

```
{   ...
    "device_type": "gpu",
    "images": {
        "mat.cubin" : "654c28186755aa92",
        "cudart.so" : "2814c867aa955265",
        "cudav3.mos": "de92d2fc587d10a6"
    },
    "mEcalls": "mat.edl",
    "resources": {"memory": "1G"},
}
```

Figure 3: The CUDA mEnclave part of a manifest file.

API adopts pre-defined runtime libraries (e.g., Intel SGX SDK and OpenEnclave) to support CPU TEE computing. Instead, mEnclave allows the execution of diverse code (e.g., CUDA code) using provided execution model (e.g., using the CUDA runtime) to enable computing using accelerators in TEE. Upon a mEnclave creation, a `manifest` (Figure 3) is required to specify the device type, the resource capacity (e.g., memory size), a list of `mECalls`, and the hash of the mEnclave runtime and image. We reused SGX's `edl` format for specifying `mECall` and instrumented the format with the synchronization/asynchronization flag for sRPC (§IV-C). The mEnclave image is a file that stores execution code. For example, the image is a dynamic library (`.so`) for CPU mEnclave and CUDA ELF (`.cubin`) for CUDA mEnclave. It can also be `null`, if a device executes only pre-defined functions (e.g., an FPGA with fixed program functions).

EnclaveManager manages mEnclaves in an mOS. Enclave Manager implements several functionalities such as attestation and bookkeeping the resources utilization, independent of the execution model. When an untrusted app or an mEnclave invokes `create`, Enclave Manager (EM) reads the manifest and mEnclave image, allocates resources and loads the execution model (i.e., runtime) into the memory according to the manifest. The caller of `create` is the *owner* of the mEnclave, and only the owner can invoke `mECall` of the created mEnclave. Then, the Enclave Manager calls `me_create` at the loaded runtime for parsing the image and initializing. Note that the created enclave is identified by a 32-bit identifier (`eid`) where the first 8 bits are the mOS id, and the last 24 bits are for the enclave id within the mOS. SPM uses the mOS part for validating cross-mOS messages (e.g., sRPC initialization).

It is subtle to guarantee the ownership of mEnclave, as mOSes are mutually untrusted before the attestation and can fail arbitrarily (§III-B). We tackle this problem by integrating Diffie-Hellman [35] into mEnclave creation for sharing a secret $secret_{dhke}$. Before the caller and the created mEnclaves setup a trusted communication channel (see §IV-C), each message is signed using $secret_{dhke}$. This is because both the caller and the created mEnclaves' mOSes can fail and be substituted with a new mEnclaves with the same `eid` (§III-B).

Remote Attestation. To accommodate the attestation of a dynamic TEE platform with diverse accelerators, we propose

a dynamic attestation protocol for mEnclave, an extension of the previous two-phase attestation protocol [106], [6]. Specifically, CRONUS's attestation protocol makes a client firstly verify a closure of hardware and software state (e.g., hashes of mOSes and mEnclaves) according to the declared hardware resource in the manifest (§IV-A) and then makes sure that clients' data are processed by only the verified software (i.e., mEnclaves and mOSes) and hardware (e.g., GPU). Note that the accelerator mEnclaves may not be created during the attestation, and CRONUS makes sure that the future accelerator mEnclaves comply with the attestation report by automatically conducting local attestation using the attestation report (§IV-C). Hence, a client does not need to attest an mEnclave each time it is created.

To prevent fabricated accelerators and misconfiguration of accelerators, CRONUS's attestation protocol verifies both accelerators' configurations and authenticities. First, CRONUS makes use of the device tree [33] in Linux's infrastructure for verifying the configuration of accelerators (e.g., GPU execution mode). A device tree (DT) provided by the untrusted OS specifies the hardware configuration and interconnects of accelerators and shared resources (e.g., PCIe bus). To avoid buggy (malicious) DT triggering various attacks (e.g., MMIO remapping and interrupt spoofing attacks [126]), CRONUS accepts only valid DT (e.g., no overlapping IRQ and MMIO, the same properties enforced in TrustPath [126]) and includes it in the attestation report. This DT is retrieved once during SPM initialization (during boot up, see V-A) and cannot be modified to resist malicious manipulations. For adding/removing accelerators, the untrusted OS updates the DT and restarts the machine to activate the new DT.

Second, for authenticity verification (i.e., real hardware instead of fabricated ones), each accelerator stores a secret key ($\langle PubK_{acc}$, $PvK_{acc}\rangle$) and signs configurations using the key, a common practice for hardware authenticity [119], [19], [39], [124], [109]. The accelerator's mOS verifies that the accelerator owns the key and includes $PubK_{acc}$ in the attestation report. The client verifies that $PubK_{acc}$ is endorsed by the accelerator vendors for authenticity.

CRONUS adopts the same root of trust (a secret key $\langle PubK$, $PvK\ \rangle$) for the platform, same as previous TEE work [124], [50]. CRONUS's secure monitor proves the ownership of the root key for generating an attestation key (AtK). CRONUS's secure monitor measures hashes of mOSes, while mOSes measure the hashes of mEnclaves. CRONUS's secure monitor signs the complete attestation report (i.e., $\langle$`hash`(mEnclave), `hash`(mOS), DT, $PubK_{acc}\rangle$) using AtK and sends the report, signature and AtK to clients. Clients verify that AtK is endorsed by the attestation service and make sure that the report is signed by AtK. CRONUS's attestation differs from previous work in including the hardware configuration and privileged component (e.g., OS) in the attestation report to support a dynamic configured TEE platform. Overall, CRONUS ensures that an mEnclave trusts only the code and hardware in the same partition (**R3.2**).

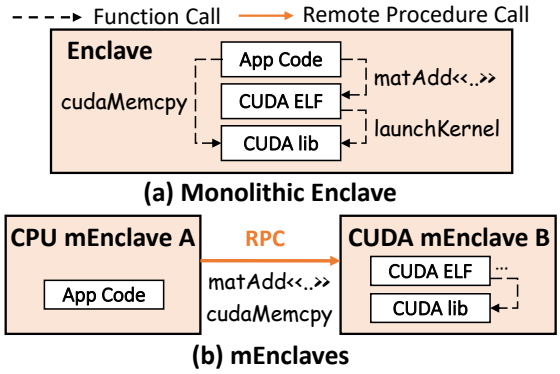

Figure 4: Partitioning a monolithic enclave into mEnclaves.

Local Attestation. CRONUS's local attestation is crucial for RPC's security (§IV-C) and takes three steps. First, a challenge mEnclave sends a local attestation request (signed by $secret_{dhke}$) to the attested mEnclave (via untrusted memory). Second, the attested mEnclave requests a measurement report signed by SM (using a local seal key LSK in SM) and signs the report using $secret_{dhke}$. Third, the challenge mEnclave verifies that the signed report is from the attested mEnclave and is co-located with the same machine with the correct identity by checking if the report is endorsed by SPM.

B. Hardware Adaptation Layer (HAL)

HAL in an mOS provides a unified interface for Enclave Manager configuring, attesting and accessing hardware resources (e.g., GPU memory and cache). Specifically, CRONUS includes several APIs (e.g., `map`) for managing hardware contexts and resources (e.g., resource mapping). HAL also handles page faults and interruptions from the device. Overall, HAL works as a "driver" and virtualization layer for a device (**R2**).

Implementing HAL for accelerators (e.g., GPU) from scratch is error-prone and tedious, yet we observe that most open-sourced device drivers in a monolithic OS are mature and modular (`.ko` format in Linux). Hence, CRONUS includes a shim runtime for running off-the-shelf device drivers in mOSes, bootstrapping the integration of general accelerators (**R1**). The shim runtime works as if a LibOS [106] for the driver by providing standard kernel functions (e.g., `ioremap`). CRONUS also includes the PCIe device driver as many heterogeneous devices are connected through PCIe. Therefore, CRONUS can support PCIe devices with almost no software modification (§V). CRONUS enables spatial sharing (**R2**) of resources by leveraging accelerators' intra-accelerator isolation techniques (e.g., by NVIDIA MIG [84]). Note that the driver may be malicious [126] as it may be developed by a third party (§III-B), so each driver needs to be isolated (**R3**).

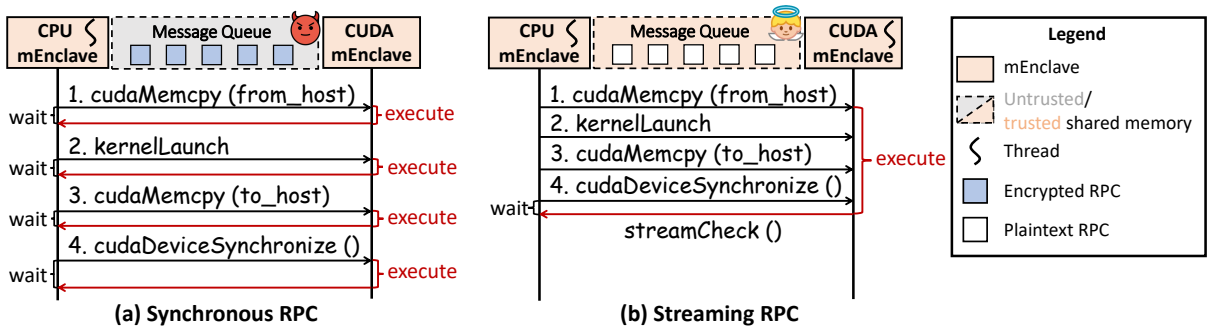

Figure 5: Synchronous RPC and Streaming RPC.

C. Constructing Heterogeneous TEE Computation with Streaming RPC

With mEnclave, CRONUS constructs heterogeneous computation within TEE using **remote procedure call (RPC)**. As shown in Figure 4, a monolithic enclave $Enclave$ is partitioned into two mEnclave mE_A and mE_B, running CPU computation and CUDA computation, respectively. With CRONUS's mEnclave abstraction, it is possible to do the partition automatically (§V).

Challenges. A straightforward implementation of RPC mechanism for mEnclaves is synchronous RPC (Figure 5a): the RPC writes the RPC metadata (e.g., function names and parameters) into the public (untrusted) memory buffer, saves the current execution context (e.g., memory stack), conducts a context switch to the normal world and do the `mECall` as in §IV-A. After the execution, the returned results are stored in the global memory buffer, and it asks the calling enclave to resume execution from the saved context.

Two subtle problems arise when adopting this naive approach. First (security), the RPC interleaving between mEnclaves may be seen or forged by the untrusted OS. Specifically, a malicious attacker can compromise the computing results by 1) modifying the metadata (e.g., parameter or returned results) of an RPC, 2) dropping or replaying RPC between mEnclave, 3) the timing of RPC (e.g., memory copy and kernel launching) potentially leaks sensitive user information [49], [90]. Hence, CRONUS needs to guarantee that executing a sequence of `mECall` in mE_A and mE_B results in the same results as the original `ECall` in $Enclave$, and eliminates the timing information of RPCs.

Second (performance), synchronous RPCs waste excessive computing resources for context switches and can significantly downgrade performance. This is because synchronous RPCs in ARM S-EL2 require at least four context switches to switch from one mEnclave to another (same number for resuming executions, see [72]).

To tackle the two problems, existing work is inefficient (i.e., lock-step RPC execution) or requires extensive application modification (see §II-C), unsuitable for CRONUS.

Streaming RPC (sRPC). Our approach (Figure 5b) is a novel streaming RPC model that resolves the above security problem and preserves high performance. The observation of sRPC is that accelerators' execution is usually asynchronous and sequential, so it does not need costly synchronization (context switches) among mEnclaves. Hence, sRPC models the RPC requests as input in a stream-processing system [1], [40]. For a mE_A conducting RPC to mE_B, a thread is running for mE_B to execute RPCs, and the thread is created on demand. mE_A continuously sends RPC requests to the thread, and checks the progress of mE_B, only when mE_A needs data from mE_B (e.g., returned results are not `null`) or needs synchronization (e.g., `cudaDeviceSynchronize`). Moreover, sRPCs are sent via secure TEE memory shared by mEnclave (§IV-C), so attackers can no longer manipulate RPCs' metadata.

When mE_A connects to mE_B, mE_A conducts local attestation (automatically by CRONUS) to verify mE_B's identity to prevent malicious mE_B and creates a trusted shared memory region $smem$ (see §IV-C). Then, mE_B proves to mE_A the ownership of $secret_{dhke}$ through $smem$ to make sure that $smem$ is indeed shared between mE_A and mE_B (i.e., `dCheck`).

When CRONUS is executing a $mECall_A$ in mE_A, calling the first $mECall_B$ in mE_B creates a stream S with a request index Rid, a progress index Sid and a ring buffer in $smem$ for storing RPC requests. CRONUS asks the normal world to create a thread T which enters the execution loop in mE_B. To support multi-threading, CRONUS makes each thread create its own stream for RPCs. If the $smem$ is out of memory, $smem$ is expanded with `dCheck`. Note that attackers cannot fabricate the RPCs from other mEnclaves, as $mECall_B$ in mE_B can only be called by its owner (i.e., mE_A, see §IV-A).

The thread T continually retrieves `mECall` from the ring buffer and executes it. After executing a `mECall`, Sid is added by one. mE_A continually writes (serialized) RPC requests to the ring buffer (and adds Rid by one), but does not wait for the return results from mE_B, unless it requires a return result or needs synchronization (e.g., `cudaDeviceSynchronize`, red arrows in Figure 5b). Otherwise, it waits for the results returned from mE_B and checks if $Sid = Rid$ (i.e., `streamCheck`). After the execution of mE_A finishes, S is closed and T is stopped. To reduce the stream creating

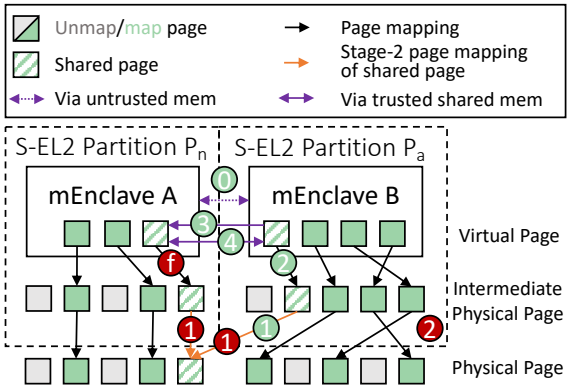

Figure 6: Secure TEE memory sharing of two mEnclaves. ⓧ is a step for sRPC's normal case, while ⓚ is a step for sRPC's failure case.

cost, CRONUS adopts an optimization to create a default thread $T_{default}$ that can be reused by multiple streams.

Implementing trusted shared memory. CRONUS enables sharing of trusted TEE memory (or accelerator memory). There are two types of memory sharing in CRONUS: intra-mOS memory sharing and inter-mOS memory sharing. CRONUS implements intra-mOS sharing by mapping two mEnclaves' addresses to the physical address of the same resource (e.g., physical TEE memory and MMIO).

For inter-mOS sharing, Figure 6 shows the workflow for mE_A sharing a page to mE_B. ⓪ Before sharing memory, mE_A conducts local attestation to mE_B (if necessary) to prevent a malicious mE'_B. This step is executed automatically for constructing sRPC through trusted shared memory (§IV-C). ① mE_A asks SPM to map its resources to mE_B's mOS, and ② mE_B's mOS map the resource to mE_B's address space. The SMMU page table is also modified if required. CRONUS's SPM also records the shared pages among mOSes in SPM for fast recovery (§IV-D). ③ `dCheck` is conducted through the shared memory to verify that the communicating mEnclave is indeed mE_B. This is essential as failures may happened during ① ∼ ②: an authenticated mE_B can be replaced as a malicious one. ④ mE_A and mE_B can communicate (e.g., doing sRPC) securely on the shared memory. To reduce the time for modifying the stage-2 page table, CRONUS asks each mOS to proactively reserve a shared memory region between partitions.

Besides RPC, trusted shared memory can also be used for implementing other inter-enclave communication approaches (e.g., pipe and peer-to-peer accelerator communication) beyond RPC. Also, inter-enclave synchronization primitives (e.g., locks and conditional variables) are implemented through atomic operations (e.g., compare-and-swap instructions) on the shared memory. CRONUS replaces mutex with spinlock, which avoids involvements of the untrusted OS.

Security analysis. sRPC prevents malicious attackers from dropping, reordering and replaying RPC requests. First, the use of trusted shared memory makes it impossible to manipulate RPC contents. Second, the executions loop in mE_B fetches RPC requests only when there are no executing RPC (i.e., Sid equals to the next element of the ring buffer), so all RPC calls are executed sequentially. sRPC also prevents information leakages through inter-enclave RPCs. This is because all RPCs go through a secure memory region that cannot be seen by an attacker, so RPC timing [49] is hidden. Overall, the normal case of sRPC is safe against attacks, and we discuss and resolve safety issues during failures at §IV-D.

D. Securely Recover from Failure/Update

sRPC's normal case protocol (§IV-C) ensures secure and efficient RPC. In this section, we will discuss sRPC's failover protocol for mOS and mEnclave failures. From the previous analysis, communication via untrusted memory (e.g., local attestation) is protected by $secret_{dhke}$ and CRONUS detects failures by `dCheck`. Therefore, this section focus on handling communications via trusted shared memory during failures.

Notations. We assume an mEnclave mE_A in P_n (e.g., managing CPU computation) is communicating with an mEnclave mE_B in P_a (e.g., managing a GPU), as shown in Figure 6. A partition P_i runs an mOS O_i on a device D_i^c with configuration c (including device content). We use $smem(mE_I)$ for the shared memory in mE_I and $pt(mE_I)$ for the corresponding page table entries. We use $smem(P_i, P_j)$ for the shared memory that P_i shares with P_j, and $pt^2(P_i, P_j)$ for the corresponding stage-2 page table entries in S-EL2. The SMMU page table entries that P_i shared with P_j are denoted as $spt^2(P_i, P_j)$.

Security attacks. To motivate the design of our system, we begin by introducing three security attacks during failures.

$\mathcal{A}_1$. *Time-of-check-time-of-use (TOCTOU) attacks.* After the failure of P_a, mE_A does not know that P_a has failed, and still sends sensitive information through shared memory to a new mE_B in P_a, which may run with a malicious P_a or device D_f^c. Therefore, mE_A is potentially sending sensitive information to a malicious party. This is because mE_A does not know if mE_B has failed, so it cannot check the authenticity.

$\mathcal{A}_2$. *Deadlock.* mE_A and mE_B may communicate with each other through a shared memory region. To avoid data races, a lock is implemented through a lock implemented by atomic memory access (e.g., compare-and-swap). Suppose that mE_B acquires the lock, and then P_a fails. At this time, the lock is acquired by a dead mE_B thread, causing deadlocks.

$\mathcal{A}_3$. *Crashed information leaks.* After a P_a recovers to P'_a, a malicious P'_a or a malicious mEnclave in P'_a can still read D_f^c and $smem$, because the content of the physical devices and shared memory $smem$ is not cleared if the application or OS does not proactively do so [89]. Therefore, P'_a can retrieve the sensitive data of the crashed mEnclaves in P'_a, causing leakages of sensitive data.

A straightforward approach for failover is to conduct a cold-reboot of the whole server (all partitions), killing all mEnclaves and mOSes (including live ones). Since all partitions are rebooted at the same time, and cold-reboot can safely clear accelerator states [127], avoiding all attacks. However, cold-reboot results in a long recovery time and is not fault-isolated: all mEnclaves in the server are shutdown (violating **R3.1**).

Our approach. sRPC's failover (Red steps in Figure 6) is a secure, low-downtime and fault-isolated *proceed-trap* failure recovery procedure. CRONUS realizes all the three properties by two key designs: (1) CRONUS proactively prevents all access to the failing partitions to avoid $\mathcal{A}_1$ and $\mathcal{A}_2$ (i.e., *proceed*), (2) later shared memory access will generate *traps* for further fault handling, and (3) CRONUS decouples the failing clearing logic and startup logic to tackle $\mathcal{A}_3$.

CRONUS identifies a failed P_a in three circumstances. First, a P_a or the untrusted OS proactively requests a restart of the P_a's mOS to the SPM. This is often caused by a update or configuration of mOS. Second, a P_a panics and transfers control to the SPM due to a hardware or software failure, which cannot be handled by the P_a's mOS. Third, the SPM proactively detects if a P_a hangs (in a spinning way) by checking the status of P_a's mOS.

After identifying a failed P_a, CRONUS recovers it in three steps: ❶ CRONUS first finds all partitions that contain shared memory with P_a. For a P_i of them, CRONUS invalidates stage-2 page table entries (orange arrows in Figure 6) in $pt^2(P_i, P_a)$ and SMMU entries in $spt^2(P_i, P_a)$, so that access to the shared memory is invalid. After that, all subsequent shared memory access will generate faults, removing the TOCTOU window ($\mathcal{A}_1$). After that, CRONUS's secure monitor marks the partition P_a as failed by setting $r_f = 1$, so that all consecutive new memory sharing requests are blocked. Afterward (❷), CRONUS loads P_a's failure handling function to clear D_i^c and $smem$. After that, CRONUS can load an mOS into P_a and initialize. After initialization finishes, CRONUS sets P_a ready by setting $r_f = 0$.

Traps handling (❢). Each shared memory access from mE_A in P_n to P_a would generate a fault that will be handled by CRONUS. Such access may be a memory read/write (atomic or not) to $smem(P_i, P_a)$. When such a fault is generated, CRONUS asks P_n to invalidate the mE_A's page table entries that map memory to P_a's. For pages that are owned by P_i, CRONUS recovers P_i's accesses to the page by changing $pt^2(P_i, P_a)$. Then, CRONUS asks P_n to generate a fault signal to mE_A to notify that the communicating mEnclave is faulty. By doing so, the mEnclave would not leak sensitive information or get stuck by a deadlock. CRONUS's sRPC automatically clears state when getting the signal. For applications that explicitly use shared memory (e.g., distributed applications), developers write an exception handler for handling failures. Note that if the shared memory is not accessed after failures, the (invalidated) shared memory is reclaimed (both in its mOS and SPM's stage-2 page table) after the mEnclave terminates.

Handling concurrent failures. If there are concurrent failures of several partitions $\langle P_{a1}, \cdots, P_{an} \rangle$, CRONUS serializes the executions of step ❶ for all failed partitions. ❷ and ❢ can be executed in concurrently (with locks for changing page tables), reducing downtime of failover handling.

Handling mEnclave failures. When an mEnclave fails, its mOS removes all its page mapping in its mOS and invalidates the physical pages' mapping of both mOSes in the stage-2 page table, so that the communicating mOS can notify the communicating mEnclave (and its mOS) for failure handling.

Security analysis. The *proceed-trap* protocol eliminates all three attacks and guarantees crash safety. First, TOCTOU attacks are prevented, because step ❶ invalidates stage-2 page tables of the shared memory, and ❢ traps consecutive memory access. Also, no crashed information in the device is leaked, as step ❷ clears device data before initialization. Moreover, deadlocks are avoided as an mEnclave received a signal when trying to memory shared with a fail partition. We note that a memory page may be shared by another mEnclave in P_i, which shares the same memory with P_a, step ❢ may result in deadlock as this mEnclave's access is not trapped. CRONUS tackles this problem by restricting that a memory page can be shared only once.

Overall, sRPC is crash-safe: the *proceed-trap* procedure ensures crash safety of sRPC via trusted shared memory (④), while communication via untrusted memory (e.g., attestation, ⓪) is protected by $secret_{dhke}$ (§IV-A); The crash safety during trusted shared memory establishment (① ∼ ②) is guarded by `dCheck` (③). Moreover, CRONUS's failover is fault-isolated: CRONUS restarts only fault-inducing mOSes and mEnclaves upon failures (§VI-D). Note that CRONUS currently uses trusted shared memory for only sRPC; mEnclaves using trusted shared memory for other purposes (e.g., sharing application data) is crash-safe with the above protocol, but requires the mEnclave developers to write trap handlers for failures (e.g., to recover data). In the future, checkpoints [24], [58] of OSes and applications can be integrated to recover application data (§III-B).

V. IMPLEMENTATION

A. Prototyping CRONUS on ARM

We prototype CRONUS on TrustZone as it supports the four required hardware primitives (§III-C), and ARM (TrustZone) is a commodity product and supports diverse hardware devices (e.g., GPU) for running sophisticated applications (e.g., DNN training [66], [63] and inference [12]) with high performance. There are other TEEs (e.g., RISC-V PMP [68]) that provide similar hardware primitives, and CRONUS's design can be directly applied to them (§VII-A). At the time of writing, there are no commercial products that support ARM S-EL2. Same as previous work using S-EL2 [72], we

prototyped CRONUS on ARM Fixed Virtual Platform (FVP) with full-featured S-EL2 for functional evaluation and on QEMU for performance evaluation. CRONUS can directly run on real ARM S-EL2 hardware once it becomes commercially available.

Functional evaluation. We implement all CRONUS's functions on FVP, including sRPC and failover protocol. Since FVP does not support passthrough devices, we emulate device execution using a spinning loop. We evaluated CRONUS's RPC performance and failover performance. We used this implementation for evaluating failover of CRONUS. CRONUS's FVP implementation adds memory sharing (§IV-C) to Hafnium [46] for sRPC and its failover.

Performance evaluation. We evaluated CRONUS's performance mainly using QEMU. We do not choose to evaluate CRONUS on real hardware platform as the current TrustZone platform does not support secure PCIe peripheral connections. To isolate normal world memory and secure world memory in CRONUS, CRONUS allocates two separate `MemRegion` for the normal and secure world, and creates an emulated ARM TZC-400 device for configuring the size of the two `MemRegion`. To isolate IO devices, we create a "secure" PCIe bus and bind its resources (e.g., BAR) to different memory addresses from the original PCIe bus. All secure devices would be connected to this bus. Overall, we realize isolated devices and memory in CRONUS's simulated execution environment. Moreover, to support large secure memory, we create a new `MemRegion` that is out of the range of the normal world (e.g., $\geq 16GB$) and is used by user enclaves. For attestation, we implement a device storing a read-only secret for PvK (§IV). Since QEMU has no support of S-EL2, we implemented the SPM in `SEL1` and isolated each mOS running at different address spaces (i.e., running at `EL0` and using the shadow page table [65]).

Table II shows the configuration of the prototyped system on QEMU. The host machine is equipped with Intel(R) Xeon(R) Silver 4116 with 24 cores, 128 GB RAM, 2TB SSD and four NVIDIA RTX 2080 GPUs. We use QEMU to emulate an AArch64 A53 machine with four cores, 8GB normal memory and 4GB secure memory. CRONUS uses `vfio` [108] to passthrough NVIDIA GPUs to CRONUS. We modify QEMU's simulated PCIe bus to allow devices in the secure PCIe bus to conduct DMA access only to the secure memory region.

Bootup of CRONUS. CRONUS adopts the same bootup procedure as TrustZone's. CRONUS first boots up Secure Monitor and SPM and initializes using the DT (§IV-A). CRONUS locks down all devices configured to the secure world to resist malicious reconfiguration [54], [13], [71]. Note that a device can be configured by DT (after reboot) to be used either by the secure world or normal world. Then, CRONUS loads each mOS using the mOS image from the normal OS.

	Host	**Guest**
OS	Ubuntu 16.04	Ubuntu 20.04
Kernel	4.15.0	5.9-rc1
QEMU	5.0.0	-
CPU	Silver 4116	A53
GPU	NVIDIA GTX 2080	Passthrough GPU

Table II: The prototyped system's configurations.

B. Case Study

We prototype CRONUS to support CPU, NVIDIA GPU, and VTA-compatible NPU [107] computation. CRONUS provides three kinds of mEnclaves: CPU mEnclave, CUDA mEnclave and NPU mEnclave. Table III shows the LoC for implementing the mEnclaves's mOS. Compared with a monolithic OS that trusts all mOS code (each with different versions, see §III-B) and hardware into the TCB, a PaaS service trusts only its own mOS's code (used by the service) and hardware (§III-B). Moreover, CRONUS enables TCB reduction as each mOS needs only include code for running a specific hardware model (e.g., NVIDIA Turing) instead of all hardware models (e.g., from Kepler to Turing), greatly reducing attack surface [69]. Overall, the porting efforts are moderate for diverse supporting diverse accelerators and frameworks, and we did not need to modify application code (e.g., DNN models).

CPU mEnclave. CRONUS reuses code in OPTEE for implementing HAL for CPU mEnclaves. We built the CPU mEnclave runtime using `musl`c and a library OS (similar to CubicleOS [97]) to run applications within mEnclave with few modifications. The library has a small TCB as each mEnclave includes only necessary syscalls (see [97], [64]).

CUDA mEnclave. CRONUS uses the open-source NVDIA driver `nouveau` to build GPU's HAL. CRONUS uses `gdev` [59] and `ocelot` [37] for building the CUDA mEnclave's runtime. We modified `valgrind-mmt` and `envytools` to reverse engineer the latest NVIDIA CUDA library to extend `gdev`'s and `nouveau`'s support to to GTX 2080, a state-of-the-art GPU. We leverage GPU virtual address isolation for isolating different mEnclaves' code, and other isolation techniques (e.g., MIG) can be directly integrated when it is available in the hardware and enabled by the driver (`nouveau` does not support MIG and GTX 2080 does not have MIG). Besides memory sharing on CPU memory, CRONUS supports shared GPU memory to enable direct GPU communication over PCIe.

NPU mEnclave. CRONUS uses TVM VTA's `fsim` simulator and runtime code for implementing the NPU accelerator and the NPU mEnclave, respectively. Specifically, we built an NPU accelerator by implementing a simulated QEMU PCIe device that runs VTA's `fsim` simulator code. The NPU accelerator enforces isolated concurrent NPU code execution within the device using virtual memory. CRONUS uses the `fsim` runtime code for the NPU mEnclave and the `fsim` driver code for its mOS's HAL, respectively. To make the

		Line of Code
CUDA mOS:	`nouveau` [2]	194,927 → **52,912**
CPU mOS:	`optee` [4]	222,700 → **12,369**
NPU mOS:	`vta-driver` [107]	**901**
CRONUS's components:		
mEnclave Manager		**4,302**
HAL core		**2,132**
Monolithic OS		**418, 528**

Table III: LoC of different mEnclaves and their mOSes. The original LoC is also shown for modules with significant code reductions.

code compatible with the NPU device, we changed the original function call access interfaces (in the original `fsim` runtime and driver code) to PCIe memory BAR access (for commands) and MMIO access (for device memory) for the NPU mEnclave.

Automatic partitioning of a monolithic enclave. To mitigate the requirements of manually partitioning a monolithic enclave into mEnclaves, CRONUS includes a tool to automatically partition the monolithic enclave into mEnclaves using the annotation of `mECall` in the manifest file (§IV-A), add mEnclave management code, and convert all CUDA/VTA calls (e.g., `cudaLaunchKernel`) within a monolithic enclave to `mECall` (i.e., mEnclave RPC). Note that automatic partition is enabled by the mEnclave model for TEE computation (§IV-A) and requires no shared application state between mEnclaves.

VI. EVALUATION

A. Evaluation Setup

Baselines. We compared CRONUS with three baseline systems: a monolithic TrustZone implementation, native Linux, and HIX-TrustZone. The monolithic TrustZone implementation includes all device drivers of GPU and NPU in the secure world. For brevity, we denote the monolithic TrustZone implementation as "TrustZone". We also evaluated native Linux execution without TEE, denoted "Linux". The TrustZone and Linux baselines are evaluated with CPU (§VI-B), GPU (§VI-B and §VI-C) and NPU (§VI-C) workloads.

Since existing TEE (e.g., Graviton [109], [54]) that supports GPU is not open-source, to further evaluate the benefits of CRONUS on GPU computations, we also compared CRONUS with HIX [54], a notable GPU TEE system. We emulated HIX in TrustZone by running the GPU driver within enclave (i.e., GPU enclave in HIX), and grants the enclave dedicated access (in the TrustZone OS) to the GPU. Then, an application enclave uses encrypted RPC over untrusted memory to communicate with the GPU enclave. We denote this HIX implementation as "HIX-TrustZone". This HIX-TrustZone baseline is evaluated with GPU workloads (§VI-B and §VI-C) as HIX supports only GPU. Unless otherwise specified, we used the QEMU implementation for all baselines in our evaluation.

Workloads. We evaluated CRONUS on microbenchmarks (§VI-B) and real-world DNN training and inference workloads (§VI-C). Specifically, we evaluated CRONUS using the standard well-recognized GPU benchmark `Rodinia` [96] and NPU benchmark `vta-bench` [110], used in previous TEE systems [5], [54], [119]. Overall, CRONUS's evaluation includes a **full** coverage of microbenchmarks and real-world software evaluated in related systems [54], [109], [127], [119].

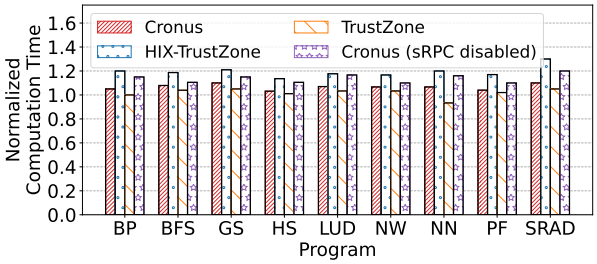

Figure 7: Normalized computation time of `Rodinia`.

We further evaluated CRONUS's performance on DNN training workloads using GPU for accelerations. We evaluated four DNN networks on three datasets, including LeNet-2 on MNIST [66], ResNet50 [47] and VGG16 [88] on CIFAR-10 [63], and DenseNet [48] on ImageNet [31], respectively. We used `PyTorch` [92] as the training framework, as it is the state-of-the-art training framework used by academia and industry. For TrustZone, HIX-TrustZone and CRONUS, we ran the whole PyTorch training program in TEE, and we protect both CPU and GPU computations. We denote PyTorch running on CRONUS as CRONUS-PyTorch.

We also evaluated DNN inference workloads on CRONUS using TVM [12], a widely used tool for compiling DNN models into high-performance executable code on various hardware, including VTA-compatible NPU (§V). We denote the TVM running on CRONUS as CRONUS-TVM. We evaluated the inference latency on three popular pre-train DNN networks, including ResNet18, ResNet50 [47], and YoloV3 [95].

Our evaluation answers the following questions:

- §VI-B What is the performance of CRONUS in the `Rodinia` and `vta-bench` benchmark?
- §VI-C What is the performance of CRONUS-PyTorch and CRONUS-TVM?
- §VI-C What is the performance gain of spatial sharing?
- §VI-D How fast can CRONUS recover from faults?

B. MicroBenchmark

Figure 7 shows the computation time of CUDA mEnclaves running different CUDA programs in CRONUS and native `gdev`, normalized to the native `gdev`. The results suggest that CRONUS incurs less than 7.1% performance overhead compared with gdev (without TEE). CRONUS is also faster than HIX-TrustZone, this is mainly because HIX-TrustZone's expensive RPC protocol and more frequent RPCs. HIX conducts an RPC for each hardware control mes-

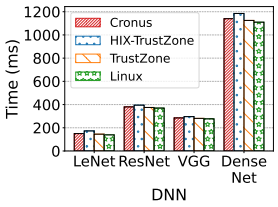

Figure 8: Training time of different DNNs on CRONUS and baselines.

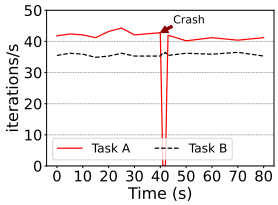

Figure 9: CRONUS's failover evaluation of running two tasks on separated partitions.

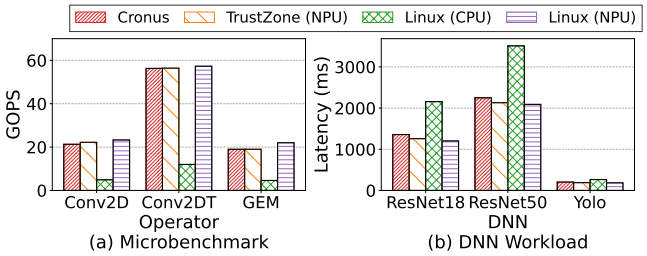

Figure 10: The throughput of `vta-bench` and the latency of DNN inference.

sage (`ioctl`), while CRONUS's mEnclave conducts sRPC for a CUDA function or a kernel launching function (§V). To investigate CRONUS's performance overhead, we disabled CRONUS's sRPC and replaced it with synchronous RPC. Figure 7 shows the results. With CRONUS's sRPC disabled, CRONUS incurs a much larger overhead. This is because of the costly context switches in TrustZone. Yet, CRONUS is still faster than HIX-TrustZone due to the elimination of encryption/decryption.

Figure 10a shows the throughput of NPU mEnclaves running different DNN operators in `vta-bench`. Overall, the performance overhead of CRONUS is negligible (<2%). The performance of CPU was worse than NPU, even we used a simulated NPU, because the CPU runs `aarch64` code executed by QEMU, while the simulated NPU device is executed natively in `x86` (§V).

C. DNN Workloads

Figure 8 shows the time for training different DNNs on one batch of data records. Similar to the performance overhead in the microbenchmark, CRONUS-PyTorch incurs less than 0.8% overhead in computation time. On the other hand, HIX-TrustZone incurs around 5.3% computation-time overhead because of the encryption/decryption costs.

To investigate the gain of CRONUS's spatial sharing, we used CRONUS-PyTorch to train multiple LeNet-2 training instances concurrently and serially (denoted as CRONUS-Temporal). Specifically, we ran $1 \sim 8$ CRONUS-PyTorch instances and measured the total computation time of all training instances and the throughput of the GPU (i.e., the number of batches executed for a duration of time). Figure 11a shows that with spatial resource sharing enabled, the training time increases slowly when the number of concurrent mEnclaves is small. This is because the latest

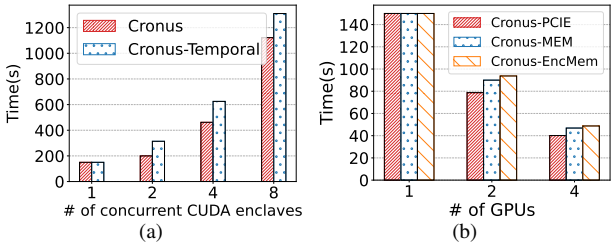

Figure 11: Training time of LeNet-2 using CRONUS-PyTorch with different numbers of mEnclaves running at the same GPU (a) or multiple GPU (b).

GPU model enables concurrent GPU kernel execution on a GPU [85]. When there are more concurrent mEnclaves (4 mEnclaves), the performance downgrades because of resource contentions. We observe up to 63.4% throughput growth with spatial sharing of GPU.

Figure 11b shows the training time of LeNet-2 with various numbers of GPU running the data-parallel distributed computations. The results suggest GPU sharing using the PCIe bus results in the best performance, compared to using secure memory or encrypted memory [54], [109] for sharing. CRONUS enables P2P communication between accelerators by using the trusted shared memory of mEnclave.

Figure 10b shows CRONUS-TVM's inference latency of various real-world DNN models running on an NPU simulator and on the CPU. Running computation on an NPU simulator is slightly slower than native execution (unprotected), and is almost the same as using the monolithic TrustZone. We observe that the execution time of ResNets is long due to its complexities and the use of simulated NPU. We expect fast inference time using real hardware implementation of NPU.

D. Failure recovery

To investigate the gain of CRONUS's fault isolation and fast and secure failure recovery protocol, we ran two matrix computing tasks with different sizes of input on GPU using CRONUS's FVP implementation. Since FVP does not support passthrough devices, we recorded the task execution time on CRONUS's QEMU execution, and CRONUS's FVP GPU enclave uses the recorded execution time to emulate GPU execution. The two tasks are on different S-EL2 partitions. During execution, we measure the throughput of each task and we create a crash during execution. We then used CRONUS's recovery approaches for recovery and measured the throughput. For the task that failed due to the crash, we resubmitted the task when the system recovered. Figure 9 shows that CRONUS's proceed-trap recovery protocol can swiftly recover. CRONUS's throughput cannot quickly recover to the peak throughput because in CRONUS the failed task needs to be resubmitted and requires initializations. We also measured the time for rebooting the system, which requires around 2 minutes.

VII. Discussion

A. Adapting Cronus to other TEEs

Cronus can be directly applied to a TEE as long as it supports the four hardware primitives (§III-C). TEEs [68], [81] using RISC-V PMP (its extension [38]) and CCA [15] support all four hardware primitives, so Cronus can be directly applied to them with some engineering efforts. For RISC-V, SecureIO is supported by configuring PMP to ensure an enclave's dedicated access to a device's MMIO addresses; shared TEE memory is enabled using overlapped PMP configuration [38], [68].

VM-based TEEs (e.g., AMD SEV [10] and Intel TDX [53]) typically lack SecureIO, and Cronus can be applied to them once SecureIO is supported (e.g., using HIX [54]). Note that Intel TDX [53] supports shared TEE memory (as TDX leverages MKTME that supports shared VM memory); AMD SEV [10] does not support shared TEE memory, and it may require hardware modification (if firmware updates are not possible) for shared TEE memory. Similar to AMD SEV, Intel SGX [50], [78] supports only Isolation and Hardware RoT, so applying Cronus to it requires hardware modifications.

B. Hardware Advice for Future TEE

Cronus currently runs on existing TEE hardware (§III-C), and new hardware primitives can be created to improve the performance of Cronus and enable new applications.

Trusted TEE shared memory. Cronus's sRPC protocol (§IV-C) currently requires co-designs of both mOSes and SPM, which can be optimized by implementing Cronus's trusted TEE shared memory mechanism (§IV-C) in hardware, which ensures the identity of two communicating enclaves and notifies during failures (e.g., crashes). Cronus's sRPC needs to be implemented only at mOS with this mechanism. This mechanism is also beneficial to TEE applications with frequent data sharing among enclaves (e.g., multi-party analytics [29]).

Direct enclave switching. Similar to previous TEE work [13], [71], Cronus currently leverages an mOS for managing the isolation of mEnclaves, which incurs costly enclave switching (and RPC, §IV-C). This can be optimized by a direct enclave switching mechanism that enables direct context switches from one enclave to another [72]. The OS can further be obviated from the burden of managing both accelerators and enclaves, by making OS manage only accelerators and making the secure monitor manage the isolation of enclaves.

C. Limitations and Future work

Cronus has three limitations. First, Cronus is currently implemented on ARM TrustZone, which cannot defend against physical attacks (e.g., memory bus [67]). To defend against physical attacks, Cronus can integrate ARM's memory encryption [105] and NVIDIA's Ampere Protected Memory [19] and NVIDIA H100's Secure Passthrough [3] for protecting RAM and bus traffic by paying some engineering efforts. Second, Cronus currently works on a single server and does not support heterogeneous computing in a distributed manner. However, by integrating with existing distributed resource scheduling techniques [43], [122], [76], Cronus can be extended to support distributed heterogeneous computing in the future. Third, Cronus currently works on discrete GPU widely used in the cloud, and supporting integrated GPU with unified CPU-GPU memory requires mechanisms for protecting the memory, an interesting future direction.

VIII. Conclusion

We presented the design and implementation of MicroTEE, the first TEE architecture that supports general heterogeneous accelerators, enables spatial sharing on one accelerator and enforces strong isolation across accelerators, enabling secure and high-performance heterogeneous computing for TEE. Cronus carries an *sRPC* protocol that eliminate unnecessary context switches for high-performance computing during execution and recover from failures securely and efficiently. Evaluation on two notable baselines using diverse workloads shows that Cronus is highly efficient on supporting diverse accelerators. Cronus's code is released on `https://github.com/hku-systems/cronus`.

Acknowledgment

We thank all anonymous reviewers for their helpful comments. This work is supported in part by a Huawei Flagship Research Grant in 2021, a HKU-SCF FinTech Academy R&D Funding Scheme in 2021, HK RGC GRF (17202318, 17207117), HK ITF (GHP/169/20SZ), the Pujiang Lab (Heming Cui is a courtesy researcher in this lab), the HKU and IS-CAS Joint Lab for Intelligent System Software.

Artifact Appendix

A. Abstract

Cronus can support general accelerators with fault isolation and security isolation. This artifact contains the source code of Cronus and its baseline systems, and the documentation on setting up and evaluating the systems. To ease the efforts of evaluating the artifact, we provide a machine with pre-installed hardware and software for the reviewers to evaluate the artifact in a short time.

B. Artifact check-list (meta-information)

- **Hardware:** The evaluation is done on a machine with an NVIDIA 2080Ti GPU, a high-performance CPU (e.g. Intel(R) Xeon(R) Silver 4116 CPU), and 128GB memory.

- **Program (software):** The experiment runs on Ubuntu 18.04 and requires CUDA-11.4; we bundled all source code of all other dependencies in our repository.

C. Description

CRONUS can be accessed via a permanent link at Zenodo and Github, as shown below.

- `https://github.com/hku-systems/cronus`
- `https://doi.org/10.5281/zenodo.7004365`

A detailed description of the repository is at `README.md`.

D. Installation

We provide a machine with pre-installed software and hardware for the reviewers, the reviewers can access the machine using the private key in the comments of hotcrp and following instructions in `docs/server.md`. We also provide steps for setting up CRONUS in a new machine at `docs/config.md`.

E. Experiment workflow

We provide detailed instructions on how to run the experiment at `README.md`.

F. Evaluation and expected results

1) Evaluation Claims

- CRONUS supports general accelerators (GPU, NPU) and programs (Experiment 1, Experiment 2 and Experiment 4)
- CRONUS incurs a low-performance overhead compared with OPTEE (TrustZone) (Experiment 1 and Experiment 4)
- CRONUS is much faster than HIX-Trustzone (Experiment 1)
- CRONUS's spatial sharing of GPU improves performance during resource (GPU) contention (Experiment 3)

2) Experiments

- Experiment 1: End-to-end performance of Rodinia
- Experiment 2: End-to-end performance of DNN Training
- Experiment 3: Spatial sharing of GPU in DNN training in OPTEE (TrustZone) and CRONUS
- Experiment 4: End-to-end performance of VTA-Bench in OPTEE (TrustZone) and CRONUS

See `README.md` for detailed instructions for running these experiments.

REFERENCES

[1] "Apache storm," http://storm.apache.org/.

[2] "muslc," https://nouveau.freedesktop.org/.

[3] "NVIDIA H100 Confidential Computing," https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/, accessed on 26/8/2022.

[4] "OPTEE OS," https://github.com/OP-TEE/optee_os.

[5] "T-sgx source code (github)." [Online]. Available: https://github.com/sslab-gatech/t-sgx

[6] "Civet: An efficient java partitioning framework for hardware enclaves," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/tsai

[7] R. Achermann, D. Cock, R. Haecki, N. Hossle, L. Humbel, T. Roscoe, and D. Schwyn, "mmapx: uniform memory protection in a heterogeneous world," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 159–166.

[8] F. Alder, J. Van Bulck, F. Piessens, and J. T. Mühlberg, "Aion: Enabling open systems through strong availability guarantees for enclaves," 2021.

[9] "Amazon ec2 f1 instances," https://aws.amazon.com/ec2/instance-types/f1/.

[10] AMD, "Amd secure encrypted virtualization (sev) development," http://developer.amd.com/amd-secure-memory-encryption-sme-amd-secure-encrypted-virtualization-sev/.

[11] F. M. Anwar, L. Garcia, X. Han, and M. Srivastava, "Securing time in untrusted operating systems with timeseal," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2019, pp. 80–92.

[12] "Apache tvm," https://tvm.apache.org/.

[13] ARM, "Security technology building a secure system using trustzone technology (white paper)."

[14] "Arm dynamiq shared unit," https://developer.arm.com/documentation/100453/0300/functional-description/l3-cache/l3-cache-partitioning.

[15] "Arm confidential compute architecture," https://www.arm.com/en/why-arm/architecture/security-features/arm-confidential-compute-architecture.

[16] N. Artenstein, "Broadpwn: Remotely compromising android and ios via a bug in broadcom's wi-fi chipsets," *Black Hat USA*, 2017.

[17] A. Aviram, S. Hu, B. Ford, and R. Gummadi, "Determinating timing channels in compute clouds," in *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, 2010, pp. 103–108.

[18] "Azure confidential computing," https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[19] "Azure confidential computing with nvidia gpus for trustworthy ai," https://azure.microsoft.com/en-us/blog/azure-confidential-computing-with-nvidia-gpus-for-trustworthy-ai/.

[20] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "CURE: A security architecture with customizable and resilient enclaves," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[21] A. Baumann, "Hardware is the new software," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017, pp. 132–137.

[22] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 29–44.

[23] R. Boivie and P. Williams, "Secureblue++: Cpu support for secure execution," *Technical report*, 2012.

[24] A. Borg, J. Baumbach, and S. Glazer, "A message system supporting fault tolerance," *SIGOPS Oper. Syst. Rev.*, vol. 17, no. 5, pp. 90–99, 1983.

[25] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, "DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization," in *Annual Computer Security Applications Conference (ACSAC)*. ACM, December 2019.

[26] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves." in *NDSS*, 2019.

[27] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich, "Verifying concurrent, crash-safe systems with perennial," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 243–258.

[28] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, 2014.

[29] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng *et al.*, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 2016, pp. 1789–1792.

[30] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation." in *USENIX Security Symposium*, 2016, pp. 857–874.

[31] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[32] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "Hybcache: Hybrid side-channel-resilient caches for trusted execution environments," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 451–468.

[33] "Device tree," https://elinux.org/Device_Tree_Reference.

[34] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[35] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[36] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. M. Tullsen, "Prime+abort: A timer-free high-precision L3 cache attack using intel TSX," in *26th USENIX Security Symposium, USENIX Security*, 2017, pp. 51–67.

[37] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan, "A framework for dynamically instrumenting gpu compute applications within gpu ocelot," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011, pp. 1–9.

[38] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Scalable memory protection in the PENGLAI enclave," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 275–294. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/feng

[39] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 287–305.

[40] A. Flink, "Apache flink," https://flink.apache.org/.

[41] X. Gong, P. Pi, and T. B. Team, "Exploiting qualcomm wlan and modem over the air," *Proceedings of the BlackHat USA 2019*, p. 58, 2019.

[42] "Google tpu," https://cloud.google.com/tpu.

[43] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic scheduling in multi-resource clusters," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 65–80.

[44] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *26th USENIX Security Symposium USENIX Security 17)*, 2017, pp. 217–233.

[45] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, "Failures in large scale systems: long-term measurement, analysis, and implications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[46] "Hafnium," https://opensource.google/projects/hafnium.

[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[48] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[49] T. Hunt, Z. Jia, V. Miller, A. Szekely, Y. Hu, C. J. Rossbach, and E. Witchel, "Telekine: Secure computing with cloud gpus," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 817–833.

[50] Intel, "Software guard extensions programming reference," https://software.intel.com/sites/default/files/329298-001.pdf.

[51] "Cache allocation technology," https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html.

[52] "Intel sgx sdk," https://github.com/intel/linux-sgx.

[53] "Intel trust domain extensions (intel tdx)," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[54] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 455–468.

[55] D. Ji, Q. Zhang, S. Zhao, Z. Shi, and Y. Guan, "Microtee: designing tee os based on the microkernel architecture," in *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 2019, pp. 26–33.

[56] J. Jiang, X. Chen, T. Li, C. Wang, T. Shen, S. Zhao, H. Cui, C.-L. Wang, and F. Zhang, "Uranus: Simple, efficient sgx programming and its applications," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 826–840.

[57] J. Jiang, S. Zhao, D. Alsayed, Y. Wang, H. Cui, F. Liang, and Z. Gu, "Kakute: A precise, unified information flow analysis system for big-data security," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC '17)*, 2017.

[58] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Fine-grained fault tolerance using device checkpoints," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 473–484.

[59] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class GPU resource management in the operating system," in *2012 USENIX Annual Technical Conference (USENIXATC 12)*, 2012, pp. 401–412.

[60] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE '12. New York, NY, USA: ACM, 2012, pp. 121–132. [Online]. Available: http://doi.acm.org/10.1145/2151024.2151042

[61] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 207–220.

[62] D. Korolija, T. Roscoe, and G. Alonso, "Do OS abstractions make sense on FPGAs?" in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, November 2020, pp. 991–1010.

[63] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[64] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ş. Teodorescu, C. Răducanu *et al.*, "Unikraft: fast, specialized unikernels the easy way," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 376–394.

[65] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek, "Pros: Light-weight privatized se cure oses in arm trustzone," *IEEE Transactions on Mobile Computing*, vol. 19, no. 6, pp. 1434–1447, 2019.

[66] Y. LeCun, C. Cortes, and C. J.C Burges, "The mnist database," http://yann.lecun.com/exdb/mnist/.

[67] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa, "An off-chip attack on hardware enclaves via the memory bus," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[68] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[69] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 523–539.

[70] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing," in *26th USENIX Security Symposium, USENIX Security*, 2017, pp. 16–18.

[71] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee, "Secloak: Arm trustzone-based mobile peripheral control," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, 2018, pp. 1–13.

[72] D. Li, Z. Mi, Y. Xia, B. Zang, H. Chen, and H. Guan, "Twinvisor: Hardware-isolated confidential virtual machines for arm," in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)*. ACM, 2021.

[73] M. Li, Y. Xia, and H. Chen, "Confidential serverless made efficient with plug-in enclaves," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 306–318.

[74] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 461–472.

[75] M. Malka, N. Amit, and D. Tsafrir, "Efficient intra-operating system protection against harmful DMAs," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 29–44.

[76] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 270–288.

[77] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "Rote: Rollback protection for trusted execution." *IACR Cryptology ePrint Archive*, vol. 2017, p. 48, 2017.

[78] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. ACM, 2016, p. 10.

[79] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen, "Skybridge: Fast and secure inter-process communication for microkernels," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–15.

[80] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017, pp. 69–90.

[81] P. Nasahl, R. Schilling, M. Werner, and S. Mangard, "Hector-v: A heterogeneous cpu architecture for a secure risc-v execution environment," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 187–199.

[82] "Nvidia cuda toolkit," https://developer.nvidia.com/cuda-toolkit.

[83] "Nvidia dgx systems," https://www.nvidia.com/en-us/data-center/dgx-systems/.

[84] "Nvidia multi-instance gpu," https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.

[85] "Nvidia multi-process service," https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[86] "Open enclave sdk," https://openenclave.io/sdk/.

[87] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh, "Nested enclave: Supporting fine-grained hierarchical isolation with sgx," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 776–789.

[88] O. M. Parkhi, A. Vedaldi, and A. Zisserman, "Deep face recognition," 2015.

[89] R. D. Pietro, F. Lombardi, and A. Villani, "Cuda leaks: a detailed hack for cuda and a (partial) fix," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 1, pp. 1–25, 2016.

[90] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa, "Visor: Privacy-preserving video analytics as a cloud service," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1039–1056.

[91] C. Priebe, K. Vaswani, and M. Costa, "Enclavedb: A secure database using sgx," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. IEEE, 2018, p. 0.

[92] "Pytorch c++ api," https://pytorch.org/cppdocs/.

[93] J. Qi, X. Chen, Y. Jiang, J. Jiang, T. Shen, S. Zhao, S. Wang, G. Zhang, L. Chen, M. H. Au *et al.*, "Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 18–34.

[94] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *Advances in neural information processing systems*, vol. 24, pp. 693–701, 2011.

[95] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[96] "Rodinia benchmark," http://lava.cs.virginia.edu/Rodinia/download_links.htm.

[97] V. A. Sartakov, L. Vilanova, and P. Pietzuch, "Cubicleos: a library os with software componentisation for practical isolation," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 546–558.

[98] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 38–54.

[99] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan, "Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors," in *Proceedings of the 17th ACM conference on Computer and communications security (CCS '10)*, 2017.

[100] T. Shen, J. Qi, J. Jiang, X. Wang, S. Wen, X. Chen, S. Zhao, S. Wang, L. Chen, X. Luo *et al.*, "SOTER: Guarding black-box inference for general neural networks at the edge," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 723–738.

[101] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with sgx enclaves," in *Proc. of the Annual Network and Distributed System Security Symp.(NDSS)*, 2017.

[102] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave programs," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1169–1184.

[103] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2435–2450.

[104] K. Tian, Y. Dong, and D. Cowperthwaite, "A full GPU virtualization solution with mediated pass-through," in *2014 USENIX Annual Technical Conference (USENIXATC 14)*, 2014, pp. 121–132.

[105] "Trustzone security features," https://developer.arm.com/architectures/architecture-security-features.

[106] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library os for unmodified applications on sgx," in *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[107] "Vta: Deep learning accelerator stack," https://tvm.apache.org/docs/vta/index.html.

[108] "Pci passthrough via ovmf," https://wiki.archlinux.org/title/PCI_passthrough_via_OVMF.

[109] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on gpus," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 681–696.

[110] https://github.com/apache/tvm/tree/main/vta/tests/python/integration.

[111] G. Wang, L. Zhang, and W. Xu, "What can we learn from four years of data center hardware failures?" in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 25–36.

[112] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2421–2434.

[113] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi, "Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v."

[114] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization," in *28th USENIX Security Symposium (USENIX Security 2019)*, 2019, pp. 675–692.

[115] Q. Wu and K. Lu, "On the feasibility of stealthily introducing vulnerabilities in open-source software via hypocrite commits," 2021.

[116] K. Xia, Y. Luo, X. Xu, and S. Wei, "Sgx-fpga: Trusted execution environment for cpu-fpga heterogeneous architecture," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 301–306.

[117] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 595–610.

[118] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "Antman: Dynamic scaling on GPU clusters for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 533–548.

[119] P. Xie, X. Ren, and G. Sun, "Customizing trusted ai accelerators for efficient privacy-preserving machine learning," *arXiv preprint arXiv:2011.06376*, 2020.

[120] M. Xue, K. Tian, Y. Dong, J. Ma, J. Wang, Z. Qi, B. He, and H. Guan, "gscale: Scaling up GPU virtualization with dynamic sharing of graphics memory space," in *2016 USENIX Annual Technical Conference (USENIXATC 16)*, 2016, pp. 579–590.

[121] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

[122] Z. Yang, C. Yin, and Y. Liu, "A cost-based resource scheduling paradigm in cloud computing," in *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 2011, pp. 417–422.

[123] M. H. Yun and L. Zhong, "Ginseng: Keeping secrets in registers when you distrust the operating system." in *NDSS*, 2019.

[124] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "$sectee$: A software-based approach to secure enclave architecture using tee," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1723–1740.

[125] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform." in *NSDI*, 2017, pp. 283–298.

[126] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building verifiable trusted path on commodity x86 computers," in *2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 616–630.

[127] J. Zhu, R. Hou, X. Wang, W. Wang, J. Cao, B. Zhao, Z. Wang, Y. Zhang, J. Ying, L. Zhang *et al.*, "Enabling rack-scale confidential computing using heterogeneous trusted execution environment," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1450–1465.

[128] Z. Zhu, S. Kim, Y. Rozhanski, Y. Hu, E. Witchel, and M. Silberstein, "Understanding the security of discrete gpus," in *Proceedings of the General Purpose GPUs*, 2017, pp. 1–11.