IronBus: A Decentralized Trusted Bus for Accelerators

A HW-SW Co-Designed Trusted Acceleration Interface

HARSHAVARDHAN UNNIBHAVI, Technical University of Munich, Germany
JULIAN PRITZI, Technical University of Munich, Germany
ATSUSHI KOSHIBA, Tokyo University of Science, Japan
NUNO SANTOS, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
PRAMOD BHATOTIA, Technical University of Munich, Germany

Cloud providers increasingly deploy accelerators like GPUs and TPUs for AI and data analytics. To secure these devices, emerging trusted I/O architectures (e.g., Intel TDX-Connect, AMD SEV-IO) extend the host CPU's trusted domain to the accelerator fabric. However, these CPU-centric designs have three limitations: (1) *TCB bloat:* any guarantee about where data flows rests on a full guest OS; (2) *Coarse-grained control:* lacking application-aware data flow policies; and (3) *Centralized bottlenecks:* host-mediated peer-to-peer traffic puts the host on every transfer's data path.

To address these challenges, we propose a *decentralized trusted bus* that provides attested, hardware-protected channels for direct data movement between accelerators. IronBus realizes this abstraction through a *cross-layer hardware/OS/compiler co-design* for next-generation interconnect fabrics. IronBus shifts access control from the centralized host to a per-device *Attested Interface Unit (AIU)*—a minimal silicon Root-of-Trust that enforces security policies at the fabric endpoints. Orchestrating these units is a *Hardware-Abstraction Layer (HAL)*—a hardware-proximate *lean software layer with formally verified protocols* that replaces the guest OS to bootstrap trust. IronBus also introduces a *DFG-based programming model* and *policy compiler*: developers express multi-accelerator pipelines as *Device Flow Graphs (DFGs)*, from which the compiler automatically synthesizes *Security Service Level Agreements (SSLAs)*, and the HAL signs *execution proofs* for auditing.

We prototype IronBus on a many-core RISC-V platform, *evaluate it in gem5*, and *formally verify its protocols with Tamarin and CryptoVerif*. The IronBus HAL is as small as the SEV and TDX firmware, whereas for their data-flow policy host-centric designs trust a guest OS and firmware that are together 3–94× larger. Direct device-to-device channels make IronBus 1.1–2× faster than host-centric designs on accelerator chains, collectives, and ML workloads, even when every device has its own host link. The cryptographic bus costs 5–23% on communication microbenchmarks and at most 10% on ML workloads over an unprotected baseline.

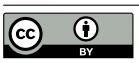
CCS Concepts: • **Security and privacy** → **Security in hardware**; **Systems security**.

Additional Key Words and Phrases: Trusted I/O, confidential computing

1 Introduction

The rapid proliferation of data-intensive workloads, particularly in AI and genomics, has led cloud providers to integrate a diverse array of hardware accelerators—such as GPUs, TPUs, and FPGAs—into the modern datacenter [25, 29, 64, 92, 102, 105]. While these accelerators deliver the high-bandwidth throughput required for data-intensive workloads, their integration fundamentally

Authors' Contact Information: Harshavardhan Unnibhavi, harshavardhan.unnibhavi@tum.de, Technical University of Munich, Garching, Germany; Julian Pritzi, julian.pritzi@tum.de, Technical University of Munich, Munich, Germany; Atsushi Koshiba, koshiba@rs.tus.ac.jp, Department of Information Science and Technology, Tokyo University of Science, Tokyo, Japan; Nuno Santos, nuno.m.santos@tecnico.ulisboa.pt, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal; Pramod Bhatotia, pramod.bhatotia@cit.tum.de, Technical University of Munich, Munich, Germany.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 1544-3973/2026/11-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

expands the system's attack surface—especially for sensitive personal and medical datasets—by introducing high-speed data paths that bypass traditional CPU-centric security boundaries. In shared cloud environments, this architectural shift demands a security model capable of shielding data not only from compromised software stacks and malicious administrators on the host CPU, but also from security attacks targeting the accelerators [22, 58] and the underlying interconnect fabrics via physical [53, 65] and software-initiated means [37, 83, 84]. Consequently, as data-protection regulations such as GDPR [35], the EU AI Act [30], and CCPA [87] impose strict obligations and steep penalties for mishandling sensitive personal and medical data, robust, hardware-rooted security is no longer a peripheral concern; it is a *foundational architectural requirement for the next generation of cloud-based acceleration.*

To meet these security requirements, industry-led efforts are now focusing on extending existing CPU-based isolation mechanisms, such as Trusted Execution Environments (TEEs), to the broader I/O subsystem and its connected accelerators. Emerging *trusted I/O architectures*, such as Intel TDX-Connect [44], AMD SEV-IO [5], and Arm CCA I/O [7], are striving to extend the CPU-centric Confidential Virtual Machine (CVM) boundary to *external accelerators*. These frameworks utilize the TEE Device Interface Security Protocol (TDISP) and PCIe Integrity and Data Encryption (IDE) to establish cryptographically secure host-to-device tunnels [76].

In these designs, the TEE Security Manager (TSM) serves as the centralized orchestration point, coordinating with the host IOMMU to enforce memory-level access control, while a device-side Device Security Manager (DSM) manages the Trusted Device Interface (TDI) state machine [3, 8, 47]. However, by maintaining a CPU-centric mediation model, these upcoming architectures introduce three fundamental security and architectural bottlenecks:

- **TCB bloat:** The hardware enforces only that a device belongs to a CVM; where data flows within and out of the CVM is decided by the guest OS's privileged driver stack, so any guarantee about data flow rests on a full guest OS, millions of lines of non-security-critical code [66].
- **Coarse-grained policy enforcement:** Trusted I/O protocols such as TDISP and SPDM [21] are restricted to link-layer authentication and session establishment. They lack the primitives for fine-grained, application-aware data flow control, leaving the system unable to restrict data movement or prevent misuse once the physical link is authorized.
- **Centralized performance bottleneck:** Enforcing a host-mediated communication model requires all peer-to-peer (P2P) traffic to be routed through the host's root complex for access control [44, TXN-11, DSM-5], [5, Table 8.13.1]. This forces all interconnect transactions through the host's root complex, a centralized bottleneck for security enforcement that imposes significant latency and throughput penalties on high-bandwidth interconnects.

To overcome these limitations, we target the following research question: *How can we co-design a trusted I/O architecture that decentralizes access-control enforcement to hardware endpoints and automates fine-grained policy synthesis—eliminating the host-mediation bottleneck and TCB bloat?*

To address these challenges, we introduce the ***decentralized trusted bus abstraction***: a cryptographically secured, hardware-enforced logical link that facilitates direct data movement between attested endpoints. Unlike traditional host-mediated I/O, these channels carry the security policy, identities, permissions, and keys, in the fabric itself.

IronBus realizes this abstraction through a *cross-layer hardware/OS/compiler co-design* built around three tightly integrated components. First, the *Attested Interface Unit (AIU)* shifts access control from the centralized host to the interconnect edge: a per-device, fixed-function silicon Root-of-Trust with a dedicated cryptographic engine that hardware-enforces per-flow policies directly at the fabric (§4). Second, the *Hardware-Abstraction Layer (HAL)* orchestrates these distributed

AIUs as a lean, hardware-proximate software layer with formally verified protocols, bootstrapping system-wide trust and managing session keys while remaining entirely decoupled from the high-speed data path (§5).

Third, to bridge the *semantic gap* between application-level security intent and low-level hardware policy, IronBus introduces a *DFG-based programming model* and *policy compiler* (§6). Developers express multi-accelerator pipelines as *Device Flow Graphs (DFGs)*; the compiler automatically synthesizes *Security Service Level Agreements (SSLAs)* from the graph topology and instantiates the corresponding trusted channels. Upon completion, the system produces *cryptographically signed execution proofs* for third-party compliance auditing. Crucially, no single layer suffices in isolation: the AIU supplies the hardware enforcement primitive, the HAL bootstraps and orchestrates trust across devices, and the policy compiler closes the gap to application intent—so trustworthy acceleration is inherently cross-layer.

We prototyped IronBus's functionality on M^3 [12] and assessed its performance in a cycle-accurate gem5 simulation [14]. Our implementation integrates the AIU as a hardware-extended memory controller interface and runs the HAL in a privileged environment on a dedicated management tile. Our evaluation shows that the hardware-proximate HAL is as small as the SEV and TDX firmware, whereas for their data-flow policy host-centric designs trust a guest OS and firmware that are together 3–94× larger. By enforcing access control at the AIUs instead of the host, IronBus removes the host from the data path and is 1.1–2× faster than current host-centric trusted I/O designs on accelerator chains, collective communication, and ML workloads with up to 16 accelerators, even when every device has its own link to the host. The cryptographic bus costs 5–23% on communication microbenchmarks and at most 10% on ML workloads over an unprotected baseline, a security tax that shrinks further on off-chip and disaggregated fabrics.

2 Background and Motivation

2.1 Trusted I/O Architectures

Emerging trusted I/O architectures—e.g., Intel TDX-Connect [44], AMD SEV-IO [5], and Arm CCA IO [7]—aim to extend hardware-rooted CPU security to external accelerators. As illustrated in Figure 1, these systems rely on a centralized orchestration model to bridge the host-side trusted domain with device-side endpoints.

Confidential Virtual Machines (CVMs). Built on CPU-centric *Trusted Execution Environments (TEEs)* [4, 7, 46], CVMs provide a hardware-encrypted memory boundary that isolates guest workloads from privileged but untrusted host software (e.g., the Hypervisor). While CVMs maintain high system compatibility by supporting a full guest OS, this design significantly bloats the Trusted Computing Base (TCB) [66]. The entire guest OS is thus on the security-critical path; any compromise of the CVM's kernel threatens the confidentiality of data on the trusted I/O channels.

TDISP and the control plane. To extend the CVM boundary, these architectures adopt the *TEE Device Interface Security Protocol (TDISP)* [76]. The host-side *TEE Security Manager (TSM)* manages the lifecycle of the *Trusted Device Interface (TDI)* by coordinating with the device-side *Device Security Manager (DSM)*. This process begins with attestation via the *Security Protocol and Data Model (SPDM)* [21], which establishes a secure session for exchanging cryptographic keys. However, TDISP is fundamentally a link-layer protocol; it focuses on establishing the physical link but lacks the architectural primitives for fine-grained, application-aware usage policies. Once the link is authorized, the architecture lacks native mechanisms to prevent data misuse within the device.

Data plane protection and IDE. For data-plane confidentiality, these frameworks use *PCIe's Integrity and Data Encryption (IDE)* [76]: an IDE stream encrypts the payload and integrity-protects the entire TLP, header included, between two endpoints. Access control is left to the host IOMMU [3,

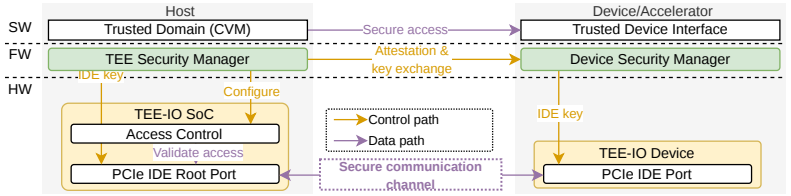


Fig. 1. State-of-the-art trusted I/O [5, 7, 44].

8, 47], which translates and checks every device address in the root complex. *Selective IDE* runs the stream directly between two devices for end-to-end encryption, but takes the IOMMU off the path: since an IDE-protected header cannot be rewritten in transit, the requester must put the peer's final physical address in the request, i.e., issue PCIe *translated* requests, which require *Address Translation Services (ATS)*. Such a request is admitted on the requester's word, so a compromised device can mark any physical address as translated and access all system and device memory [39, 58, 61, 78]. Selective IDE thus requires arbitrary memory access by design: what makes it end-to-end removes the only access-control point. Existing defenses disable ATS or re-check translated requests at the root complex [3, 7, 8, 44], which direct transfers never traverse, so today's architectures route all P2P traffic through the host [5, 44]. *Securing direct P2P requires a per-request permission check at the device that owns the memory, which no architecture provides.*

2.2 The Case for a Decentralized Trusted Bus

Current trusted I/O implementations achieve isolation by composing PCIe Access Control Services (ACS), the host IOMMU, and link-layer IDE. By design, ACS disables direct peer-to-peer (P2P) communication, so all traffic crosses the host's root complex, where the IOMMU centralizes access control. This host-centric model (§2.1) simplifies the security state machine at the expense of performance and flexibility, and has three fundamental limitations:

TCB bloat. Anchoring secure channels in CVMs places a massive, non-security-critical software stack in the trusted domain. This stack is not needed to isolate tenants and devices from each other, but it decides where a tenant's data flows, which distinguishes two TCBs. The *isolation TCB*, whose compromise breaks isolation between tenants and devices, is the small TSM firmware [5, 44]. The *flow-policy TCB* is the software whose compromise lets data leave the flows an application declared. In today's host-centric designs, it is the guest OS: TDISP binds a device to a whole CVM [76], and inside the CVM, the guest kernel programs which memory each device can access [5, 44] and can itself read and copy any buffer, including into memory shared with the host [66] (§7.2).

Coarse-grained policy enforcement. Centralizing access control at the host prevents the enforcement of fine-grained, component-level restrictions on data processing. Existing hardware primitives are too coarse to restrict data flows to specific software versions or temporal windows at the device level. This architectural rigidity makes it impossible to generate verifiable, cryptographically rooted proofs of compliance required by stringent regulations such as HIPAA [94] or GDPR [35]. Without device-level policy awareness, the system cannot verify how data is used once a link is authorized.

Performance loss due to host mediation. Centralized access control precludes efficient P2P data movement. Because Selective IDE is non-viable due to the security risks of Address Translation Services (ATS) (§2.1), current standards—such as TDX-Connect [44] and SEV-IO [5]—route all P2P transactions through the host for IOMMU translation and access control. This lack of end-to-end encryption forces a reliance on hop-by-hop link encryption, doubling the latency of device-to-device traffic. Consequently, these CVM-based architectures incur a 2× performance penalty for accelerator chains of length 2–4 compared to the direct, secure P2P paths of IronBus (§7.3).

All three limitations share a cause: the host IOMMU is the only access-control point, and it is configured per CVM. Hence where data flows is left to the guest OS (limitation 1), no policy is finer than a whole device (limitation 2), and every P2P transfer detours through the host (limitation 3). *Moving access control to per-device endpoints removes all three.*

2.3 SSLA-Compliant Hardware Acceleration

Clients that process regulated data on cloud accelerators need assurance that only approved hardware and software touch their data, that it moves only where intended, and that this holds for the required time. We call the agreement that fixes these conditions between a client and a cloud provider a *security service level agreement (SSLA)*. An SSLA has four kinds of attributes: *hardware attributes* constrain execution to devices with given characteristics, e.g., vendor and model, geographic region, or TEE support; *software attributes* restrict which tasks or OS services may handle data; *data attributes* require integrity, authenticity, or confidentiality of a data flow; and *temporal attributes* constrain when data may be used, e.g., not past an expiration time under GDPR or not deleted before a retention deadline under HIPAA [94].

Consider an application for image classification using a GPU and a TPU (Figure 2). It consists of two jobs. The initialization job allocates the two accelerators and loads an ML model onto the TPU. The classification job runs in two stages: first the GPU loads an image from the file system, resizes it, and forwards it to the TPU; then the TPU classifies the image with the model and stores the result in the file system. An SSLA for this application may require that the GPU and TPU be of a given vendor and model, located in a given region, and allocated exclusively (hardware); that only the developer's resize and classification tasks and a certified file service handle the images (software); that images flow only from the GPU to the TPU and results only to the file service, encrypted and integrity-protected (data); and that nothing flows after an upper time bound set for the job (temporal).

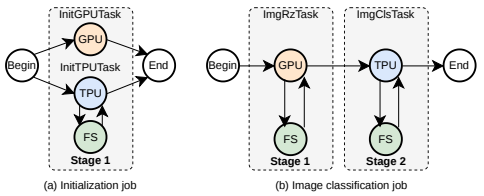


Fig. 2. DFGs for an image processing application.

Such an application generalizes to a *Device Flow Graph (DFG)*: *nodes* are the devices and system services that execute the application's *tasks*, *edges* are the data flows between them over the system bus, and a job is a sequence of *stages*, each a synchronization point across its tasks; Figure 2 shows the two jobs above as DFGs. The SSLA is applied to this graph: hardware and software attributes are *node attributes*, since they constrain the devices and the tasks that run on them; data and temporal attributes are *edge attributes*, since they constrain the flows between nodes.

An SSLA is only as good as its enforcement: devices and software must be attested and checked before use, transfers on the bus confined to the declared edges with confidentiality and integrity, closed at the time bound, and all of it proven to a third party. Host-centric trusted I/O offers none of this below the granularity of a whole device assigned to a CVM (§2.1). IronBus enforces node attributes when it attests and allocates devices and edge attributes on every transfer at the devices themselves, and signs a proof of both (§3).

2.4 Background on the M³ Platform

We evaluate IronBus on the M³ platform [12], a hardware/OS co-design that exemplifies the shift toward heterogeneous accelerator architectures. M³ provides an ideal architectural substrate for our work due to two characteristics that distinguish it from traditional host-centric systems:

(a) Standardized hardware-interconnect shims. M³ utilizes a Data Transfer Unit (DTU)—a hardware shim integrated with every system component, including CPUs, accelerators, and memory

controllers. These DTUs are interconnected via a Network-on-Chip (NoC) fabric, providing a standardized communication interface across heterogeneous tiles.

(b) Capability-based decentralized management. Unlike traditional systems that rely on a centralized IOMMU managed by a bloated kernel, M³ utilizes a capability-based resource management model. A microkernel on a dedicated tile holds *capabilities*—permissions for memory regions and communication channels—and is the only software that can install them into the *endpoints* of a DTU. Every request is checked against them: an outgoing request is admitted only if the tile holds a capability for its target and the request stays within that capability's region and permissions, and the DTU attaches the capability to the request so that the receiving DTU verifies it against its own endpoints before accepting it. Creating, deriving, delegating, and revoking capabilities are thus endpoint configurations by the kernel. Capabilities thus enable access control at the device side, on every request, instead of at a central IOMMU.

Security research gaps in decentralized substrates. While M³ provides the structural foundation for decentralized communication, it lacks the primitives required for trusted acceleration. Specifically, it lacks: (1) *Hardware attestation* to verify device identity; (2) *End-to-end cryptographic protection* for data in transit across the NoC; and (3) *Semantic policy enforcement* to ensure data flows comply with high-level security agreements.

While our IronBus prototype leverages these M³ primitives to fill these research gaps, the IronBus design remains functionally independent and transferable to emerging standards like TDISP-compatible PCIe and CXL fabrics.

2.5 System Model and Assumptions

This *decentralized architecture* moves the Root-of-Trust from the host CPU to per-device endpoints, with a security boundary that covers both a compromised host and an untrusted interconnect.

Physical and interconnect-layer. We assume the integrity of the CPU/accelerator die, but consider all off-chip interconnects and external links untrusted. We consider an adversary that may probe or modify bus traffic. IronBus defends against these attacks by enforcing end-to-end authenticated encryption via the AIUs. We do not claim protection against invasive die-level attacks, which require orthogonal hardening (e.g., encrypted DRAM controllers).

Non-physical and privileged software. We assume a strong software adversary, including malicious co-tenants and privileged administrators (e.g., the hypervisor). Adversaries may attempt to: (1) bypass isolation, (2) tamper with P2P computations, or (3) forge policy attributes to circumvent SSLA restrictions. IronBus mitigates these by keeping the host OS and hypervisor out of the trusted path, rooting enforcement in the IronBus HAL and distributed AIUs.

Assumptions and scope. We assume the Certificate Authority (CA) and AIU private keys remain uncompromised, protected by tamper-resistant hardware. Key revocation and re-keying is supported via standard revocation lists maintained by the HAL. *Out-of-scope* threats include Denial-of-Service (DoS) [15], microarchitectural side-channels [50, 55, 82], and supply-chain attacks [98, 99], all of which require orthogonal hardware or infrastructure-level mitigations.

3 Overview

To address the architectural requirements for secure, high-performant acceleration, IronBus is guided by *three core design goals*: (1) **TCB minimization**, which keeps complex, general-purpose software out of the flow-policy TCB by relying on *a minimal silicon Root-of-Trust (RoT)* and a *lean trusted software layer*; (2) **Verifiable fine-grained enforcement**, providing hardware-anchored mechanisms to *specify and enforce fine-grained data flow policies* while producing *auditable execution proofs*; and (3) **Secure heterogeneous P2P**, enabling *direct, line-rate interoperation between diverse accelerators* without relying on host-CPU mediation.

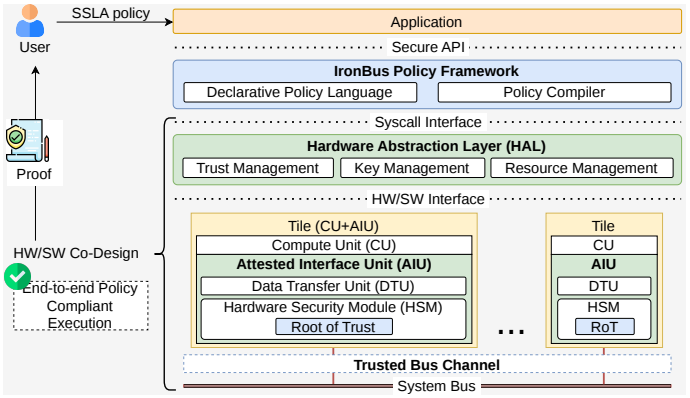


Fig. 3. IronBus HW-SW co-designed architecture.

To achieve these design goals, we introduce **Trusted Bus Channels** as the primary architectural abstraction: cryptographically secured, hardware-enforced logical links for direct data movement between authenticated endpoints. Unlike host-mediated I/O, each channel encapsulates its security policy (identities, permissions, and cryptographic state) and its data-plane protection. Treating the interconnect as a set of discrete, policy-bound channels lets applications instantiate complex logic dynamically under fine-grained data-flow policies.

IronBus realizes the trusted bus abstraction through a three-layered architecture designed to provide verifiable isolation across heterogeneous accelerators in a decentralized setup (Figure 3):

1. Attested Interface Unit (AIU) (§4): The AIU is a *per-device hardware extension* that serves as the decentralized root of trust. It augments the device's interconnect interface with a *Hardware Security Module (HSM)* containing a silicon Root-of-Trust (RoT) and a private keystore. Offloading encryption and access control to the AIU secures the data plane against physical interconnect attacks and keeps P2P movement within a hardware-enforced trusted domain.

2. Hardware Abstraction Layer (HAL) (§5): The HAL is a *thin software layer* that orchestrates the system's security primitives through formally verified protocols. Operating on a privileged management tile, the HAL bootstraps system-wide trust and manages the lifecycle of *trusted bus channels*. It translates abstract security policies into localized hardware configurations—specifically endpoint mappings and cryptographic keys—maintaining a minimal TCB by *strictly limiting functionality to resource delegation and security orchestration.*

3. SSLA programming framework (§6): The framework enables developers to specify application logic as *Device Flow Graphs (DFGs)*. It includes *a policy compiler* that infers data flow restrictions and generates corresponding *Security Service Level Agreements (SSLAs)*. During execution, the framework interacts with the HAL to dynamically instantiate trusted channels reflecting the DFG topology. Upon completion, the system provides *cryptographically signed proofs*, allowing third-party auditors to verify that the hardware execution strictly adhered to the specified policies.

Through these components, IronBus resolves the bottleneck of host-centric designs by decoupling policy decisions from policy enforcement. The HAL, running on a single privileged tile, acts solely as the decision maker: it attests devices, distributes keys, and configures the AIU endpoints for the SSLA that the programming framework produced, once per job. Enforcement, however, is fully decentralized: every AIU independently checks and encrypts its device's transfers in hardware. IronBus thus keeps trust orchestration centralized and data movement decentralized.

4 Attested Interface Unit (AIU)

To secure the I/O path between CPUs and accelerators—collectively referred to as **Compute Units (CUs)**—we design the Attested Interface Unit (AIU). The AIU is a minimal, accelerator-agnostic hardware extension that enables bootstrapping trust into the CU and establishing *trusted bus channels*. By tightly coupling each CU with an AIU to form a **Tile**, we shift security enforcement to the device side, thereby building a decentralized, trusted bus architecture.

4.1 Trusted Bus Channels

The AIU enables establishing secure communication channels via *trusted bus channels*. For enhanced security, IronBus ensures that only the trusted HAL (explained in § 5) is authorized to maintain these channels. IronBus classifies tiles (AIU+CU) into two types: *HAL tiles* running privileged code and *user tiles* running unprivileged (application) code. We design the AIU to satisfy two requirements (R*i*) and adhere to two design principles (D*i*) as follows:

R1: HAL and user tiles must have distinct attestation roles; HAL tiles are securely booted and certified by a trusted third party, and only trusted HAL tiles can attest user tiles.

R2: AIUs must distinguish HAL tiles, which are privileged to attest tiles and establish trusted channels, and user tiles, which only use HAL-configured channels. The AIUs must grant or deny channel access based on tile privileges.

D1: The AIUs must not reuse a cryptographic key for multiple purposes to reduce exposure to cryptanalysis attacks [67].

D2: All secret symmetric or private asymmetric keys must remain within the hardware layer, with access limited to the trusted HAL to ensure secure key management.

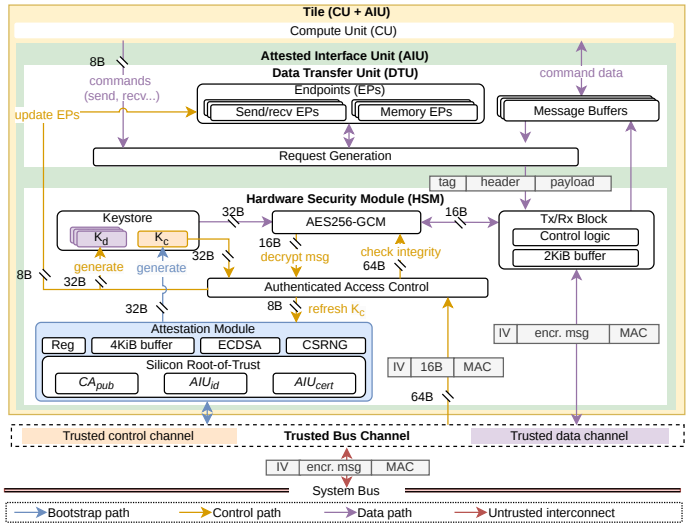
Depending on privileges, trusted bus channels are classified as: (*i*) *trusted control channels* managed/used by the HAL, and (*ii*) *trusted data channels* used by applications.

— **Trusted control channels** are used by the HAL to establish trusted data channels. Consider a system with n tiles: CPU (C), accelerator (A), and memory (M) with a single HAL T_H and many user tiles T_U. The HAL first establishes trusted control channels c between T_H and each T_U. The AIU of T_H facilitates this through attestation requests to T_U. If the attestation quote is invalid (e.g., signed using an invalid certificate), no control channel is formed with T_U, effectively blocking it. Valid attestation creates a unique symmetric *control channel key* K_c for encrypting the interconnect communication. These keys, stored in the AIUs of T_H and T_U, are only accessible by the HAL (**D1** + **D2**). Each tile's privileges are set by a flag: `H` (HAL tile), `UC` (user CPU tile), `UA` (user accelerator tile), or `M` (memory tile), ensuring that only the tile flagged `H` can process attestation requests and accept quotes (**R1**).

— **Trusted data channels** enable secure communication across CUs. For example, if an application on T_{U_1} needs to offload computation to a GPU T_{U_2} the HAL (T_H) establishes a trusted data channel between T_{U_1} and T_{U_2}. Using control channels c_{U_1} and c_{U_2}, the AIU of T_H coordinates with T_{U_1} and T_{U_2} to generate a dedicated encryption key K_d for secure data exchange. A channel handle is then created and assigned to the application on T_{U_1}, allowing secure communication with T_{U_2} through their AIUs. Each channel is secured with unique keys K_d that remain within the AIUs (**D1** + **D2**). Access to these channels is limited to tiles marked with `UC`, `UA`, or `M`, based on permissions set by the HAL (**R2**).

4.2 AIU Hardware Architecture

IronBus enables *direct P2P communication* by offloading access control and sender verification to the receiving AIUs: the centralized HAL only configures endpoints and their keys, and during communication the receiving AIU verifies each message's origin, so traffic is routed directly without a centralized host. The AIU (Figure 4a) borrows M³'s DTU as its unified communication



(a) Hardware architecture.

Key	Storage	Use
CA_{pub}	Each AIU	CA key to verify AIUs
AIU_{id}	Each AIU	Unique key pair for quotes
AIU_{cert}	Each AIU	Certificate signed by CA
K_{boot}	HAL's AIU	Key pair from secure boot
K_{cert}	HAL's AIU	Boot certificate chain
K_c	Each AIU	Control channel key
K_d	Each AIU	Data channel key

(b) Cryptographic primitives.

Fig. 4. The AIU: (a) its hardware architecture and (b) its cryptographic primitives.

interface [12] and extends it with the HSM, which contains a silicon RoT, a private keystore, and cryptographic accelerators, to establish trusted bus channels.

Data Transfer Unit (DTU). The DTU provides the CU with a register interface and message buffers for command submission and data exchange. Applications use the interface to send/receive messages or access memory via DMA. The DTU exposes hardware endpoints for the HAL to manage on-CU resources and establish communication channels. Any data passed through the DTU is encrypted and verified by the HSM.

Silicon Root-of-Trust (RoT). To enable bootstrapping trust into tiles, we include a hardware RoT in the HSM along with the cryptographic material listed in Figure 4b. A trusted CA verifies the AIU's authenticity and connection to the CU, which has the public key CA_{pub} and a unique asymmetric key pair AIU_{id}. The HAL uses the CA-signed certificate AIU_{cert} to bootstrap trust in the CUs (§ 5.1). CA_{pub}, AIU_{id}, and AIU_{cert} are burned into each AIU at manufacturing.

Attestation module. The attestation module allows the HAL to bootstrap trust into the AIU (§ 5.1). The module contains a 64-bit register and 4 KiB data buffer accessible over MMIO by the HAL. The HAL uses the buffer to (i) bootstrap trust and generate the control key (K_c), and (ii) relinquish exclusive configuration access to the AIU. The HAL and attestation module use the buffer to exchange attestation data, such as certificates and challenges. The module includes an ECDSA engine for verifying certificates and responding to challenges, and a cryptographically secure pseudorandom number generator (CSRNG) for generating nonces.

Keystore. The HSM has a private memory region inaccessible to applications to store control and data channel keys (K_c, K_d). Only K_c is not configured directly by the HAL; instead, the HAL reestablishes trust (§ 5.1) to refresh K_c. The AIU uses K_d to encrypt data on the interconnect.

Authenticated access control module. This module ensures that only the HAL can configure the AIU's endpoints and keystore. It verifies that all MMIO access to an AIU originates from the HAL's AIU via the trusted control channel. Upon incoming configuration requests, the AIU checks the integrity of the requests through a decryption module (AES256-GCM) and routes them to the target location if the check passes. Otherwise, it drops the request and informs the HAL. Only the HAL and target AIU share the control channel's symmetric key to make configuration requests.

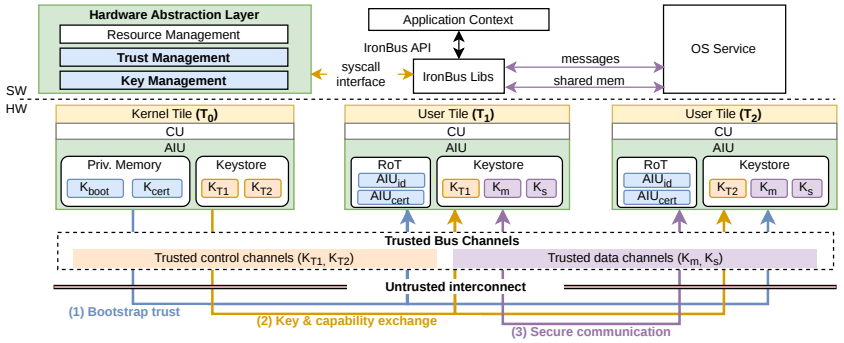


Fig. 5. The HAL (1) bootstraps trust and (2) exchanges keys with AIUs; (3) data transfers bypass it.

Message passing via trusted data channels. These channels are structured as *send/receive* endpoints with encryption keys. Each message includes a header with metadata (e.g., length) and a payload. Upon transmission, the AIU holds a message in a 2KiB buffer, encrypts it, adds the initialization vector (IV), and appends an authentication tag (MAC) using the AES256-GCM module and the endpoint's key. On reception, the AIU decrypts the message and verifies its integrity. Since we use AES-GCM, these operations are performed in parallel. The message is placed into a receive buffer for application access upon successful integrity verification.

Memory access via trusted data channels. These are structured as *memory endpoints* with keys. They are used to issue DMA requests to CU's external memory, provided that the memory side has sufficient endpoint storage. When reading data, the AIU encrypts requests, containing the memory size and offset, and sends them to the memory-side AIU. The memory AIU decrypts the request, retrieves and encrypts data from memory, and sends it to the requesting AIU. The receiving AIU decrypts the data and writes it to the CU's internal memory. Writing to memory works analogously.

5 Hardware Abstraction Layer (HAL)

The HAL is a hardware-proximate software layer that manages CU resources through the AIU with a minimal TCB and formally verified protocols. It runs the security lifecycle—bootstrapping trust, establishing secure channels, and deploying application logic—and stays off the data path.

5.1 HAL Architecture

Figure 5 shows the HAL architecture, where the HAL runs on a dedicated HAL tile (T_0), and applications and OS services run on user tiles (T_1, T_2). The HAL enforces SSLA policies at the hardware level by building on M^3's capability-based resource management (§2.4), which the AIU turns into device-side authenticated access control: every request is checked against an endpoint that also carries a channel key and the identity of its attested peer (§4). A policy that the HAL configures once into the endpoints of the two nodes of an edge is therefore enforced on every transfer, without the HAL on the data path. Without such device-side access control, the check would fall back to the host IOMMU, i.e., the host-centric design, with the guest OS in its flow-policy TCB (§2.2) and its slowdown (§7.3, §7.4). On top of this, the HAL introduces two new features. First, *trust management* allows the HAL to bootstrap trust into user tiles and services using the AIU's hardware RoT. For this, the HAL generates control channel keys (K_{T_1}, K_{T_2}) shared between HAL and user tiles and creates trusted control channels used for key management, an operation permitted only to the HAL. Second, *key management* enables the HAL to securely generate, distribute, and store data channel keys (K_m, K_s), which user tiles use for encrypted communication and memory access; keys never leave the HAL and the AIUs.

Secure boot. IronBus ensures a secure HAL boot using a three-stage process (BL0, BL1, HAL) that enables updates to BL1 and HAL. Each stage includes an executable and a CA-signed certificate. The previous stage validates the certificate, records the executable's hash, generates an asymmetric key pair, and passes it to the next stage. Finally, the HAL has the hashes of all stages, an asymmetric key pair (K_{boot}), and a three-stage certificate chain (K_{cert}) stored in AIU-private memory and is used to prove the HAL's integrity and the presence of a trusted AIU to remote challengers.

Trust management. To establish trust in CUs and control channels, the HAL uses a trusted protocol with the AIU's hardware RoT (Figure 6). The HAL sends the AIU a nonce (n_{HAL}), Diffie-Hellman key material, and a certificate chain (K_{cert}). The AIU verifies it, responds with its nonce (n_{AIU}), certificates (AIU_{cert}), and key material, all signed by AIU_{id}. After mutually verifying certificates and nonces, both derive a symmetric key (K_c).

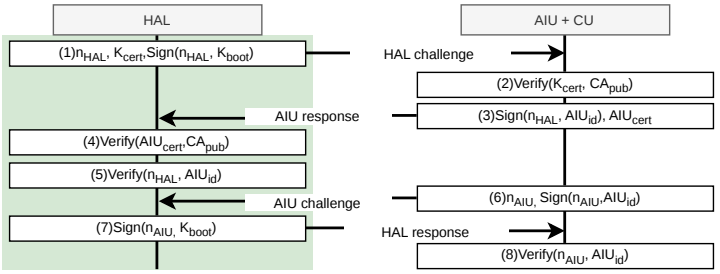


Fig. 6. Protocol for creating trusted control channels.

Key management. The HAL manages keys for both the trusted control channel and data channel. Firstly, it manages the unique per-AIU control channel keys (K_c). Secondly, it is involved in the exchange and revocation of data channel keys (K_d). This is because data channels for message-passing and memory channels are explicitly configured, managed, and shared, as capabilities between application contexts and OS services by the HAL. Key management operations such as creation, exchange, and revocation map one-to-one with the respective capability operations.

Remote attestation. Users attest the authenticity of the HAL and AIUs before deploying applications. The IronBus network-accessible deployment service handles this by responding to a user's challenge (with a nonce) using the HAL's *measure* system call. This returns the HAL's certificate chain (K_{cert}), the AIU_{cert}, and the signed nonce. The user verifies this data to ensure the expected hardware and software are present and then proceeds with deployment. Additionally, we assume that all AIUs are signed by the same CA, and a trusted HAL implies trust in all AIUs.

Communication between user tiles. For two user tiles, the HAL acts as key distributor: it generates a symmetric key and configures both tiles' AIUs over the encrypted control channels.

CU-level isolation. IronBus governs only the data flows that cross the interconnect boundary, i.e., those entering and leaving an accelerator. Isolating co-resident contexts *within* a single accelerator—e.g., preventing one tenant from reading another tenant's on-device memory state—is the responsibility of accelerator-internal partitioning such as NVIDIA MIG [72], and is orthogonal to IronBus. Consistent with our threat model (§2.5), we likewise do not address side- or covert-channels between co-located partitions that share on-device resources. Rather, IronBus authenticates and constrains the inter-device flows over which such partitioned accelerators communicate.

Keystore scalability. IronBus requires unique symmetric keys for trusted channels, causing linear growth in the AIU's key storage. To mitigate this, (1) the OS can use secure context switching to swap keys with context when clients exceed AIU endpoints, and (2) OS services can support many clients by reusing a single receive endpoint and a single key. However, leaking this key compromises communication with all clients.

5.2 Formal Verification of HAL Protocols

In a decentralized architecture, security depends entirely on the protocols binding its autonomous units. Because IronBus shifts the Root-of-Trust to the edge, any flaw in the HAL-AIU handshake

would break the system's isolation. We therefore formalize the Trusted Control Channel Creation protocol in symbolic and computational models to prove its resilience against interconnect attacks.
Formalization. Both models assume that: (i) authentic parties follow the protocol (Fig 6), i.e., the HAL and AIU only respond to valid messages; (ii) attackers can read, modify, delete, and drop messages exchanged over the interconnect. *Symbolic model* uses a Dolev-Yao attacker and ignores bit-level details. All functions are assumed to be perfect, e.g., no hash collisions, and known to the attacker. The protocol is modeled by mapping its steps to transition rules on a global state. The notation *Fact(...)@#i* means that *Fact* is created at timestamp *#i*, during the execution of a transition rule. The fact *K(x)* means that an attacker knows *x* and is used to express properties in first-order logic along with: (1) *HALDone(HAL, aiu, key)* and *AIUDone(AIU, HAL, key)*, marking shared key establishment, on the HAL and AIU respectively; (2) *AIUReply(HAL, aiu, keyh)*, indicating the AIU responds to a HAL nonce with key material; (3) *HALTrust(HAL, aiu, keyh)*, indicating the HAL trusts the key material was sent by the AIU. *Computational model* treats messages as bitstrings and functions as bitstring operations. We assume unforgeable signature schemes (UF-CMA), and Diffie-Hellman key exchange under the computational Diffie-Hellman (CDH) assumption, implying negligible forgery and collision probabilities. Each participant is modeled as a process with oracles for protocol steps.Oracle responses follow the protocol, using cryptographic primitives under these assumptions.
Verification. In the symbolic model, we use Tamarin [63] to prove the following two properties:

1. Only the HAL and AIU share the symmetric key.

$$\forall\ HAL\ aiu\ key\ \#i\ \#j\ .\ ((\text{HALDone}(HAL, aiu, key)@\#i \\ \wedge\ \text{AIUDone}(aiu, HAL, key)@\#j) \\ \Longrightarrow\ (\nexists\ \#k\ .\ \text{K}(key)@\#k))$$

2. A symmetric key is established between authenticated partners, ensuring it comes from a trusted party.

$$\forall\ HAL\ aiu\ keyh\ \#i\ .\ (\text{HALTrust}(HAL, aiu, keyh)@\#i \\ \Longrightarrow\ (\exists\ \#k\ .\ \#k < \#i \wedge \text{AIUReply}(aiu, HAL, keyh)@\#k)) \\ \wedge\ \forall\ HAL\ aiu\ key\ \#j\ .\ (\text{AIUDone}(aiu, HAL, key)@\#j \\ \Longrightarrow\ (\exists\ \#k\ .\ \#k < \#j \wedge \text{HALDone}(aiu, HAL, key)@\#k))$$

In the computational model, we use CryptoVerif [17] to prove key secrecy and authenticity of both parties. We prove that an attacker can predict a single key bit with a probability of 0.5. Authenticity is shown injectively: trusting a party's authenticity implies it completed the prior protocol step. Thus, a Dolev-Yao attacker cannot learn critical information like encryption keys but may infer which applications run, which components communicate, and for how long.
Generality of the formal models. Both models formalize the trusted control channel creation between the HAL and an AIU, i.e., mutual attestation and key agreement between a security manager and a device root of trust. They contain nothing specific to M³: the roles map onto the TSM and the DSM of a TDISP platform, so the proofs hold for a deployment that runs this protocol between them.

6 SSLA Programming Framework

To close the "semantic gap" between high-level application logic and low-level hardware security, IronBus integrates a programming framework that automates SSLA policy synthesis. Rather than requiring developers to manually create and configure the secure channels, our framework captures application intent as a *Device Flow Graph (DFG)*. The *compiler* then translates these graphs into hardware-enforced *trusted bus channels*, ensuring that security boundaries are derived directly from the dataflow logic and verified through cryptographic execution proofs.

6.1 DFG-Based Programming Model

IronBus provides an intuitive API that allows developers to implement applications using the DFG-based programming model. To demonstrate its functionality, we present an example implementation of the application depicted in Figure 2. Listing 1a shows a simplified Rust implementation, illustrating the four main steps in the application lifecycle:

```
let gpu = NodeLet::new(
  init: || {},
  task: |filename: &[u8]| {
      let buf      =
      ↪  Service::FS::Load(filename);
      let img_data = resize_image(buf);
      img_data});

let tpu = NodeLet::new(
  init: || {
    Service::FS::Create("/results");
    let model =
    ↪  Service::FS::Load("/weights.pb");},
    task: |img_data: &[u8]| {
    let class_res =
    ↪  image_classification(model);
    Service::FS::Append("/results",
    ↪  class_res);});
```

```
fn main() {
  // @①@ Reserve hardware according to layout policy
  let layout = IronBus::ReserveLayout(
  [Node::new("GPU"), Node::new("TPU")], layout_policy);
  // @②@ Create DFG with one stage (Figure 2a)
  let init_job = layout.newJob("InitJob")
       .newStage([Task{"GPU", &gpu.init}, // InitGPUTask
                  Task{"TPU", &tpu.init}]); // InitTPUTask
  // Create DFG with two stages (Figure 2b)
  let comp_job = layout.newJob("ImgClassJob")
       .newStage([Task{"GPU", &gpu.task}])  // ImgRzTask
       .newStage([Task{"TPU", &tpu.task}]); // ImgClsTask
  // @③@ Run jobs
  let res_init = init_job.run(None, &mut proof_init);
  let res_comp = comp_job.run(img, &mut proof_comp); // @④@
  ↪  proofs
}
```

(a) Image classification code with the DFG model.

```
# HW/SW attributes of DFG nodes
GPU = {"vendor":"NVIDIA", "model":"A100", "ca":0x0, "loc":"DE"}
TPU = {"vendor":"Google", "model":"v3",   "ca":0x0, "loc":"DE"}
FS  = {"version":"latest", "ca":0x1, "name":"m3fs", "loc":"DE"}
# Layout policy
node:-HWVersionIs(GPU) & HWIsExclusive()
node:-HWVersionIs(TPU) & HWIsExclusive()
node:-SWVersionIs(FS)
```

(b) Node policies of the application.

Listing 1. The image classification application of Figure 2: (a) code with the DFG model, (b) node policies.

Step 1. Policy-compliant device allocation. The application requests a specific *device layout* (L17-18). The HAL verifies device availability against the `layout_policy`. If compliant, it instantiates *trusted bus channels*; else, the request is denied. Access is revoked automatically upon completion.
Step 2. DFG job specification. The application defines data processing pipelines as jobs bound to the layout (L20-26). Each job specifies one or more *execution stages*. In our example, `init_job` parallelizes device setup, while `comp_job` sequences GPU preprocessing and TPU inference.
Step 3. Policy-compliant job execution. Upon job invocation (L28-29), the runtime infers the DFG structure and compiles a corresponding dataflow policy. The HAL validates this policy before deploying tasks. Arguments are marshaled into encrypted messages over the reserved channels, and IronBus synchronizes the stages.
Step 4. Proof-of-compliance. Throughout execution, the HAL monitors compliance and generates a verifiable proof-tuple signed by its private key (K_{boot}): $\langle K_{cert}, L, DFG, S, T, SSLA \rangle K_{boot}$

This tuple contains the certificate chain (K_{cert}), device layout (L), DFG topology, task and TCB hashes (S), timestamps (T), and the SSLA. A remote user audits the execution by checking the signature and the tuple against the SSLA.

6.2 Policy Specification and Generation

To minimize manual configuration errors, IronBus employs a hybrid model: users declaratively specify *node policies*, while the compiler automatically synthesizes *edge policies* for the interconnect.
Node policy specification. IronBus uses a declarative language (*node :- condition*) to define hardware/software constraints. Predicates (AND &, OR |) restrict allocation based on manufacturer, model, location, or certificates (Listing 1b, L1–8). For example, the `HWIsExclusive()` predicate reserves accelerators exclusively against co-tenant side channels.
Automated dataflow inference. Instead of manual P2P link configuration, the compiler derives the DFG topology from the application structure: within a job, each `newStage` call implicitly adds

Operation	Key Size (B)	Data Size (B)	Cycles	Attestation count
AES (de)encryption	32	16	15	–
Random number generation	NA	16	500	2
ECDSA signature generation	32	128	762k	5
ECDSA signature verification	64	128	482k	8
Symmetric key generation	32	NA	1000	2

Table 1. Cycles for HSM operations on OpenTitan [74] and their count in the attestation protocol (§5.1).

directed edges between consecutive task sets, from which the compiler generates edge policies, e.g., restricting data flow from a GPU source to a TPU sink.

Policy granularity. Policies differ in the unit they constrain. A *coarse-grained policy* constrains a device as a whole: its vendor, model, firmware, or location, and whether it is used exclusively, as the node policies of Listing 1b do. A *fine-grained policy* constrains individual tasks and flows. For example, only the resize task may run on the GPU and only the classification task on the TPU; images leave the GPU only towards the TPU and results only towards the certified file service; the model must be the attested version; and none of this holds after the job's deadline.

Enforcement and violations. IronBus enforces an SSLA in two phases with two components.

- **Static enforcement by the HAL, at bootstrap:** before the application runs, the HAL enforces the node policies. For the image classification application, it attests the GPU, the TPU, and the file service through their AIUs, checks the attested attributes against the node predicates, and allocates the devices exclusively, aborting on any violation. It then validates the edge policies that the compiler inferred from the DFG and configures the AIU endpoints and channel keys of the devices for exactly these edges; after that it is off the data path.
- **Dynamic enforcement by the AIU, during execution:** every transfer the application issues is checked by the AIUs of the two devices at line rate. A transfer on a configured endpoint is encrypted and integrity-protected with the channel key; a transfer to any other destination is dropped and reported to the HAL as a dataflow policy violation. No software is involved in this phase; the HAL acts again only when the time bound expires, revoking the channels.

When the job completes, the HAL signs the proof that it ran on these devices with these flows.

7 Evaluation

Our evaluation covers the setup (§7.1), security (§7.2), the trusted bus (§7.3), scalability (§7.4), applications (§7.5), the AIU's hardware cost (§7.6), and the policy framework (§7.7).

7.1 Experimental Setup

Platform. We implement and evaluate IronBus on the RISC-V cores in gem5 [16]. The system has 23 tiles running at 2 GHz, one DRAM tile and 22 RISC-V tiles that simulate accelerators. Each tile connects to the interconnect through an AIU with 192 physical endpoints, key slots for HAL configuration, and a 2 KiB internal buffer. To implement the AIU, we extend the DTU in M³'s open-source version of gem5 [14] with AES-GCM (de)encryption, random number generation, and ECDSA signature generation and verification modules. We measure the cycles (see Table 1) for these operations using OpenTitan [74] and incorporate the results into gem5 for a realistic evaluation. The interconnect is a crossbar with 16 B per cycle (32 GB/s) per link. DMA keeps up to 16 packets of 2 KiB in flight, every encrypted packet carries 32 B of IV and MAC, and the AIU has one AES-GCM engine per link direction (16 B per cycle, 15 cycles latency). We vary the interconnect from on-chip to off-chip with 500 cycles (250 ns) latency per packet.

Workload	Description	Tiles	Bus traffic
Microbenchmarks			
IPC	client/server messages, 8 B–1 KiB	2	messages
DMA read/write	2 MiB DRAM buffer, 512 B–4 KiB chunks	2	DMA to memory tile
DMA pipelining	2 MiB tile-to-tile transfer, 1–64 packets in flight	2	DMA between tiles
m3fs read/write	2 MiB file, 4 KiB chunks	3	messages + DMA
Accelerator chain	512 B exchanged across 2–4 stages	2–4+1	messages
Control channel	attestation and key agreement per tile	23	messages
Application interaction with OS services			
imgproc	edge detection: FFT, multiply, iFFT accelerators [11]	3+1	640 kB per stage
facever	GPU face verification of 256 images [97]	1+1	histograms, results to m3fs
imgclass	classifies 256 images on a 16×16 systolic array [103]	1+1	images, results to m3fs
dist	imgclass split over two systolic arrays	2+1	images, results to m3fs
*Collective communication (*1 MiB *per tile)*			
all-reduce	ring over N tiles	2–16	$2(N-1)/N$ MiB per tile
all-gather, reduce-scatter	ring over N tiles	2–16	$(N-1)/N$ MiB per tile
all-to-all	pairwise over N tiles	2–16	$(N-1)/N$ MiB per tile
PP chain	send/receive along N stages	2–16	1 MiB per stage
ML traces (sent per tile, replay window)			
Llama-7B prefill	TP4; TP2·PP2; TP4·PP2 (STAGE, seq 512)	4; 4; 8	16.5; 8; 8.8 MiB
Llama-7B decode	TP4 (STAGE, seq 1)	4	33 KiB
Llama-7B training	TP4; DP4; TP4·PP2; DP8 (STAGE, batch 8)	4; 4; 8; 8	89; 80; 46; 93 MiB
Transformer training	hybrid parallel (ASTRA-sim)	4; 8	58; 204 MiB
DLRM training	all-to-all embeddings, all-reduce MLP (ASTRA-sim)	4; 8	0.3; 0.3 MiB
ResNet-50 training	data parallel: all-reduce gradients (ASTRA-sim)	4; 8	9; 11 MiB

Table 2. Workloads. Tiles: accelerator tiles (+ CPU tile). TP, PP, DP: tensor, pipeline, data parallelism.

Metrics. We report latency in cycles, throughput in GiB/s, completion time in µs, channel-setup time in ms, TCB size in lines of code, and AIU area in FPGA LUTs, flip-flops, and BRAM. Overheads are given relative to M³, and every experiment that runs all three systems also reports absolute times.
Baselines. We compare IronBus against two baselines. *M³* is unprotected direct P2P: no encryption, attestation, or channel keys. *Host-centric* is secure P2P with access control at the root complex, as in TDX-Connect and SEV-IO: every device-to-device transfer is relayed through the host, which decrypts and re-encrypts it. Each device has its own link to the host, i.e., a root port per device, modeled as one relay tile per destination that performs no IOMMU checks. Both favor the host-centric design: behind a PCIe switch, devices would share its uplink and add hops. The comparison thus isolates the cost of the extra host hop with its decryption and re-encryption, not the complete TDX-Connect or SEV-IO path. *IronBus* is secure direct P2P with access control at the AIU. We also compare both against Linux (§7.3).
Workloads. Table 2 lists all workloads, their parameters, and their tiles. The microbenchmarks exercise the primitives of the trusted bus: messages, DMA, and trust bootstrapping. The applications interacting with OS services read and write files in *m3fs*, M³'s in-memory file system, which runs on its own CU and gives clients direct AIU access to a file's extents after a message-based request. The accelerated ones write their results to m3fs; imgproc and imgclass use gem5-aladdin [85] and SMAUG [103] traces. The collectives and ML workloads replay Chakra traces (STAGE [57] for Llama-7B, ASTRA-sim [79] for Transformer, DLRM, and ResNet-50), converted per accelerator into compute, send, and receive operations. Compute times are taken from the trace or from an A100-class roofline and busy-waited. Payloads are dummy, since the AES-GCM cost is data-independent, and accesses to a tile's own memory do not cross the bus and count as compute. Llama-7B traces replay a window of 4 layers, and ASTRA-sim traffic is scaled to the simulated link. The reported time is that of the slowest accelerator, and channel setup is measured separately.

TCB	System	Guest OS	Firmware	Total (vs. IronBus)
Isolation	TDX module	–	45	45 (1.4×)
	SEV firmware	–	29	29 (0.9×)
	IronBus HAL	–	33	33 (1×)
Flow policy	SEV-Linux	2,513	29	2,542 (77×)
	TDX-Linux	3,068	45	3,113 (94×)
	Gramine-TDX	57	45	102 (3×)
	CoCo	1,471	901	2,372 (72×)
	IronBus HAL	0	33	33 (1×)

Table 3. TCB comparison in thousand lines of code; the isolation and flow-policy TCBs are defined in §2.2.

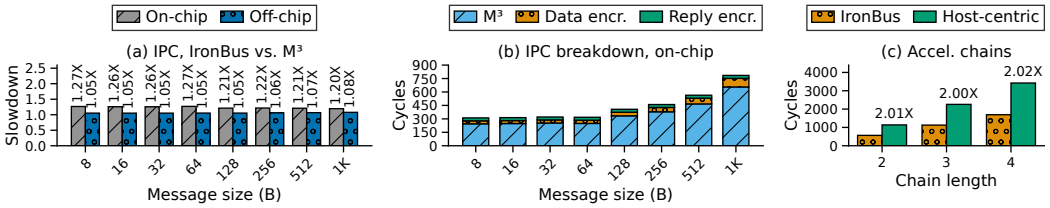


Fig. 7. Inter-tile IPC: (a) slowdown by message size, (b) on-chip cycle breakdown, (c) accelerator chains.

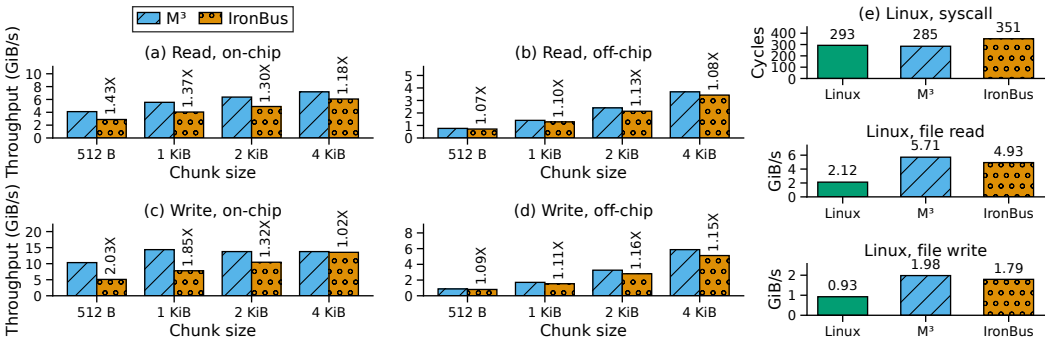


Fig. 8. (a)–(d) CU–DRAM read/write throughput by chunk size. (e) Comparison to Linux.

7.2 Security Analysis

To compare the software that IronBus and the host-centric designs trust, we measure the two TCBs of §2.2: the isolation TCB, which is the TSM firmware (the TDX module [45] or the SEV firmware [2]) in host-centric designs and the HAL in IronBus, and the flow-policy TCB, which is the guest OS plus the TSM firmware in host-centric designs and the HAL alone in IronBus. We count lines of code of a Linux guest as used by SEV-Linux and TDX-Linux, of Gramine-TDX [52], and of CoCo [19]. The hypervisor is outside the TCB of all systems and is not counted.

Results. Table 3 shows both TCBs. For isolation, the HAL (33K lines of code) is of the same size as the TSM firmware (29K–45K) while additionally mapping SSLAs to hardware. For flow policy, host-centric designs must trust 102K (Gramine-TDX) to 3.1M (TDX-Linux) lines of guest OS and firmware, 3–94× more than IronBus. By enforcing data flows at the devices, IronBus removes the guest OS from the flow-policy TCB and with it its largest attack surface.

7.3 Architectural Analysis

In this section, we measure the cost of bootstrapping trust through the HAL and of the AIU's encryption on messages, DMA, host-mediated channels, and the number of engines a link needs.

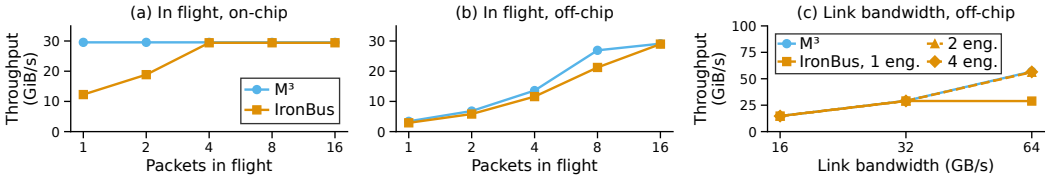


Fig. 9. DMA throughput over packets in flight (a, b) and over link bandwidth and AES-GCM engines (c).

Trusted Control Channel Creation Latency. To quantify the cost of bootstrapping trust (§5.1), we measure the time to create a trusted control channel, from the start of the protocol to the derivation of the shared symmetric key, averaged over all AIUs as seen by the HAL. Trusted control channel creation takes 10 million cycles or 5 ms, with Table 1 giving the operation count. Verification of ECDSA signatures in certificate chains, signed challenges, and nonces dominates these overheads. This one-time cost is paid at bootup.

Inter-tile IPC. To understand the cost of the AIU's encryption on messages, which slows down syscalls and IPC between tasks, a client on one CU sends messages of 8 B to 1 KiB to a server on another CU, which replies. We measure the round trip in cycles on M³ and IronBus on both interconnects and break the on-chip cost of IronBus down into M³'s IPC and the encryption of message and reply. Figure 7(a) shows the IPC slowdown of IronBus over M³ across message sizes and interconnects. On average, secure inter-CU IPC is 23% slower on-chip and 5% off-chip. The overhead is the encryption of the message and the reply (Figure 7(b)), which the longer off-chip link latency largely hides.

Comparison to Host-centric Designs. To understand what direct P2P channels save over host-mediated ones, we run accelerator chains of 2–4 stages exchanging 512 B messages, either directly on IronBus or through a host tile that decrypts, checks, and re-encrypts every message, as in the host-centric design. As shown in Figure 7(c), host-centric designs are 2× slower on average. Host-centric setups require 2N–2 send/receive operations for N stages, while IronBus needs only N–1, reducing copies and encryptions.

Inter-tile I/O. To measure the impact of line-rate encryption on DMA, an application on one CU reads and writes a 2 MiB DRAM buffer over the interconnect in chunks of 512 B to 4 KiB. We compare the throughput of IronBus and M³ on both interconnects. Figure 8(a)–(d) shows the read and write throughput by chunk size. Chunks up to 2 KiB are single packets, so each command exposes the packet's store-and-forward encryption latency of 100–200 cycles. At 4 KiB, two packets are pipelined and the read overhead drops to 18% on-chip. On-chip writes are bounded by the DRAM channel from 2 KiB on (about 14 GiB/s, on M³ as well), which hides the encryption latency at 4 KiB (2%). Off-chip, every command waits for the link round trip, so the same latency costs 8% for reads and 15% for writes at 4 KiB.

Comparison to Linux. To put IronBus's results into perspective, we compare it against M³ and Linux on the same simulated hardware: a no-op system call and a 2 MiB file read and write in 4 KiB chunks on Linux v5.10 (tmpfs), M³ (m3fs), and IronBus. We compare OS primitives rather than the accelerator workloads because Linux's peer-to-peer DMA is orchestrated by the host [91], i.e., the host-centric design we model directly. Figure 8(e) shows the results: a no-op system call costs 293 cycles on Linux, 285 cycles on M³, and 351 cycles on IronBus, and M³ reads and writes the file 2.7× and 2.1× faster than Linux, IronBus 2.3× and 1.9×. IronBus adds 66 cycles to M³'s system call for encrypting the request and the reply. File I/O is faster on M³ and IronBus because Linux enters the kernel on every 4 KiB read and write and copies the data through the page cache, whereas an M³ application requests a file's extents once and then reads and writes them directly through the DTU, which IronBus keeps and encrypts.

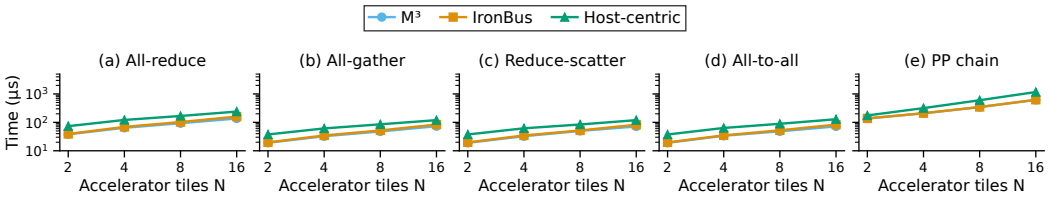


Fig. 10. Completion time of the collectives (1 MiB per tile, off-chip) over the number of accelerator tiles.

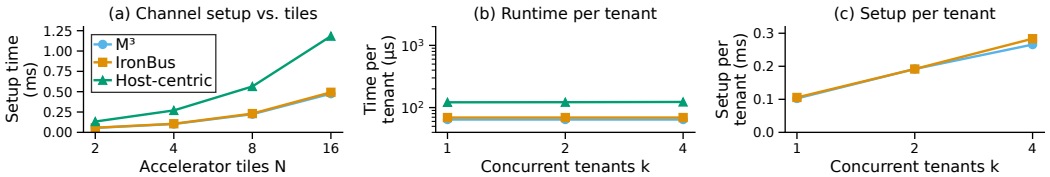


Fig. 11. All-reduce, off-chip: (a) channel setup over tiles, (b) runtime and (c) channel setup per tenant.

DMA Pipelining. To understand how DMA hides the encryption latency that the single-packet commands above expose, two accelerators exchange 2 MiB tile to tile in 1 MiB DMA commands while we vary the packets in flight per command from 1 to 16 on both interconnects. Figure 9(a) and (b) show the throughput. With one packet in flight, every 2 KiB packet exposes its encryption latency and IronBus reaches 41% of M³'s throughput on-chip. With 4 (on-chip) or 16 (off-chip) packets in flight, the latency is hidden and IronBus runs at the link rate, 0.5% below M³ (IV and MAC).
Crypto Engine Scaling. To determine how many AES-GCM engines a link needs, we vary, with 64 packets in flight off-chip, the link from 16 to 64 GB/s and the engines per direction from 1 to 4 (Figure 9(c)). An engine processes one 16 B block per cycle, 32 GB/s at 2 GHz: on the 32 GB/s link one engine per direction suffices, on a 64 GB/s link one halves the throughput and two restore it, so the AIU needs ⌈link bandwidth/32 GB/s⌉ engines per direction.

7.4 Scalability Analysis

Next, we examine how IronBus scales with more accelerators and more tenants sharing a HAL.
Collective Communication. To understand how the overhead grows with the number of accelerators, we replay the five collective patterns of Table 2 with 1 MiB per tile over 2–16 accelerator tiles on M³, IronBus, and the host-centric design, on the off-chip interconnect. Figure 10 shows the completion times. IronBus's overhead over M³ grows from 3% at N=2 to 13–15% at N=16 because the per-step transfer shrinks with N (64 KiB at N=16): every transfer pays a fixed encryption latency before its packets stream at line rate, which weighs more on a smaller transfer. The four collectives coincide because their per-step chunk is the same, 1 MiB/N. The host-centric design is 1.9–2.0× slower than M³ at N=2 and 1.6–1.8× at N=16: the host stores and forwards every transfer for access control, which doubles its transfer time. In the pipeline chain, IronBus stays within 2% and host-centric grows from 1.3× to 1.9×, since every stage adds a host hop.
Multi-tenancy. To understand what the single HAL costs when accelerators and tenants are added, we measure channel setup and multi-tenant runtime. Channel setup is the time the HAL needs to create the channels of an all-reduce over 2–16 tiles, i.e., to create and delegate the memory and message capabilities, to activate the endpoints, and, on IronBus, to configure the keys. Attestation happens once at boot (§7.3) and is not included. For multi-tenancy, 1–4 tenants each run an all-reduce on 4 tiles under one HAL; on the host-centric design all tenants share the host, and every device has its own link to it. We report the runtime and the setup time per tenant.

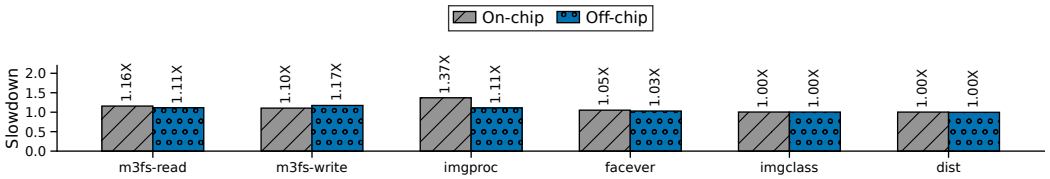


Fig. 12. Slowdown of IronBus over M³ for the applications interacting with OS services (Table 2).

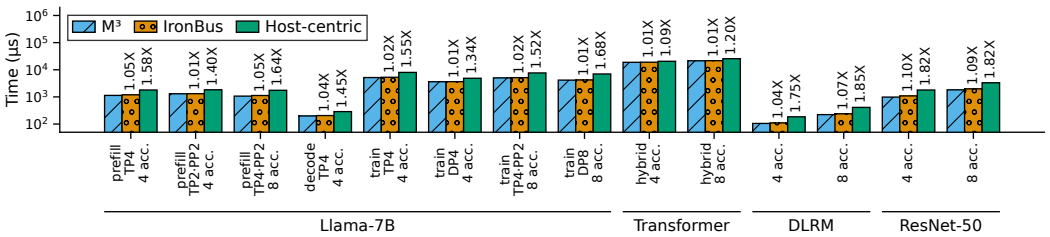


Fig. 13. Completion time of the ML traces (off-chip), annotated with the slowdown over M³.

Results. Figure 11(a) shows the channel setup time. Channels follow the edges of the DFG: a ring collective needs N channels, all-to-all $N(N-1)$, and the host-centric design $2N$. Setup is therefore linear in N for the ring collectives (0.06 ms at N=2 to 0.49 ms at N=16 for IronBus, below host-centric's 0.13 ms to 1.18 ms) and quadratic for all-to-all (5.1 ms at N=16), about 20 µs per channel. These are capability operations that M³ performs identically, and IronBus adds only the key configuration over the control channel, 2–4%. Figures 11(b) and (c) show the tenants. Runtime per tenant is constant for M³ (64 µs) and IronBus (69 µs), because tenants share nothing on the data path. The host-centric runtime per tenant is constant as well, but at 122 µs. The HAL is a control-plane component that is only involved in setup, whereas the host-centric design keeps the host on the data path of every transfer. What sharing the HAL costs is setup: the HAL serializes the tenants' channel creation, 0.11 ms to 0.28 ms per tenant.

Key storage. Each endpoint holds 352 bits of key material (a 256-bit AES key and a 96-bit IV counter), plus one 256-bit control key per AIU: 8.3 KiB for 192 endpoints, linear in the number of endpoints. An AIU needs one endpoint per DFG edge it terminates, summed over the tenants using the device: 2 per tenant for a ring collective, $2(N-1)$ for all-to-all, 6 for the Llama TP4 layout, so 192 endpoints hold 96 ring tenants, 32 TP4 tenants, or 6 all-to-all tenants over 16 tiles. Beyond that, the HAL swaps keys (§5). A Selective IDE port doing direct P2P has the same growth (12 keys per stream and peer pair); the host-centric design avoids it only by having no direct channels.

7.5 Application Workloads

Next, we detail the overhead of secure OS service access and application execution on IronBus compared to M³, and how IronBus and the host-centric design fare on modern ML workloads.

Application Interaction with OS Services. To understand how the overheads of the primitives combine when applications use OS services, *m3fs-read* and *m3fs-write* read and write a 2 MiB file in 4 KiB chunks through the m3fs service, and *imgproc*, *facever*, *imgclass*, and *dist* run as described in Table 2. Figure 12 shows the slowdown of IronBus over M³ for these workloads on both interconnects, and §7.3 gives M³'s absolute performance. m3fs reads slow down by 16% on-chip and 11% off-chip (5.7 GiB/s to 4.9 GiB/s and 3.2 GiB/s to 2.9 GiB/s) and writes by 10% and 17% (2.0 GiB/s to 1.8 GiB/s and 0.86 GiB/s to 0.73 GiB/s). The slowdown stems from encrypting file data and metadata on the interconnect. *imgproc* has the highest overheads (11% off-chip, 37% on-chip) due to its large image transfers. In contrast, *facever*, *imgclass*, and *dist* are compute-intensive with fewer data transfers.

Group	Synthesized block	LUTs	FFs	BRAM
Common to DSM and AIU	Pipelined AES, Tx and Rx (IDE engines)	81,732	28,172	0
	Key store, 352 bits × 192 (IDE keystore)	0	256	5
	OpenTitan AES and HMAC (SPDM session)	21,876	7,764	0
	OpenTitan OTBN, ECDSA (SPDM signer)	86,083	21,567	16.5
	OpenTitan RNG	24,300	14,900	0
AIU only	Capability store, 256 bits × 192	0	0	4
	Permission check per command	135	2	0
Total (AIU-only share)		214,126 (0.1%)	72,661 (0.003%)	25.5 (16%)

Table 4. FPGA cost of AIU hardware, DSM terms in brackets (UltraScale+, out-of-context, 36 kbit BRAMs).

ML Workloads. To understand how IronBus performs on current accelerator workloads, we replay the 14 ML traces of Table 2, 4 inference and 10 training runs of Llama-7B, Transformer, DLRM, and ResNet-50 at several parallel layouts on 4 and 8 accelerators, on M³, IronBus, and the host-centric design on the off-chip interconnect.
Results. Figure 13 shows the completion times. IronBus stays within 0–6% of M³ on the Llama-7B and Transformer workloads because their tensor exchanges are large and pipelined, so the AIU encrypts them at line rate (§7.3). DLRM and ResNet-50 exchange embeddings of 2.5 KiB and gradient chunks of 11 KiB, many small transfers that each expose the per-packet store-and-forward latency of §7.3, and cost 4–10%. The host-centric design is 1.1–1.9× slower than M³ because each device-to-device transfer takes two hops through the host, so the cost grows with the communication share. The compute-dominated Transformer exposes the host least (1.1–1.2×) and DLRM and ResNet-50 on 8 accelerators the most (1.9× and 1.8×), since each of their many small transfers pays the hop. The collectives of §7.4 bound this overhead; a workload's compute-to-communication ratio and transfer sizes (Table 2) decide how much shows.

7.6 AIU Hardware Cost

To understand the AIU's area overheads, we implement it in RTL and synthesize it out of context with Vivado, separating the blocks a TDISP DSM also needs from those only the AIU adds (Table 4).
Results. Over a DSM, the AIU adds only the capability store and the per-command check: 0.1% of LUTs for the security hardware's logic and 16% (6 KiB) of BRAM for capability tokens for 192 endpoints, an upper bound since the table omits the PCIe logic both need. These blocks draw 0.8% of its dynamic power (Vivado, 250 MHz, 12.5% toggle rate). Both stores grow linearly with the endpoints; the key store holds 8.3 KiB for 192. Each extra engine per direction adds 19% (40.9k LUTs, 14.1k FFs) and is needed only above 32 GB/s per engine (§7.3).

7.7 SSLA Programming Framework

We next evaluate the policy framework's compliance support and its overheads.
Policy Expressiveness. We demonstrate the framework's expressiveness through case studies addressing data residency [26], pipeline integrity [32, 33], and secure processing [34, 35, 87].
Execution integrity. To ensure only authorized GPU kernels access sensitive data, the node policy uses the SWVersionIs predicate. The HAL verifies the kernel's hash and certificate before execution and includes this in the compliance proof: `node :- & SWVersionIs(GPU_KERNEL)`.
Data security. The runtime synthesizes edge policies that enforce encryption and integrity checks.
Temporal data deletion. To purge data across distributed components, users can specify an upper time bound T on DFG dataflows, and the runtime generates edge policies that deny access after T.
Policy Enforcement Overheads. We measure the latency to compile manually specified node policies and synthesize access control rules [32] for various pipelines.

Results. Overhead correlates with node complexity: compiling the baseline policies takes only 0.11 ms for the software-only *m3fs-read/write* flows due to their small node count, 5.2 ms for *facever*, 10.5 ms for *imgclass*, and 15.7 ms for the hardware-heavy *imgproc* and *dist* pipelines. Most of this is the attestation and control channel setup of each hardware tile, which arises only at job setup, so the HAL can hide it by bootstrapping asynchronously and caching attestation results.

8 Deployment Considerations

While IronBus is developed on M³, its design could be deployed on platforms with TDISP-capable PCIe devices [5, 44]. We outline this deployment path for each of its three layers, what would carry over and what would require engineering.

Hardware layer. The AIU's functions could be integrated into a device's PCIe controller: its memory endpoint would correspond to an ATS cache with capabilities, and its AES-GCM engine to the IDE engine [76]. Table 4 shows that a TDISP/IDE port already contains the cryptography, key storage, and root of trust; the addition is the endpoint table with its permission check. This would require two changes: DSM-configurable registers that associate IDE streams with the AIU's endpoints, and capability-carrying encrypted packets generated and verified at the IDE ports.

Software layer. This would require changes to the TSM and DSM software. The TSM would take the HAL's role: it would manage IronBus's keys and capability tokens and perform capability-based resource management for IDE streams. The DSM would configure the device-side IDE port with its keys and capability tokens. Software in the host's trusted domain could remain unchanged; data that enters a CVM would then remain under its guest OS, and IronBus would remove the guest OS only from the flows between devices. Existing device libraries such as CUDA, RoCE, or a network stack would bind peers and buffers once, at setup, through the policy compiler described next, then let the devices move data; the AIU endpoints would confine them to the declared flows.

Policy compiler. The policy compiler does not depend on the platform. On PCIe platforms it could be a thin shim in a library's transport layer: it would turn each binding of peers and buffers at setup into a channel request that the TSM validates against the DFG and maps onto IDE streams whose permissions the endpoints restrict. In distributed ML, for example, the communication graph that a collective library plans is the DFG, so the edge policies could be inferred from it without changing the framework. The same holds for other dataflow programming frameworks [38, 75, 88, 107].

9 Related Work

IronBus relates to accelerator TEEs, device I/O protection, and dataflow policy enforcement.

Accelerator TEEs. Beyond CPU TEEs [4, 9, 10, 43, 46, 81, 90], TEEs exist for GPUs (Graviton [100], Telekine [41], HIX [48]), FPGAs [18, 23, 42, 73, 104, 106, 108], ASICs [20, 96], and NICs [36]. Each fuses trusted logic into one microarchitecture, so each device class needs its own TEE. TDX-Connect [44] and SEV-IO [5], built on TDISP [76], extend the CPU TEE to devices but remain host-centric (§2.1). IronBus's device-agnostic AIU instead enforces policy at the endpoints.

Device I/O protection. IOMMUs [3, 8, 56, 59, 60, 77] and IOPMPs [28, 80] guard against malicious DMA, and other designs add access control to the I/O bus [13, 27, 48, 71, 101]; being host-rooted, they do not defend the interconnect. HETEE [109] does, isolating an accelerator pool behind a centralized secure controller. Others secure memory traffic for generic [40] or specific accelerators [1, 54, 70, 86], and CRONUS [49] reuses TEE hardware for spatially shared secure I/O. IronBus instead enforces authenticated access control at device endpoints and binds every transfer to a declared SSLA.

Dataflow programming and policy enforcement. Accelerator frameworks [38, 68, 69, 75, 88, 107] express applications as dataflow graphs, a style IronBus's DFG API mirrors, but offer only piecemeal security guidance [6, 89] and no way to specify or enforce an SSLA. Policy compliance has been studied for file systems [31, 95] and databases [24, 51, 62, 93], but not for accelerators and

their interconnect. IronBus's compiler infers data-flow restrictions from the DFG, lowers them to hardware-enforced channels, and signs execution proofs.

10 Conclusion

We present IronBus, a hardware-software co-design that replaces host-centric bottlenecks with cryptographically anchored, hardware-enforced trusted bus channels at the interconnect edge. Its contributions are: (1) *the AIU*, a device-agnostic silicon RoT for hardware-secure data movement; (2) *the HAL*, a lean trusted software layer with formally verified protocols; and (3) *the SSLA programming framework*, a DFG-based model that generates audit-ready proofs of policy compliance. Our evaluation shows that IronBus runs accelerator chains, collective communication, and ML workloads 1.1–2× faster than host-centric designs with a HAL of the size of their firmware, providing a verifiable substrate for regulated cloud environments.

Artifact evaluation and availability. The IronBus hardware-software implementation, along with formal verification proofs, is available at https://github.com/TUM-DSE/ironbus-artifact.

Acknowledgments

We thank Nils Asmussen, Carsten Weinhold, and Michael Roitzsch for feedback on an early draft. This work was supported in part by an ERC Starting Grant (ID: 101077577); the Chips Joint Undertaking (JU), European Union (EU) HORIZON-JU-IA, under grant agreement No. 101140087 (SMARTY); the DFG Priority Program SPP-2378 "Resilient Worlds"; the Intel Trustworthy Data Center of the Future (TDCoF); Google Research Grants; JST CREST, Japan, Grant Number JPMJCR22M3; IAPMEI under grant C6632206063-00466847 (SmartRetail); and Fundação para a Ciência e a Tecnologia I.P. (FCT) under project UID/50021/2025 (DOI: https://doi.org/10.54499/UID/50021/2025).

References

[1] Rahaf Abdullah, Huiyang Zhou, and Amro Awad. 2023. Plutus: Bandwidth-efficient memory security for GPUs. In *HPCA*.
[2] AMD. 2026. AMD-ASPFW. link. Accessed: September 2026.
[3] AMD. 2026. AMD I/O Virtualization Technology (IOMMU) Specification. link. Accessed: September 2026.
[4] AMD. 2026. AMD Secure Encrypted Virtualization (SEV). link. Accessed: September 2026.
[5] AMD. 2026. AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization. link. Accessed: September 2026.
[6] Apache Spark. 2026. Spark security. link. Accessed: September 2026.
[7] Arm. 2026. Arm Realm Management Extension (RME) System Architecture. link. Accessed: September 2026.
[8] Arm. 2026. Arm System Memory Management Unit Architecture Specification, SMMU Architecture Version 3. link. Accessed: September 2026.
[9] Arm. 2026. Building a secure system using TrustZone technology. link. Accessed: September 2026.
[10] Nils Asmussen, Sebastian Haas, Carsten Weinhold, Nicholas Gordon, Stephan Gerhold, Friedrich Pauls, Nilanjana Das, and Michael Roitzsch. 2026. TEEM^3: Core-independent and cooperating trusted execution environments. In *ASPLOS*.
[11] Nils Asmussen, Michael Roitzsch, and Hermann Härtig. 2019. M^{3}x: Autonomous accelerators via context-enabled fast-path communication. In *USENIX ATC*.
[12] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *ASPLOS*.
[13] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2021. CURE: A security architecture with CUstomizable and resilient enclaves. In *USENIX Security*.
[14] Barkhausen Institut. 2026. M^3: gem5 simulator with TCU support. link. Accessed: September 2026.
[15] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding applications from an untrusted cloud with Haven. In *OSDI*.
[16] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* (2011).
[17] Bruno Blanchet. 2005. A computationally sound automatic prover for cryptographic protocols. In *Workshop on the Link between Formal and Computational Models*.

[18] Jiyang Chen, Anubhav Panda, Harshavardhan Unnibhavi, Atsushi Koshiba, and Pramod Bhatotia. 2026. μShell: A microkernel-based FPGA shell architecture. In *OSDI*.
[19] Confidential Containers. 2026. Confidential Containers. link. Accessed: September 2026.
[20] Aritra Dhar, Clément Thorens, Lara Magdalena Lazier, and Lukas Cavigelli. 2025. Guardain: Protecting emerging generative AI workloads on heterogeneous NPU. In *IEEE S&P*.
[21] DMTF. 2025. Security Protocol and Data Model (SPDM) Specification. link. Accessed: April 2025.
[22] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. 2011. What if you can't trust your network card?. In *RAID*.
[23] Ken Eguro and Ramarathnam Venkatesan. 2012. FPGAs for trusted cloud computing. In *FPL*.
[24] Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel. 2016. Thoth: Comprehensive policy compliance in data retrieval systems. In *USENIX Security*.
[25] Andre Esteva, Katherine Chou, Serena Yeung, Nikhil Naik, Ali Madani, Ali Mottaghi, Yun Liu, Eric Topol, Jeff Dean, and Richard Socher. 2021. Deep learning-enabled medical computer vision. *npj Digital Medicine* (2021).
[26] European Parliament and Council of the European Union. 2025. Regulation (EU) 2016/679 (General Data Protection Regulation). link. Accessed: February 2025.
[27] Erhu Feng, Dahu Feng, Dong Du, Yubin Xia, and Haibo Chen. 2024. sNPU: Trusted execution environments on integrated NPUs. In *ISCA*.
[28] Erhu Feng, Dahu Feng, Dong Du, Yubin Xia, Wenbin Zheng, Siqi Zhao, and Haibo Chen. 2024. sIOPMP: Scalable and efficient I/O protection for TEEs. In *ASPLOS*.
[29] Forbes. 2023. What large models cost you – there is no free AI lunch. link. Accessed: July 2024.
[30] Future of Life Institute. 2026. The EU Artificial Intelligence Act. link. Accessed: June 2026.
[31] Deepak Garg and Frank Pfenning. 2010. A proof-carrying file system. In *IEEE S&P*.
[32] GDPR-Info.eu. 2025. Art. 32 GDPR security of processing. link. Accessed: February 2025.
[33] GDPR-Info.eu. 2025. Recital 64 identity verification. link. Accessed: February 2025.
[34] GDPR.eu. 2025. Art. 5 GDPR principles relating to processing of personal data. link. Accessed: February 2025.
[35] GDPR.eu. 2026. Complete guide to GDPR compliance. link. Accessed: June 2026.
[36] Dimitra Giantsidi, Julian Pritzi, Felix Gust, Antonios Katsarakis, Atsushi Koshiba, and Pramod Bhatotia. 2025. TNIC: A trusted NIC architecture: A hardware-network substrate for building high-performance trustworthy distributed systems. In *ASPLOS*.
[37] Philipp Giersfeld, Benedict Schlüter, and Shweta Shinde. 2026. BreakFAST: Confused deputy attack on Infinity Fabric to break AMD SEV-SNP. In *IEEE S&P*.
[38] Google. 2026. TensorFlow. link. Accessed: November 2026.
[39] Google Cloud. 2026. Fuzzing PCI Express: Security in plaintext. link. Accessed: September 2026.
[40] Weizhe Hua, Muhammad Umar, Zhiru Zhang, and G. Edward Suh. 2022. MGX: Near-zero overhead memory protection for data-intensive accelerators. In *ISCA*.
[41] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. 2020. Telekine: Secure computing with cloud GPUs. In *NSDI*.
[42] Dongil Hwang, Sanzhar Yeleuov, Jiwon Seo, Minu Chung, Hyungon Moon, and Yunheung Paek. 2023. Ambassy: A runtime framework to delegate trusted applications in an ARM/FPGA hybrid system. *IEEE Transactions on Mobile Computing* (2023).
[43] Intel. 2026. Intel Software Guard Extensions (Intel SGX). link. Accessed: September 2026.
[44] Intel. 2026. Intel TDX Connect TEE-IO Device Guide. link. Accessed: September 2026.
[45] Intel. 2026. Intel TDX Module. link. Accessed: September 2026.
[46] Intel. 2026. Intel Trust Domain Extensions (Intel TDX). link. Accessed: September 2026.
[47] Intel. 2026. Intel Virtualization Technology for Directed I/O. link. Accessed: September 2026.
[48] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous isolated execution for commodity GPUs. In *ASPLOS*.
[49] Jianyu Jiang, Ji Qi, Tianxiang Shen, Xusheng Chen, Shixiong Zhao, Sen Wang, Li Chen, Gong Zhang, Xiapu Luo, and Heming Cui. 2022. CRONUS: Fault-isolated, secure and high-performance heterogeneous computing for trusted execution environment. In *MICRO*.
[50] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: Exploiting speculative execution. In *IEEE S&P*.
[51] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. 2018. Pesos: Policy enhanced secure object store. In *EuroSys*.
[52] Dmitrii Kuvaiskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij. 2024. Gramine-TDX: A lightweight OS kernel for confidential VMs. In *CCS*.

[53] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. 2020. An off-chip attack on hardware enclaves via the memory bus. In *USENIX Security*.
[54] Sunho Lee, Jungwoo Kim, Seonjin Na, Jongse Park, and Jaehyuk Huh. 2022. TNPU: Supporting trusted execution with tree-less integrity protection for neural processing unit. In *HPCA*.
[55] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading kernel memory from user space. In *USENIX Security*.
[56] Moshe Malka, Nadav Amit, and Dan Tsafrir. 2015. Efficient intra-operating system protection against harmful DMAs. In *FAST*.
[57] Changhai Man, Joongun Park, Hanjiang Wu, Huan Xu, Srinivas Sridharan, and Tushar Krishna. 2026. Scalable synthesis of distributed LLM workloads through symbolic tensor graphs. In *ISCA*.
[58] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. 2019. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *NDSS*.
[59] Alex Markuze, Adam Morrison, and Dan Tsafrir. 2016. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *ASPLOS*.
[60] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. 2018. DAMN: Overhead-free IOMMU protection for networking. In *ASPLOS*.
[61] Alex Markuze, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. 2021. Characterizing, exploiting, and detecting DMA code injection vulnerabilities in the presence of an IOMMU. In *EuroSys*.
[62] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. 2017. Qapla: Policy compliance for database-backed systems. In *USENIX Security*.
[63] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN prover for the symbolic analysis of security protocols. In *CAV*.
[64] Microsoft. 2024. Introducing GPT-4 in Azure OpenAI service. link. Accessed: July 2024.
[65] Nimish Mishra, Anirban Chakraborty, and Debdeep Mukhopadhyay. 2024. Faults in our bus: Novel bus fault attack to break ARM TrustZone. In *NDSS*.
[66] Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. 2024. Confidential VMs explained: An empirical analysis of AMD SEV-SNP and Intel TDX. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2024).
[67] MITRE. 2026. CAPEC-97: Cryptanalysis. link. Accessed: September 2026.
[68] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A distributed framework for emerging AI applications. In *OSDI*.
[69] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. 2021. tf.data: A machine learning data processing framework. *Proceedings of the VLDB Endowment* (2021).
[70] Seonjin Na, Sunho Lee, Yeonjae Kim, Jongse Park, and Jaehyuk Huh. 2021. Common counters: Compressed encryption counters for secure GPU memory. In *HPCA*.
[71] Pascal Nasahl, Robert Schilling, Mario Werner, and Stefan Mangard. 2021. HECTOR-V: A heterogeneous CPU architecture for a secure RISC-V execution environment. In *AsiaCCS*.
[72] NVIDIA. 2026. NVIDIA Multi-Instance GPU. link. Accessed: September 2026.
[73] Hyunyoung Oh, Kevin Nam, Seongil Jeon, Yeongpil Cho, and Yunheung Paek. 2021. MeetGo: A trusted execution environment for remote applications on FPGA. *IEEE Access* (2021).
[74] OpenTitan. 2026. OpenTitan: Open source silicon root of trust. link. Accessed: September 2026.
[75] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*.
[76] PCI-SIG. 2026. PCI Express 6.0 Specification. link. Accessed: September 2026.
[77] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. 2015. Utilizing the IOMMU scalably. In *USENIX ATC*.
[78] Bharat Srinivas Pillilli, Vishal Soni, and Bryan David Kelly. 2024. Securing ATS from rogue devices for confidential computing. link. Accessed: September 2026.
[79] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2020. ASTRA-SIM: Enabling SW/HW co-design exploration for distributed DL training platforms. In *ISPASS*.
[80] RISC-V International. 2026. RISC-V IOPMP Architecture Specification. link. Accessed: September 2026.
[81] Patrick Sabanic, Masanori Misono, Teofil Bodea, Julian Pritzi, Michael Hackl, Dimitrios Stavrakakis, and Pramod Bhatotia. 2026. Wallet: Confidential serverless computing. In *NSDI*.

[82] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. 2021. Opening Pandora's box: A systematic study of new ways microarchitecture can leak private data. In *ISCA*.
[83] Benedict Schlüter and Shweta Shinde. 2026. Staleus: Breaking AMD SEV-SNP via memory incoherence. In *USENIX Security*.
[84] Benedict Schlüter, Christoph Wech, and Shweta Shinde. 2026. Fabricked: Misconfiguring Infinity Fabric to break AMD SEV-SNP. In *USENIX Security*.
[85] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. 2016. Co-designing accelerators and SoC interfaces using gem5-Aladdin. In *MICRO*.
[86] Nivedita Shrivastava and Smruti Ranjan Sarangi. 2023. Securator: A fast and secure neural processing unit. In *HPCA*.
[87] State of California Department of Justice. 2026. California Consumer Privacy Act (CCPA). link. Accessed: June 2026.
[88] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering* (2010).
[89] TensorFlow. 2026. Using TensorFlow securely. link. Accessed: September 2026.
[90] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. 2021. rkt-io: A direct I/O stack for shielded execution. In *EuroSys*.
[91] The Linux Kernel documentation. 2026. PCI peer-to-peer DMA support. link. Accessed: September 2026.
[92] Yatish Turakhia, Gill Bejerano, and William J. Dally. 2018. Darwin: A genomics co-processor provides up to 15,000X acceleration on long read assembly. In *ASPLOS*.
[93] Harshavardhan Unnibhavi, David Cerdeira, Antonio Barbalace, Nuno Santos, and Pramod Bhatotia. 2022. Secure and policy-compliant query processing on heterogeneous computational storage architectures. In *SIGMOD*.
[94] U.S. Department of Health and Human Services. 2024. Health information privacy. link. Accessed: October 2024.
[95] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Aastha Mehta, Deepak Garg, Peter Druschel, Rodrigo Rodrigues, Johannes Gehrke, and Ansley Post. 2015. Guardat: Enforcing data policies at the storage layer. In *EuroSys*.
[96] Kapil Vaswani, Stavros Volos, Cédric Fournet, Antonio Nino Diaz, Ken Gordon, Balaji Vembu, Sam Webster, David Chisnall, Saurabh Kulkarni, Graham Cunningham, Richard Osborne, and Daniel Wilkinson. 2023. Confidential computing within an AI accelerator. In *USENIX ATC*.
[97] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. 2022. Slashing the disaggregation tax in heterogeneous data centers with FractOS. In *EuroSys*.
[98] Rahul Vishwakarma and Amin Rezaei. 2023. Risk-aware and explainable framework for ensuring guaranteed coverage in evolving hardware trojan detection. In *ICCAD*.
[99] Rahul Vishwakarma and Amin Rezaei. 2024. Uncertainty-aware hardware trojan detection using multimodal deep learning. In *DATE*.
[100] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted execution environments on GPUs. In *OSDI*.
[101] Chenxu Wang, Danqing Tang, Changxu Ci, Junjie Huang, Yankai Xu, Fengwei Zhang, Jiannong Cao, Jie Song, Shoumeng Yan, Tao Wei, and Zhengyu He. 2025. ccAI: A compatible and confidential system for AI computing. In *MICRO*.
[102] Lisa Wu, David Bruns-Smith, Frank A. Nothaft, Qijing Huang, Sagar Karandikar, Johnny Le, Andrew Lin, Howard Mao, Brendan Sweeney, Krste Asanović, David A. Patterson, and Anthony D. Joseph. 2019. FPGA accelerated INDEL realignment in the cloud. In *HPCA*.
[103] Sam Likun Xi, Yuan Yao, Kshitij Bhardwaj, Paul Whatmough, Gu-Yeon Wei, and David Brooks. 2020. SMAUG: End-to-end full-stack simulation infrastructure for deep learning workloads. *ACM Transactions on Architecture and Code Optimization* (2020).
[104] Ke Xia, Yukui Luo, Xiaolin Xu, and Sheng Wei. 2021. SGX-FPGA: Trusted execution environment for CPU-FPGA heterogeneous architecture. In *DAC*.
[105] Xilinx. 2022. F1 acceleration for Montecarlo: Financial algorithms on FPGA. link. Accessed: November 2022.
[106] Lei Xu, Weidong Shi, and Taeweon Suh. 2014. PFC: Privacy preserving FPGA cloud - A case study of MapReduce. In *IEEE CLOUD*.
[107] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*.
[108] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. 2022. ShEF: Shielded enclaves for cloud FPGAs. In *ASPLOS*.
[109] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, and Dan Meng. 2020. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *IEEE S&P*.