Confidential Computing within an AI Accelerator

Kapil Vaswani, Stavros Volos, Cédric Fournet, Antonio Nino Diaz, and Ken Gordon, *Microsoft;* Balaji Vembu, *Meta;* Sam Webster and David Chisnall, *Microsoft;* Saurabh Kulkarni, *Lucata Systems;* Graham Cunningham, *XTX Markets;* Richard Osborne, *Graphcore;* Daniel Wilkinson, *Imagination Technologies*

https://www.usenix.org/conference/atc23/presentation/vaswani

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Confidential Computing within an AI Accelerator

Kapil Vaswani[1], Stavros Volos[1], Cédric Fournet[1]
Antonio Nino Diaz[1], Ken Gordon[1], Balaji Vembu[3,†], Sam Webster[1], David Chisnall[1], Saurabh Kulkarni[4,†]
Graham Cunningham[5,‡], Richard Osborne[2], Daniel Wilkinson[6,‡]

[1]*Microsoft* [2]*Graphcore* [3]*Meta* [4]*Lucata Systems* [5]*XTX Markets* [6]*Imagination Technologies*

Abstract

We present IPU Trusted Extensions (ITX), a set of hardware extensions that enables trusted execution environments in Graphcore's AI accelerators. ITX enables the execution of AI workloads with strong confidentiality and integrity guarantees at low performance overheads. ITX isolates workloads from untrusted hosts, and ensures their data and models remain encrypted at all times except within the accelerator's chip. ITX includes a hardware root-of-trust that provides attestation capabilities and orchestrates trusted execution, and on-chip programmable cryptographic engines for authenticated encryption of code/data at PCIe bandwidth.

We also present software for ITX in the form of compiler and runtime extensions that support multi-party training without requiring a CPU-based TEE.

We included experimental support for ITX in Graphcore's GC200 IPU taped out at TSMC's 7nm node. Its evaluation on a development board using standard DNN training workloads suggests that ITX adds $< 5\%$ performance overhead and delivers up to 17x better performance compared to CPU-based confidential computing systems based on AMD SEV-SNP.

1 Introduction

Machine learning (ML) is transforming many tasks such as medical diagnostics, video analytics, and financial forecasting. Their progress is largely driven by the computational capabilities and large memory bandwidth of AI accelerators such as NVIDIA GPUs, Alibaba's NPU [2], Google's TPU [18], and Amazon's Inferentia [3]. Their security and privacy is a serious concern: due to the nature and volume of data required to train sophisticated models, the sharing of accelerators in public clouds to reduce cost, and the increasing frequency and severity of data breaches, there is a realization that machine learning systems require stronger end-to-end protection mechanisms for their sensitive models and data.

[†]Work done while at Microsoft; [‡]Work done while at Graphcore.

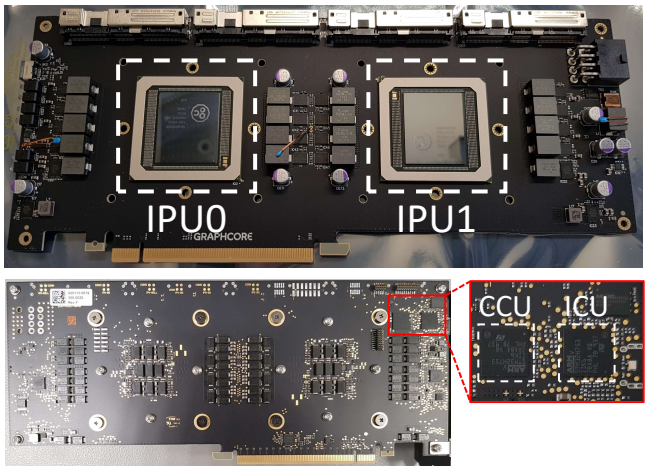

Figure 1: Graphcore Intelligence Processing Unit (IPU) development board (May 2020) with ITX extensions, showing two IPUs on the front side connected to the CCU via the ICU on the back.

Confidential computing [1, 4, 11, 31] relies on custom hardware support for trusted execution environments (TEE), also known as enclaves, that can provide such security guarantees. Abstractly, a TEE is capable of hosting code and data while protecting them from privileged attackers. The hardware can also measure this code and data to issue an *attestation report*, which can be verified by any remote party to establish trust in the TEE. In principle, confidential computing enables multiple organizations to collaborate and train models using sensitive data, and to serve these models with assurance that their data and models remain protected. However, the predominant TEEs such as Intel SGX [22], AMD SEV-SNP [5], Intel TDX [16], and ARM CCA [6] are limited to CPUs. Recently, NVIDIA has announced TEE support in upcoming Hopper GPUs [25] that works in conjunction with CPU TEEs.

Adding native support for confidential computing into AI accelerators can greatly increase their security, but also involves many challenges. Security features such as isolation, attestation, and side-channel resilience must be fitted in their

highly optimized architecture, with minimal design changes, and without degrading their functionality, performance, or usability. An additional requirement is the flexibility to operate with various hosts, including CPUs with no TEE support, CPUs with process-based TEEs such as Intel SGX, and CPUs with VM-based TEEs such as AMD SEV-SNP. None of the accelerator TEE designs that have been proposed meet this requirement, including NVIDIA GPU TEEs. Finally, the manufacturing and assembly process and protocols must be hardened against supply chain attacks.

This paper describes our effort to support TEEs in Graphcore's GC200 Intelligence Processing Unit (IPU), a state-of-the-art custom AI accelerator. We introduce *IPU Trusted Extensions* (ITX), a set of experimental hardware capabilities in IPUs. We show that, using ITX in conjunction with appropriate compiler and runtime support, we can delegate ML tasks to the IPU with strong confidentiality and integrity guarantees while delivering accelerator-grade performance. In particular, ITX can guarantee isolation of an ML application from an untrusted host: application code and data appears in cleartext only within the IPU, and remains encrypted otherwise, including when transferred over the PCIe link between the host and the IPU. Once an application is deployed within an ITX TEE, the host can no longer tamper with the application state or the IPU configuration. ITX can also issue remotely verifiable attestations, rooted in a Public Key Infrastructure (PKI), enabling a relying party to establish trust in a given ML task before releasing secrets such as data decryption keys.

The main components of ITX are a new execution mode in the IPU for isolating all security sensitive state from the host and securely handling security exceptions, programmable cryptographic engines capable of encrypting/decrypting CPU-IPU PCIe traffic at line rate (32 GB/s bidirectional throughput supporting PCIe Gen4), and a novel authenticated encryption protocol for ensuring confidentiality and integrity of code/data transfers without requiring trust in the host.

Trust in ITX is rooted in the *Confidential Compute Unit (CCU)*, a new hardware Root-of-Trust (HRoT) on the IPU board. The CCU provides each device with a unique identity based on a hardware secret sampled within the CCU at the end of manufacturing. The CCU firmware is responsible for managing the entire lifecycle of TEEs on the IPU, including creation, issuing attestation reports that capture IPU and task specific attributes, key exchange, launch, and termination of TEEs. Our design also features protocols for securely provisioning firmware to the IPU in a potentially hostile manufacturing environment, for issuing certificates that capture the identity of all updatable firmware, and for supporting firmware updates without requiring device re-certification.

Several distinguishing aspects of ITX and the IPU programming model result in stronger security than one may expect from CPU-based TEEs:

- An ITX TEE spans the entire IPU, and has exclusive access to all IPU resources until it terminates. Therefore, it is not possible for an adversary to run concurrently on the same resources and exploit the resulting side-channels. This execution model is feasible as most AI workloads require at least one accelerator, with larger workloads requiring thousands of accelerators for hours.

- The IPU memory system consists of large on-chip SRAM attached to its cores, which is loaded with data from untrusted external memory during explicit synchronization phases. Thus, during computational phases, code and data accesses to IPU memory have a fixed latency. This has two security implications: (1) traffic between IPU cores and memory need not be encrypted, as it stays within the chip; (2) this avoids the need for optimizations such as caching or speculation to hide memory access latency, and the resulting side-channels.

- The IPU supports a programming model where allocation and scheduling of all resources on the IPU (cores, memory, and communication channels) are statically managed by the compiler. Hence, the IPU application binary defines its entire data and control flow, including data transfers within the IPU, and between IPU and host memory. This is unlike GPUs where the host software stack (runtime and driver) remain in full control of the execution, and must be trusted to guarantee integrity.

There are many ways for software to utilize ITX to provide end-to-end guarantees for ML workloads, depending on the threat model and capabilities of the host. This paper focuses on configurations where a multi-party ML training workload is deployed to the IPU *without trusting the host CPU*. This mode has the strongest security properties and can be used with any CPU. We describe a prototype software stack and protocols for it, and present its end-to-end evaluation using standard DNN training workloads. Software to support other configurations, e.g., where the IPU is coupled with a hardware-protected CPU TEE, are left for future work.

We have fully implemented ITX in the IPU, taped out in 2020 and manufactured in TSMC's 7nm node. Our extensions use less than 1% of this large ASIC, and do not require any changes to its compute core or memory subsystem. Its evaluation on a development board using confidential ML training workloads suggests a performance overhead of less than 5% compared to non-confidential IPU workloads. While our prototype demonstrated promising results, significant work remains to turn our work into production.

Due to implementation constraints, our prototype uses a discrete HRoT (instead of an on-die core) and it does not encrypt traffic over IPU-IPU links. It is therefore vulnerable to physical attacks, e.g., on the link between the CCU and IPU, or between multiple IPUs. These vulnerabilities are not limitations of our design and can be addressed in future IPU generations by integrating the HRoT on the IPU chip, and by introducing encryption engines on IPU-IPU links.

In summary, this paper makes the following contributions:

1. A set of experimental hardware extensions to the IPU, a commodity custom AI accelerator, that enable high-performance confidential multi-party machine learning.
2. Support for remote attestation and secure key exchange based on a discrete hardware root-of-trust.
3. A pipelined application-level protocol for authenticated encryption & decryption of code and data over PCIe.
4. Protocols for securely provisioning secrets, firmware, and certificates to a device during manufacturing.
5. Prototype software support for enabling confidential multi-party training of ML models expressed in TensorFlow on the IPUs without requiring trust in the CPU.
6. Hardware implementation of ITX in the IPU ASIC manufactured by TSMC in 7nm node, and its initial evaluation on a development board in 2020, suggesting low overheads and orders of magnitude improvements over CPU TEEs. This made our prototype the first AI accelerator to support confidential computing.

While some aspects of our design are specific to IPUs, we hope it can serve as a blueprint for adding TEE support in other specialized devices and accelerators.

2 Background

This section outlines the IPU architecture and programming model, focusing on aspects relevant to security. The section also reviews hardware-based confidential computing.

2.1 IPU Hardware Architecture

Tiles. Each IPU consists of a set of *tiles*, each with a multi-threaded core and a small amount of private on-chip SRAM. The IPU features 1472 tiles, totalling roughly 900 MB of on-chip SRAM. The cores support an instruction set tuned for AI, including specialized vector instructions and low-precision arithmetic. Each core can execute up to six statically scheduled threads. Since on-chip memory is accessed at fixed latency, instructions can be exactly scheduled by the compiler.

Interconnects. The tiles are connected over *internal exchange*, an all-to-all, stateless, synchronous and non-blocking high-bandwidth interconnect whose operation is similarly orchestrated by software. The internal exchange is connected to an *external exchange* interconnect via a set of *exchange blocks*. Each exchange block manages a subset of the tiles and mediates traffic between the two interconnects. Each IPU has a pair of PCIe links that connect to a host server, and additional IPU-Links that connect to other IPUs.

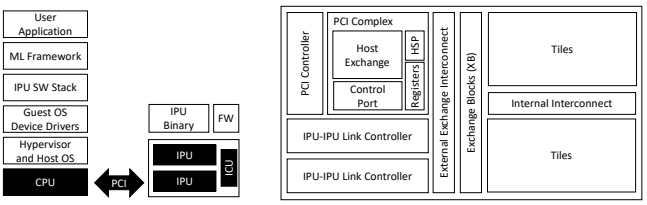

Figure 2: System stack (left) and IPU floorplan (right).

The external interconnect is a packet-switched Network-on-Chip. Tiles use the external interconnect to dispatch packets to the host via PCIe links and unicast/multi-cast packets to tiles on other IPUs via IPU-links. Tiles read data from the host by issuing a *read request* packet and waiting for all associated *read completion* packets. Tiles write data to the host by issuing one or more *write request* packets. Packets are routed based on tile identifiers. For requests, packets from exchange blocks are placed onto lanes based on the source tile identifier of the exchange packet. For read completions, the exchange lane is chosen based on the destination tile identifier, which is recorded in a lookup table in the PCI complex for each outstanding read request.

Exchange Address Spaces. The IPU exposes three address spaces to facilitate communication between the host and the IPU and between IPUs. The *Tile address space* is used by tiles to address one another. The *Host PCI space* is used by the host to address tile memory and on-chip page tables in the Host Exchange block. The *Tile PCI space* is used by tiles to address read requests to host memory over PCI. The IPU can be configured to re-map read requests from tiles to the PCI domain using on-chip page tables.

Host-IPU Interface. The IPU exposes a set of configuration registers to the host via a PCI BAR space. These registers are hosted in a component known as the *PCI Complex*. The PCI complex consists of a *Host Sync Proxy (HSP)* responsible for external synchronization between the host and the IPU, a *host exchange* that translates packets between PCI format and a proprietary external-exchange packet format, *on-chip page tables* for address translation of read/write requests from tiles to the PCI domain, on-chip lookup tables for keeping metadata for outstanding PCI read requests, and a *control port* that provides access to configuration registers of all other internal components.

The host exchange subsystem also includes a component known as the *autoloader*, which enables efficient scrubbing and initialization of tile memory. To initialize a binary in tile memory, the host can load small programs (e.g., a bootloader) into the autoloader, which can then broadcast it to all tiles.

Host-IPU Synchronization. The IPU execution model is based on the Bulk Synchronous Parallel (BSP) paradigm with barriers and supersteps. A superstep involves a global synchronization barrier between all tiles on one or more IPUs, followed by an exchange phase that transfers data between

tiles, followed by a compute phase which ends at another barrier. This process repeats until some application specific criteria is met—e.g., loss is under a threshold.

Using the HSP registers, the host can configure the frequency of synchronization barriers and indicate barriers at which it expects to be notified—e.g., when one or more batch of data has been processed, at epoch boundaries. Once configured, the IPUs can execute multiple supersteps independently without requiring involvement from the host.

IPU Control Unit (ICU). The ICU is a microcontroller integrated on the board and connected with the IPUs via JTAG, and with PCB peripherals for power supply and environmental monitoring. It is responsible for initialization of the IPUs.

Resets. The main means of resetting the IPU from the host is a *secondary bus reset (SBR)* that resets the entire device including the IPUs, the ICU, and the host link; the ICU must re-enable the host link once it comes out of reset. Alternatively, a *Newmanry Reset* can be triggered by writing the IPU control register; it resets the device logic including the host and IPU links, but does not reset the physical links. In both of these resets, tile memory is not scrubbed.

2.2 IPU Software Stack

The IPU software stack compiles and executes applications written in ML frameworks such as TensorFlow and PyTorch. It consists of a compiler, a host runtime, and a set of libraries supported by the IPU device driver.

Compiler. Given a computation graph representing a task (e.g., a TensorFlow XLA graph), the compiler partitions each layer of the graph between tiles, so that each tile holds a part of the model state (weights and activations for some layers) and a part of the input data. The compiler also assigns resources (threads, memory) to each node of the graph, schedules its computation, and finally emits specialized code for each tile.

The resulting IPU binary captures the different phases of execution, including I/O for reading batches of data, code for running the training loop, and I/O for writing the weights of the trained model. I/O phases also include synchronization and internal exchange code for exchanging data among tiles.

The compiler maps all data transfers between the host and IPUs to an abstraction called *streams* supported by the runtime. Data transfers from the host to an IPU (and IPU to the host) are mapped to input (output) streams and compiled to sequences of read (write) instructions to the Tile PCI address space. The compiler also uses streams to implement checkpoints: checkpoint creation maps all model weights to a single output stream, and checkpoint restoration reads them back from a single input stream.

The compiler supports an offline mode, which decouples compilation from execution. In this mode, the compiler generates self-contained IPU binaries, which can be persisted and loaded into one or more IPUs at a later point in time.

Host Runtime. The runtime provides code for loading IPU binaries, and for streaming data in and out of the IPUs. It loads IPU binaries by deploying a small bootloader into a reserved section of each tile memory. The bootloader in turn reads each tile-specific binary from the host into tile memory.

The runtime implements input streams by repeatedly copying data into a ring buffer in host memory and mapping the pages of the ring buffer into Tile PCI space in the on-chip page table. Once the ring buffer is ready and the mapping is defined, code on tiles can issue read requests. Similarly, output streams are implemented by copying data from the ring buffer to application memory.

2.3 Confidential Computing

Confidential computing is a paradigm where code/data remain protected from privileged attackers throughout their lifecycle: at rest, in transit, and *during use*. Central to confidential computing is the notion of a *trusted execution environment* (TEE) with two key capabilities: it can host an application in a hardware-isolated environment, which protects the application from any external access including access from privileged attackers; and it can issue remotely verifiable attestations that capture security claims about the application hosted in the TEE and the platform supporting the TEE. The attestations can be used by a relying party to gain trust in an application.

TEEs are supported by recent processors from Intel and AMD; ARM also recently defined a specification for supporting TEEs. There are broadly two classes of CPU TEEs: process-based and VM-based. Process-based TEEs (e.g., Intel SGX) are designed to isolate a user-space application from an untrusted operating system (both guest and host) and the hypervisor. VM-based TEEs (e.g., AMD SEV-SNP, Intel TDX) are designed to protect an entire guest VM from the host operating system and the hypervisor. TEEs offer varying degrees of protection from attackers with physical access to the CPU. Most TEE implementations assume that attackers can snoop on interconnects between the CPU package and external components (e.g., off-chip DRAM) and protect data by encrypting and integrity-protecting memory traffic.

Remote attestation is typically rooted in an on-die hardware root of trust (HRoT) with exclusive access to a unique device secret provisioned into one-time programmable fuses during manufacturing. During boot, the HRoT uses the secret to derive a device-specific identity key. This key typically endorses keys used for signing attestation reports. The corresponding public key is endorsed by the hardware manufacturer.

3 Threat Model

TEE hardware is subject to a variety of attacks throughout its lifecycle, from chip design and manufacturing up until the hardware is decommissioned.

Trust in TEEs is rooted in hardware, and consequently in the chip designers and their OEMs involved in designing and manufacturing the chips. Additional trust is also required in the infrastructure for issuing certificates to each chip, and for publishing the last known good version of firmware trusted computing base (TCB). While this is also the case with the IPU, we wish to minimize trust in the rest of the supply chain. Hence, we conservatively assume that attackers control the manufacturing and assembly process after tapeout, including the process of provisioning firmware and/or secrets to each device and harvesting their Certificate Signing Requests.

After deployment, we assume a strong adversary that controls the entire system software stack, including the hypervisor and the host operating system, and also has physical access to the host. The adversary can access or tamper with any code and data transferred between the host and the IPU, either in operating system buffers or over PCIe. The adversary can also tamper with device memory directly via the PCI BAR, or map the victim application's tile PCI address space to host-side memory controlled by the attacker. Information leakage through side-channels such as timing, power analysis, and physical probes on the IPU are generally out of scope. However, we wish to offer protection from side-channels based on memory access patterns, and from low-level integrity attacks such as glitching.

We trust the IPU and HRoT packages, and we assume that the adversary cannot extract secrets or corrupt state within the packages. In particular, the IPU includes trusted SRAM within the IPU tiles accessed only via on-chip channels.

With the current generation of IPUs, we make additional trust assumptions in the ICU, which provides connectivity between the hardware RoT and the IPUs, and in links between IPUs. We trust the ICU firmware and the physical links that connect the HRoT, the ICU and the IPU. These trust assumptions can be removed in subsequent generations of the IPU by placing the HRoT on the IPU die, and encrypting communication over IPU-IPU links.

The ML source script and configuration are trusted. The ML framework and the compiler are trusted for integrity of the computation—i.e., to compile the model defined in the ML script correctly into a manifest and IPU binaries. In multi-party configurations (involving parties that do not trust one another), these assumptions can be met by having all parties review the script and configuration for the workload, then confirm that they all locally compile to the same manifest and binaries. Each party is trusted with the integrity and confidentiality of the data streams they provide for the computation; in particular, honest parties are trusted to correctly encrypt their data streams with a fresh encryption key, and to release this key to IPUs only after verifying their attestation report.

In configurations that couple the IPU with a host CPU TEE (e.g., Intel SGX and TDX, AMD SEV-SNP), the CPU package is also trusted, along with any software hosted in the TEE.With process-based TEEs, the CPU-based software

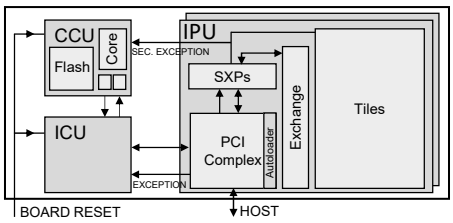

Figure 3: IPU hardware extensions to support trusted execution.

TCB may include the ML training or inferencing script and its framework (e.g., TensorFlow, PyTorch), compiler, and runtime. With VM-based TEEs the TCB may additionally include the kernel-mode driver and a guest operating system. The host runtime is trusted for confidentiality—i.e., to setup a secure, attested channel between the CPU TEE and IPU, and to transfer code/data over the channel.

Under this threat model, we wish to provide confidentiality and integrity guarantees for model code and data, including initial weights, input data, checkpoints and outputs. For training, integrity implies that the trained model is bitwise equivalent to the model obtained in the absence of the attacker. For inferencing, integrity implies that requests yield same results as those obtained in the absence of the attacker.

We wish to provide remote attestation, which refers to the ability of the platform to make remotely verifiable claims that a relying party can use to reason about the TEE's security properties and thereby establish trust in the application hosted within the TEE. Specifically, we wish to ensure that the attestation can deliver temporally fresh evidence that contains all security-sensitive parts of the platform and application state.

4 Overview

Trusted execution in IPUs enables model developers to securely offload an ML job (training or inference) while protecting both its model and data from the hosting platform. In turn, model developers can prove to data providers that their data remains protected from both the hosting platform and the model developers themselves. (Appendix A.1 provides a comprehensive security analysis of ITX.)

The workflow for offloading a job involves TEE creation, generation of an attestation report, its verification by remote parties, code/data encryption, secure exchange of encryption keys, job execution, and decryption of the outcome.

4.1 Hardware Extensions (ITX)

The IPU hardware contains several components (shown in Figure 3) to support this workflow, including a new hardware root-of-trust (RoT), called the CCU, and a new execution mode, called the *trusted mode*, in which all security sensitive state is isolated from a potentially malicious host. This mode

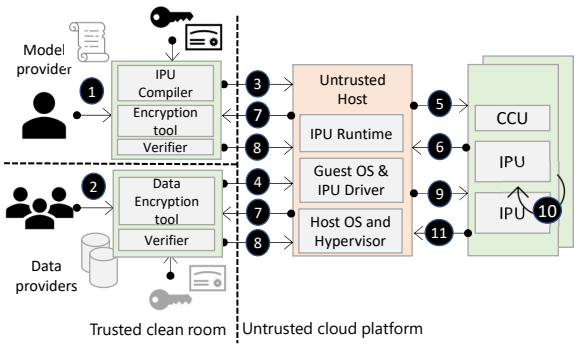

Figure 4: Multi-party training in trusted offline mode. Before training, the remote parties upload their encrypted code, data and certificates (1–4) Once training starts (5–6), they verify the attestation report (7) then release their encryption keys to the CCU (8-9); they can be offline for the rest of the computation. The IPUs train the model in a TEE (10) and releases an encrypted trained model (11), whose key can be shared with model receiver(s).

is entered by writing to a configuration register. (For remote verifiability, this register is measured by the CCU and included in the attestation report.) Once the IPU enters this mode, its configuration registers and tile memory can be accessed only by the CCU and ICU. The only way to exit this mode is via a chip reset, which is extended to scrub all key registers and tile memory.

The IPU also includes programmable AES-GCM engines for authenticated encryption and decryption of code and data transferred between the host and the IPU at PCIe line rate. These engines are hosted in new components, called *Secure Exchange Pipes (SXP)*, located on the interconnect between the PCIe block and the exchange blocks. The SXP and its use are described in Section 6.1.

4.2 Software Support

There are many ways to utilize ITX. For this paper, we illustrate a particular mode, called the *offline mode* (Figure 4). In this mode, a multiparty ML training workload can be deployed in an IPU-based TEE *without requiring a CPU-based TEE*. This mode has strong security properties (e.g., small TCB) and minimal dependencies on the host CPU hardware.

Job Preparation. In offline mode, a model developer uses an extended IPU compiler to statically compile a model training job expressed in an ML framework such as TensorFlow or PyTorch to standalone IPU binaries in a trusted, offline *clean room environment* (1). In addition to the binary, the compiler generates a *job manifest*, which contains auxiliary information required at runtime to execute the job. Next, the model developer encrypts binaries and parameters such as initial weights and learning rate using encryption keys that remain in the clean room environment. The model developer also generates a fresh public key share for key exchange, and a signature over the key share using their certificate. These artifacts, along with the model developer's certificate are packaged together to create an *application package*. Separately, data providers pre-process and encrypt their input data and labels in their own clean room environments, and create *data packages* which include their key shares and certificates (2). The resulting packages are uploaded to an IPU server (3, 4).

Job Initialization. Any entity can initiate execution of the training job using the host runtime, which has been extended to load encrypted code/data into IPUs. For confidential computing jobs, the runtime provides user-mode APIs for operations such as creating TEEs (5) for a job, fetching their attestation reports and additional collateral such as device-specific certificates (6), and relaying key-exchange messages from relying parties to the CCU (8). This runtime is not trusted.

Remote Attestation. In trusted mode, the CCU can issue remotely verifiable attestations, which are relayed to relying parties (7) as proof of TEE configuration for their workload. The attestation is a certificate chain from the IPU manufacturer root CA to an end-certificate signed by the CCU with custom extensions that embed initialization attributes (e.g., measurement of all security-sensitive IPU registers) and job-specific attributes, such as the measurement of the job manifest, and the hash digest of other runtime attributes, including certificate fingerprints of all parties and the CCU's fresh public keyshare. The model developer and data providers verify this report, the model, and identities of other participants. If they decide to make their data available for this job, they derive shared keys using the CCU's public key share and securely exchange their secrets with the CCU.

Job Execution. After the model developer and data providers have released their keys to the CCU, the CCU deploys the keys into the SXPs and starts the job (9) by installing a bootloader into the IPU tiles using the autoloader. The bootloader fetches the application binary from host memory to each tile in 1KB blocks. In trusted mode, these blocks are decrypted and integrity checked by the SXPs before being written to tile memory (see Section 7.3). Once the application binary has been transferred, the runtime initiates execution of the job. During execution, tiles generate read requests for data, also in blocks of 1KB. In trusted mode, the blocks are fetched from host memory over PCIe, and decrypted and integrity checked by the SXPs before being written to tile memory. Similarly, all write requests (e.g., checkpoints and trained model) are encrypted and extended with authentication tags before being written to host memory. The encryption protocol is mostly transparent to the compiler, which can compile *any training algorithm* into binary relying on the data being in tile memory in cleartext and utilizing all available IPU compute resources. Finally, the IPU encrypts the trained model with a key made available only to the model receivers listed in the job manifest.

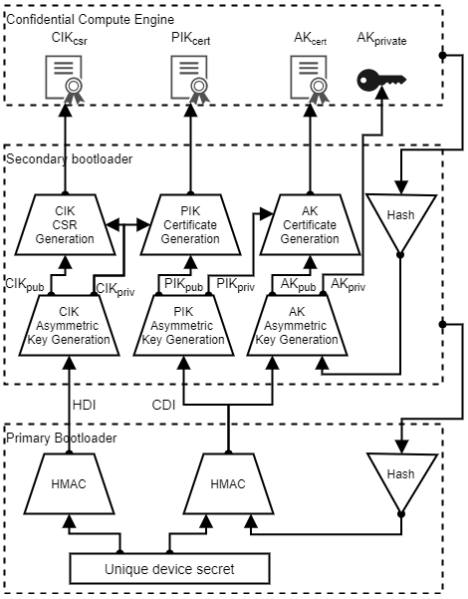

Figure 5: CCU firmware architecture and key hierarchy.

5 Trusted Execution on IPUs

5.1 Confidential Compute Unit (CCU)

The CCU is responsible for associating each device with a unique cryptographic identity and managing trusted execution in its IPUs. The CCU is a discrete chip based on STMicro's STM32H753 microcontroller [24]. This chip was selected as the RoT based on several security features required to implement measured boot and to offer protection from a variety of attacks throughout the IPU lifecycle, such as the ability to provision a custom bootloader during manufacturing and a mode that prevents external access via interfaces such as JTAG. As shown in Figure 3, the CCU is connected to the IPU via the ICU. A dedicated pin receives all exceptions generated by the IPU in trusted mode, giving the CCU firmware full control over exception handling. The CCU reset pin is coupled in hardware with the ICU reset pin and IPU reset, so they cannot be independently reset.

Firmware Architecture and Attestation. The CCU implements a measured boot protocol which is a variant of the Device Identity Composition Engine (DICE) architecture [12, 33]. DICE ensures that each device is assigned a unique identity while minimizing exposure of hardware secrets. Except for the stable device identity, all derived secrets and keys automatically change when firmware (and its measurement) changes, which ensures that low-level firmware attacks do not compromise secrets used within other firmware.

The CCU firmware (Figure 5) consists of three layers: an immutable primary bootloader provisioned in one-time programmable flash memory at manufacturing; a mutable secondary bootloader responsible for device identity and attestation certificates; and a confidential compute engine (CCE) that manages the TEE lifecycle.

During manufacturing, the CCU is provisioned with the primary bootloader firmware. When the device is brought out of reset for the first time, this primary bootloader receives control from ROM firmware, samples a *unique device secret* (UDS) using a hardware-based TRNG, stores it in a region of flash memory, and permanently blocks its access from any other firmware layers. The UDS is the root of the IPU key hierarchy, and this protocol ensures that it is never exposed outside the CCU, not even to the manufacturer.

On every subsequent boot, the CCU loads and authenticates the secondary bootloader from flash using the IPU manufacturer's firmware signing key. Next, it derives two intermediate secrets: a *Hardware Device Identifier* (HDI) from UDS, and a *Composite Device Identifier* (CDI) from UDS and the measurement of the secondary bootloader. HDI is unique to each card, while CDI is unique to each card and secondary bootloader. It then scrubs UDS from memory and transfers control to the secondary bootloader, handing over HDI and CDI.

The secondary bootloader further derives two public-private key pairs: a *Card Identity Key* (CIK) from HDI, and a *Platform Identity Key* (PIK) from CDI. Hence, CIK gives each card a stable identity whereas PIK is unique to each card and secondary bootloader. The bootloader also generates a self-signed CSR for CIK, a PIK CSR, and a PIK certificate signed by CIK. The PIK CSR and certificate contain a custom extension that records measurements of the secondary bootloader and the ICU firmware along with additional device-specific information. The CSRs can be securely harvested during manufacturing and endorsed by the IPU manufacturer CA, which issues CIK and PIK certificates. (See Appendix A.2.)

The secondary bootloader derives the *Attestation Key* (AK) from CDI and the CCE measurement. Hence, AK is unique to each device, secondary bootloader, and CCE. The bootloader issues an AK certificate with the CCE measurement in a custom extension, signed by PIK, and finally scrubs all secrets and transfers control to CCE, handing over AK.

The CCE uses AK to sign attestation reports containing IPU- and job-specific information (Section 5.2). A relying party can validate attestation reports using the device-issued AK certificate, and manufacturer-issued CIK/PIK certificates.

Firmware Update. Per DICE, a secondary bootloader update invalidates PIK certificates issued by the manufacturer and, as UDS is provisioned within each device, the IPU manufacturer cannot independently derive and certify the updated PIK. Instead, we rely on CIK, acting as a local CA, to sign the updated PIK certificate. Additionally, the manufacturer would issue TCB update certificates containing measurements of old and new versions of firmware. A relying party can validate attestations using device-issued PIK certificates, the original PIK and CIK certificates, and TCB update certificates. (See Appendix A.3–A.4 for more details and a hardened variant.)

5.2 TEE Lifecycle Management

The CCU exposes an API for TEE management on the IPU.

TEE Initialization. The first step in securely offloading a job to an IPU is to create a fresh TEE for this job. TEE initialization requires a job manifest (Appendix A.5), public key shares, signatures over the key shares and certificates for each relying party, and a checkpoint counter indicating whether the job is starting or resuming from a checkpoint. During TEE initialization, the CCU first *quiesces* the IPU, ensuring that there are no in-flight read and write requests between the host and IPU. It then switches the IPU into trusted mode, scrubs all tile memory using the autoloader, and measures the state of the configuration registers. It then checks the signatures over the key shares using the certificates, and generates its own fresh EC share, which is used to establish an ECDH shared secret between each relying party and the CCU.

The CCU generates an attestation report signed by the attestation key containing various IPU-specific attributes (e.g., configuration register measurements) and job-specific attributes such as the job manifest, certificate fingerprints for all parties, and the checkpoint counter consisting of the epoch counter and checkpoint identifier. (See Appendix A.6 for the details.)

Each relying party can review the attestation together with the supporting certificate chain, to validate the device and the initial state of the CCU and IPU, then it can compute its ECDH shared secret and wrap a key package that contains the party's data encryption keys and nonces to run the job. (See Table 2 in Appendix A.6 for the keying details.)

TEE Launch. After gathering wrapped encryption keys from all relying parties, the host launches the execution of a job.

First, the CCU computes the ECDH shared secret for each party and uses them to unwrap the key package(s) received from each party. It then combines the nonces to derive a checkpoint key and a final-model encryption key for this run of the job (and, if resuming from another run, the checkpoint key from that previous run to restore its state). This key derivation ensures both that the checkpoint key for this run is fresh (as long as one relying party's nonce is fresh) and that the checkpoint key of a prior run can be recomputed once all relying parties agree to resume from a checkpoint. (See Table 2 in Appendix A.6 for the keying details.)

Next, the CCU deploys a pre-defined bootloader on the IPU tiles using the autoloader, and it deploys a first set of encryption keys to the SXP (including the model key) as specified in the job manifest. It then activates the bootloader (whose measurement is included in the attestation report) on every tile, which issues requests to read their encrypted application binary from host memory. Responses to these read requests are authenticated and decrypted by the SXPs before being copied into private tile memory.

Finally, the CCU deploys the next set of encryption keys (including data keys) as specified in the job manifest, and triggers the main execution loop on the IPU tiles.

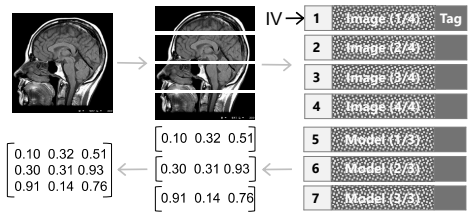

Figure 6: Authenticated encryption with explicit IVs. Data is partitioned into frames with unique IVs. Hardware-level decryption ensures their integrity based on their authentication tag; the receiver must verify that frame IVs match the expected IVs.

TEE Termination. At any point after initialization of a TEE, the host runtime can also request that the TEE be terminated. The CCU may also terminate the TEE in the event of a security exception raised from the IPU such as failure to authenticate a response of a read request. During TEE termination, the CCU quiesces the IPUs, scrubs tile memory using the autoloader, and disables all SXP keys. Finally, the CCU switches the IPU into normal mode via *Newmanry* reset.

A TEE may also be terminated by a hard reset of the device. In this case, all CCU state is cleared and the IPU reverts to normal mode. When it comes out of reset, prior to re-enabling the host links, the ICU scrubs tile memory to ensure that any secrets left over from a previous execution are erased before the host re-gains access to the device.

6 Encrypted Direct Memory Access

Next, we describe the ITX protocol for encrypted code and data transfers to/from IPU tiles. The protocol is application-level as opposed to transport-level. While it is transparent to ML frameworks, it relies on application software (e.g., the IPU compiler) to assign IVs for authenticated encryption and to program the tiles to securely load code, initial weights, training data, and save/reload checkpoints and results. The protocol is supported in IPU hardware by fully pipelined AES256-GCM engines for authenticated encryption at PCIe line rate. This choice results in simpler hardware, allows the IPU to be coupled with untrusted CPUs (or CPUs with varying TEE support) and retains the compiler's ability to maximize PCIe utilization by parallelizing data transfers across tiles.

6.1 Data Format

In the encryption protocol (illustrated in Figure 6), application software partitions each code and data stream into equally-sized encrypted *frames*. Each frame consists of a 128-bit IV, followed by a series of cipher blocks that carry the encrypted contents of the frame, and by a 128-bit authentication tag. Application software is free to use different frame sizes for different streams, as long as the total frame size (including IV and authentication tag) is a multiple of 128 bytes with a maximum of 1KB, which is the largest supported PCIe read. Application

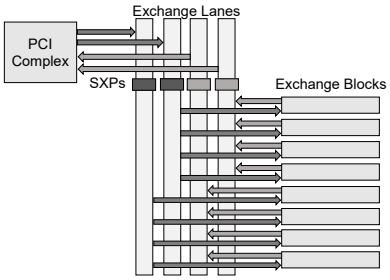

Figure 7: IPU External Exchange Interconnect, with an SXP on each exchange lane. Traffic is forwarded from and to exchange blocks to exchange lanes based on the exchange block identifier.

software can use different keys to encrypt different streams. This is critical for multi-party scenarios where streams are provided and accessed by different parties. Crucially, application software must ensure that IVs are never reused across frames encrypted with the same key, which would be catastrophic with AES-GCM. In our implementation, this invariant is ensured by the compiler, which constructs the IV by combining stream-specific identifiers and frame indexes, and the fact that both code and data streams are write-once abstractions. Together, this guarantees that unless the associated key has been compromised, authenticated decryption with the correct IV yields the correct payload.

6.2 Hardware Support

Multiple components in the IPU support ITX encryption. The IPU includes blocks, called *Secure Exchange Pipes*, extensions to packet formats for carrying encryption-related information, and extensions to exchange blocks and the PCIe complex for supporting the task of mapping frames to keys.

Secure Exchange Pipe (SXP). The SXP is a programmable hardware block that supports AES256-GCM authenticated encryption and decryption of frames. Each SXP achieves 16 GBps unidirectional throughput with negligible impact on latency. As shown in Figure 7, there are four SXPs placed on the exchange interconnect (two per direction) to support encryption/decryption at PCIe Gen4 line rate (32GBps bidirectional). In trusted mode, each SXP is configured to intercept read/write requests from four exchange blocks.

AES-GCM Engine. The SXP's core is a fully pipelined AES-GCM engine that supports 16 *physical key contexts* to enable concurrent requests. Each context can be programmed by loading a 256-bit key into control registers exposed to the CCU via an internal control bus. While frames may be interleaved, for functional correctness we require that *each context processes a single frame at a time*. This invariant is enforced by the compiler, as detailed in Section 7.1.

The core implements the standardized AES256-GCM algorithm with two restrictions: the additional authenticated data is always empty; and the plaintext is block-aligned and not empty. For convenience, we also treat the IV as a full 16-byte block, including the 32 bits of internal block counter. In each cycle, the core performs one of the following operations on its context: (i) the context is idle and the core receives the IV for the frame, (ii) the context is active and the core receives a block of data, or (iii) the context is active and the core receives a MAC. The core detects context switches by comparing key context identifiers between consecutive cycles, so that it can fetch the next context before the next operation.

Frame encryption/decryption. The SXP receives three types of external exchange packets: read requests (egress); read completions requiring decryption (ingress); and write requests requiring encryption (egress). Their headers are extended to carry additional information to help the SXP determine how the packets should be handled: an `AES` bit indicates that the read completion or the write request is encrypted; a 4-bit `KEY_INDEX` field identifies the physical key context to use; and a `CC` bit indicates the last packet of the frame and triggers the computation of its authentication tag.

In write request packets (outbound to the host PCIe domain), the `AES` and `CC` bits are set by the tile, whereas the `KEY_INDEX` is set by the SXP. In read completions packets, the information is set by the PCIe complex based on trusted state it maintains about pending read requests.

Read request packets and packets with the `AES` bit unset do not require encryption/decryption; they are passed unchanged. For all other packets, the header bypasses the AES core, then the AES core handles each packet (and its blocks) depending on whether a frame starts, a frame continues, or a frame ends.

Key Selection. Each SXP supports multiple physical key contexts to enable encryption/decryption of concurrent I/Os. The SXP provides a set of programmable (by CCU) registers to define a mapping between packets and the physical key context to use for encrypting/decrypting their payload. The compiler produces this mapping by assigning a set of tiles associated with an exchange block context to access a single stream. Upon receiving a packet, the SXP looks up the physical key context using the exchange block context index computed from the source tile identifier in the header.

Once the SXP infers the physical key context, it updates the `KEY_INDEX` field in the request packet header. For write requests, the field is then used by the SXP to switch the AES core to the inferred physical context for encrypting its payload. However, for read requests, the situation is more involved, as the read requests bypass the AES core, and the inferred physical key context must be used to decrypt the read completions that will be returned by the host after the read request has been processed. When the PCI complex receives the read request, it caches the `KEY_INDEX` and `AES` fields in an on-chip lookup table along with other metadata, such as the source tile identifier. When the corresponding read completions arrive from the host, the PCI complex retrieves these fields from this lookup tables and inserts them into the read completion packets. The SXP can then use these values to identify the physical key context to use for decrypting the payload. The

PCI complex tracks the number of pending read completion packets for each request, and sets the `CC` bit on the last one.

7 Software Extensions

We now describe a set of extensions to the IPU software stack to compile and execute confidential ML tasks using ITX in offline mode. This mode is triggered by a Tensorflow configuration option. When enabled: (1) The XLA backend transforms the computation graph to use a new abstraction called *confidential data streams* for all data transfers including initial weights, training data, checkpoints and the trained model. (2) The IPU compiler compiles the computation graph into a set of IPU application binaries (one for each IPU), where each binary is a concatenation of tile-specific binaries. The compiler encrypts tile binaries into a set of encrypted frames using a freshly sampled model key. A frame is assigned a unique IV comprised of code type, IPU/tile IDs and frame index. (3) The IPU runtime is extended to securely bootstrap the task, then transfer the encrypted binaries and data between the host and IPU. (See Appendix A.8 for a sample scenario.)

7.1 Confidential Data Streams

Confidential data streams is a compiler and runtime abstraction for transferring data to/from the IPU with confidentiality and integrity guarantees, leveraging SXPs. Each stream is a sequence of data instances encrypted with the same symmetric encryption key. Each data instance is partitioned into a sequence of frames, and each frame is encrypted using a unique IV composed of a stream type (data), a stream identifier, and the index of the frame within the stream.

The compiler and the runtime implement reads and writes to confidential data streams as follows. As discussed in Section 6.1, the compiler first assigns a region in tile PCI space to each stream, subject to the constraint that it never exceeds the total capacity of the IPU ring buffer (e.g., 256 MB).

Next, the compiler assigns sets of tiles to read from or write to each stream, reserves SRAM on each tile to hold a part of the stream, and generates SXP mappings, subject to the constraints that (a) the exchange block context associated with these tiles map to physical key contexts assigned to the stream, and (b) the number of physical key contexts in use at any point in the program does not exceed 16 for any SXP.

To maximize performance under these constraints, the compiler may introduce synchronization points in the application where existing keys are invalidated and new keys are loaded. The compiler includes these synchronization points in the job manifest, along with their key identifiers; and the (untrusted) IPU runtime uses this part of the manifest to ask the CCU to load the next decryption keys into the SXPs at these points. The key changes apply only to input streams. Keys for output streams are derived and loaded by the CCU at TEE launch, and do not change throughput its lifetime. A malicious runtime not following the job manifest's key schedule can only cause decryption failures, resulting at most in denial-of-service.

Next, the compiler schedules read/write operations on each tile. The schedule is required to satisfy a hardware constraint that, at any point, the tiles that generate requests targeting any given physical key context be associated with a single exchange block context. This is because, while the exchange block can dynamically synchronize and regulate requests within each exchange block context (so that its physical key context is used by one tile at a time) there is no such synchronization across exchange block contexts.

Finally, the compiler generates code on each tile that implements the schedule, to issue read/write requests for the accessed frames. (Details are omitted due to space constraints.)

7.2 Secure Checkpointing

Checkpoints are saved to and restored from host memory, and are implemented via data streams. Secure checkpointing is implemented via a special form of confidential data streams, where the IV captures the epoch (counting the number of checkpoint resumptions for this job) and the checkpoint identifier (counting the number of checkpoints stored within an epoch.) The CCU uses a separate checkpoint key for each epoch, and makes the epoch counter and checkpoint identifier available to IPU tiles via the bootloader at the start of the application. (See Appendix A.7 for more details.)

7.3 Secure Bootstrapping

Secure bootstrapping is the process of securely loading encrypted application binaries into the IPU, either at the start of a job, or while resuming a job from a checkpoint.

Bootstrapping involves the following steps. First, the IPU runtime loads the encrypted IPU binary in host memory and creates a TEE using the CCU APIs; this switches the IPU into trusted mode. Next, the CCU installs a *bootloader* (shown in Appendix A.5) onto every IPU tile using the autoloader described in Section 2.1, and also configures the SXPs with the model-decryption key. The bootloader on each tile fetches the tile's binary from host memory by issuing a sequence of read requests. Each frame received from the host is intercepted by the SXPs, authenticated and decrypted, and copied into tile memory. The bootloader then checks that the received IV matches the expected IV built into the bootloader logic; this check is performed in software because the SXPs only guarantee authenticity of each frame, not the integrity of the entire stream. Failure of this check indicates an attempt by the host to tamper with the code stream, such as by replaying/reordering frames. In such event, the tile raises a security exception, which is handled by the CCU. If all checks pass, the bootloader reconstructs the original binary by stripping IVs and authentication tags from all frames.

Finally, the bootloader computes a hash of the tile binary; the tiles accumulate a hash of the whole application; and the CCU checks that it matches the measurement in the job manifest, or generates a security exception otherwise. This protocol, together with bootloader integrity (its measurement is included in the attestation) guarantees application integrity.

8 Evaluation

Our evaluation focuses on TEE overheads for ML training when using CPUs and IPUs.

Implementation. We have implemented ITX on the IPU on a non-production development board. The IPU chip has been fabricated in TSMC's 7nm node, including the on-chip security extensions, which account for $< 1\%$ of the chip size. As part of post-fabrication validation, these extensions have been tested to verify they conform to their specified behavior.

We have integrated the CCU on the board and implemented the architecture described in Section 5, including the protocols for measured boot and TEE management.

We have implemented a software prototype for confidential training tasks where the host CPU server is untrusted. Our prototype includes experimental support in the ML framework, IPU compiler and runtime. There are a few gaps in our prototype: (1) our implementation currently supports only one IPU on the board; (2) the compiler makes use of only one logical key region onto which code, data, label, checkpoints, and outputs are mapped; nevertheless, every encrypted frame is statically assigned a unique IV, preserving the invariant that each IV is used only once; (3) secure resumption is not yet implemented; and (4) the bootloader deployed on IPU tiles does not measure the IPU binary after decryption.

Experimental Results. Figure 8 summarizes the hardware and software configuration of our testbeds. We evaluate the performance of confidential training on ResNet models of various sizes (20, 56, and 110) on the Cifar-10 dataset. The dataset consists of 60,000 32x32 images spanning 10 classes; 50,000 of these images are used for training the dataset and the remaining are used for testing the resulting model. We ran the same training code and data configurations in clear and confidential modes, and confirmed that they both yield models with the same prediction accuracy.

We compare IPU TEEs against CPU TEEs based on the largest available AMD SEV-SNP server. The early IPU development boards operate at reduced frequency of 900 MHz. The AMD CPU testbed utilizes 48 single-threaded cores; hyper-threading does not improve performance due to high vector unit utilization leaving little room for another hyper-thread. Scaling from 32 to 48 cores improved performance by 10%.

Figure 9 shows the training throughput that we achieve in clear and confidential modes. IPU-based training even with a single IPU operating at reduced frequency is 12-20x and 13-17x faster than CPU-based training in clear and confidential

Testbed	Training configuration
AMD SEV-SNP 48-core VM on EPYC 7763	ResNet-20. Batch size: 1534; 32 epochs. ResNet-56. Batch size: 768; 32 epochs. ResNet-110. Batch size: 384; 64 epochs.
ITX IPU @ 900 MHz, Intel Xeon 8168	ResNet-20. Batch size: 64; 32 epochs. ResNet-56. Batch size: 32; 32 epochs. ResNet-110. Batch size: 16; 64 epochs.

Figure 8: Testbed configuration for TensorFlow training of ResNet models on Cifar-10 dataset. In each configuration, batch sizes are optimized to yield maximum performance. (Smaller batches do not affect correctness, but may improve convergence or accuracy.)

Figure 9: Training throughput of ResNet models on cifar-10.

modes respectively. Enabling SEV-SNP introduces modest overheads, ranging from 8% (small model) to 14% (large model) while the overheads of enabling ITX range from 3% (large model) to 58% (small model). The ITX overhead is dominated by one-time setup cost, which is amortized over large training times; this cost includes TEE initialization and attestation (40%), TEE launch including SXP setup (43%) and TEE termination including SXP scrubbing (13%). Runtime encryption introduces only 4% of the total overhead. More generally, we expect the one-time cost to be negligible with state-of-the-art models, which take weeks or days to train. With ResNet-110 model, the overall overhead is just 3% (1123 vs 1089 seconds for running 64 epochs). We also expect that utilizing both IPUs at full frequency would deliver an additional performance improvement (up to 3.5x) over CPUs.

In summary, the evaluation shows that using ITX, AI workloads can continue to benefit from the use of accelerators without compromising on performance or security.

9 Discussion

Trusted CPU TEEs. While this paper mainly focuses on a configuration where IPUs are attached to an untrusted CPU, ITX supports configurations with varying trust in CPU TEEs.

For instance, ITX can be used in a configuration where IPUs are coupled with a process-based CPU TEE (e.g., Intel SGX) hosting TensorFlow, the IPU compiler and the IPU runtime, with the IPU kernel-mode driver and the OS running outside the TEE. In this configuration, the enclave would receive an encrypted model script from the model developer,

and the IPU runtime would encrypt the compiled IPU binary with fresh keys. Similarly, the enclave would receive encrypted training data from data providers on the basis of an Intel SGX and IPU attestation. Within the process-based CPU TEE, the data can be decrypted, pre-processed and aggregated (in parallel with job execution), and re-encrypted by the IPU runtime with fresh keys. Encrypted code/data still then need to be copied to a run buffer allocated outside the enclave (in the host process) accessible to the IPU. While this configuration has a larger TCB and incurs higher CPU cost, it offers greater flexibility and support for inferencing scenarios.

Inferencing. While this paper mainly focuses on training, ITX does support inferencing workloads. Scenarios, where the number of remote inference clients using a model is small and mostly static, can be served by the existing architecture (no trust in the CPU) without additional overhead by assigning a key context to each client. Scenarios, where the number of clients is large and dynamic, could be supported by introducing a trusted front-end server component (in a process-based TEE on the host CPU or a remote CPU) that terminates TLS connections, receives inference requests from the clients, and re-encrypts them in batches using a smaller number of key contexts on the IPU (to avoid frequent expensive re-keying in the IPU). This architecture still allows decoupling the choice of CPU TEEs and accelerator TEEs. This is unlike GPU TEEs that require the CPU and GPU TEEs to be on the same platform. Finally, the TCB of the re-encrypting TEE is relatively small and independent of the application.

Side-channels. Our design intends to minimize leakage from side-channels. On the IPU itself, leakage due to memory access patterns and timing is minimized for two reasons. First, computation on the IPU is statically scheduled by a trusted compiler. It is therefore possible to analyze a workloads at compile time to ensure that memory access patterns are data independent, or add padding otherwise. Second, the IPU cores do not rely on speculative execution and on-chip memory accesses incur a fixed latency. As a result, all I/O between the untrusted host CPU and IPUs occurs at fixed time intervals. Thus, the attacker can observe the time taken to process an entire batch, as opposed to time taken to process each layer in the model [15].

10 Related Work

Trusted hardware. There is a history of work [7, 8, 10, 13, 20, 20, 21, 27, 32, 35] on trusted hardware that isolates code/ data from the rest of the system. Intel SGX [22] and AMD SEV-SNP [5] are the latest in this line of work. Our work extends this approach from general-purpose CPUs to accelerators.

Trusted execution on accelerators. Our work is the first to demonstrate an ASIC with confidential computing capabilities and the only one that does not require trust in CPU TEEs.

NVIDIA recently announced confidential computing support in upcoming Hopper GPUs [25]. Their design shares the same core principles as ITX on IPUs. Hopper GPUs are equipped with an on-package hardware RoT responsible for attestation and enforcing course-grained GPU isolation under the assumption that on-package GPU memory is trusted. Hopper GPUs also support encrypted and integrity-protected communications (kernels and data) to and from the GPU. However, the NVIDIA design requires a VM-based CPU TEE as the responsibility of attesting and establishing a secure channel with the GPU lies within the kernel-mode driver.

Numerous mechanisms have been proposed to enable CPU TEEs to securely interact with I/O devices—e.g., GPUs [17, 34, 37], FPGAs [19, 26, 30, 38], and AI accelerators [14, 36, 39]. Some of this work has attempted to reduce trust on privileged host via hardware support on the GPU [34] or on the CPU [17]. Graviton [34] extends the GPU with support for secure resource management, and relies on a trusted GPU runtime hosted in a process-based CPU TEE to manage the TEE lifecycle. HIX [17] requires extensions to process-based CPU TEEs, including the PCI interconnect and the CPU's MMU. GuardNN has attempted to remove the CPU from the TCB [14] by introducing instructions for establishing a secure channel between remote users and the device, and for decrypting/encrypting inputs/outputs. However, such architecture does not guarantee integrity as the instruction schedule can be tampered by attackers controlling the CPU.

TEE-I/O. In parallel with our work, there has been an industry-wide effort to develop TEE-I/O, a standard framework for assignment of devices to VM-based CPU TEEs. This effort also includes the development of TEE Device Interface Security Protocol (TDISP [29]), an architecture for devices that support TEE-I/O. TDISP provides specifications for establishing trust between the VM-based TEE and the device (SPDM [9]), and for secure TEE-device communication (IDE [28]) and secure management of the device's lifecycle.

CPUs and devices that support TDISP are expected to be deployed in the next couple of years. Compared to application-level protocols (such as Section 6), TDISP is more efficient and transparent. CPUs that support TDISP provide hardware encryption for PCIe communication, removing the need for software encryption using explicit IVs. However, TDISP currently supports only VM-based TEEs, which brings the OS, device drivers, and other user-mode components in the TCB.

11 Conclusion

We presented ITX, a set of experimental hardware extensions for Graphcore IPUs. Our design provides application-level confidentiality and integrity for ML tasks offloaded to an untrusted cloud provider. We also presented a software architecture that removes trust from host CPUs, thereby minimizing the trusted computing base and removing dependencies on CPU TEEs. We implemented them in the GC200 IPU taped out at TSMC's 7nm node, and experimentally confirmed small performance overheads for training large models.

References

[1] Confidential computing consortium. https://confidentialcomputing.io/, 2022.

[2] Alibaba. Alibaba unveils AI chip to enhance cloud computing power. https://www.alibabacloud.com/blog/alibaba-unveils-ai-chip-to-enhance-cloud-computing-power_595409, 2022.

[3] Amazon. AWS Inferentia: High performance machine learning inference chip, custom designed by AWS. https://aws.amazon.com/machine-learning/inferentia, 2021.

[4] Amazon. Confidential Computing. https://aws.amazon.com/blogs/compute/tag/confidential-computing/, 2022.

[5] AMD. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2021.

[6] ARM. Arm Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, 2023.

[7] Rick Boivie. SecureBlue++: CPU support for secure execution. 2011.

[8] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

[9] DMTF. Security Protocol & Data Model (SPDM). https://www.dmtf.org/sites/default/files/standards/documents/DSP0274_1.3.0.pdf, 2022.

[10] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry V. Ponomarev, Nael B. Abu-Ghazaleh, and Ryan Riley. Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution. In *International Symposium on Microarchitecture*, 2014.

[11] Google. Confidential Computing. https://cloud.google.com/confidential-computing, 2022.

[12] Trusted Computing Group. Hardware requirements for a device identifier composition engine family 2.0, level 00, revision 78. https://trustedcomputinggroup.org/wp-content/uploads/Hardware-Requirements-for-Device-Identifier-Composition-Engine-r78_For-Publication.pdf, 2018.

[13] Owen S. Hofmann, Sangman M Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure applications on an untrusted operating system. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[14] Weizhe Hua, Muhammad Umar, Zhiru Zhang, and G. Edward Suh. GuardNN: Secure DNN accelerator for privacy-preserving deep learning. In *Design Automation Conference*, 2022.

[15] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbah, and Emmet Witchel. Telekine: Secure computing with cloud GPUs. In *USENIX Symposium on Networked Systems Design and Implementation*, 2020.

[16] Intel. Intel Trust Domain Extensions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2023.

[17] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity GPUs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[18] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, and et. al. In-datacenter performance analysis of a Tensor Processing Unit. In *International Symposium on Computer Architecture*, 2017.

[19] A. Khawaja, Landgraf. J, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS. In *USENIX Symposium on Operating System Design and Implementation*, 2018.

[20] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[21] Jonathan M. McCune, Ning Qu, Yanlin Li, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[22] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[23] ST Microelectronics. STM32 MCUs secure firmware install overview. https://www.st.com/resource/en/application_note/an4992-stm32-mcus-secure-firmware-install-sfi-overview-stmicroelectronics.pdf, 2022.

[24] ST Microelectronics. STM32H573 microcontroller with crypto accelerators. https://www.st.com/en/microcontrollers-microprocessors/stm32h743-753.html#documentation, 2022.

[25] NVIDIA. NVIDIA Confidential Computing. https://www.nvidia.com/en-in/data-center/solutions/confidential-computing/, 2022.

[26] Hyunyoung Oh, Kevin Nam, Seongil Jeon, Yeongpil Cho, and Yuneheung Paek. MeetGo: A trusted execution environment for remote applications on FPGA. *IEEEAccess*, 9:51313–51324, 2021.

[27] Emmanuel Owusu, Jorge Guajardo, Jonathan M. McCune, James Newsome, Adrian Perrig, and Amit Vasudevan. OASIS: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *ACM Conference on Computer and Communications Security*, 2013.

[28] PCI-SIG. Integrity and Data Encryption (IDE). https://members.pcisig.com/wg/PCI-SIG/document/15149, 2022.

[29] PCI-SIG. TEE Device Interface Security Protocol (TDISP). https://members.pcisig.com/wg/PCI-SIG/document/18268, 2022.

[30] S. Pereira, D. Cerdeira, C. Rordigues, and S. Pinto. Towards a trusted execution environment via reconfigurable FPGA. In *ArXiv*, 2021.

[31] Mark Russinovich, Manuel Costa, Cedric Fournet, David Chisnall, Antoine Delignat-Lavaud, Sylvan Clebsch, Kapil Vaswani, and Vikas Bhatia. Toward confidential cloud computing. *Communications of ACM*, 64:54–61, 2021.

[32] Richard Ta-Min, Lionel Litty, and David Lie. Splitting Interfaces: Making trust between applications and operating systems configurable. In *USENIX Symposium on Operating System Design and Implementation*, 2006.

[33] Trusted Computing Group. DICE. https://trustedcomputinggroup.org/work-groups/dice-architectures/, 2022.

[34] Stavros Volos, Kapil Vaswani, and Rordrigo Bruno. Graviton: Trusted execution environments on GPUs. In *USENIX Symposium on Operating System Design and Implementation*, 2018.

[35] Samuel Weiser and Mario Werner. SGXIO: Generic trusted I/O Path for Intel SGX. In *ACM Conference on Data and Application Security and Privacy*, 2017.

[36] Peichen Xie, Xuanle Ren, and Guangyu Sun. Customizing trusted AI accelerators for efficient privacy-preserving machine learning. In *ArXiv*, 2020.

[37] Miao Yu, Virgil D. Gligor, and Zongwei Zhou. Trusted display on untrusted commodity platforms. In *ACM Conference on Computer and Communications Security*, 2015.

[38] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. ShEF: Shielded enclaves for cloud FPGAs. In *International Conference on Architectural Support for Programing Languages and Operating Systems*, 2022.

[39] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Lixin Zhang, and Dan Meng. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *IEEE Symposium on Security and Privacy*, 2020.

A APPENDIX

A.1 Attack Vectors and Security Analysis

Table 1 summarizes the attack vectors discussed in Section 3 and, for those covered by our threat model, how ITX mitigates each of these attacks.

A.2 Firmware Provisioning and Device Certification

In this section, we describe an example process for firmware provisioning and device certificates that would be followed if ITX were to be used for IPUs in a production environment. During board manufacturing, the CCU would be provisioned with firmware followed by a board reset to harvest certificate signing requests (CSRs) generated by the execution of the primary and secondary bootloaders. The CSRs would then be used by the IPU manufacturer to issue device certificates.

Firmware Provisioning. The CCU is provisioned with firmware using the SoC's Secure Firmware Install (SFI) feature [23]. The firmware package consists of all firmware layers discussed in Section 5.1 and the configuration bytes (called `OPTION`), whose secure user memory registers are configured so that secure user memory includes only the regions onto which the secure bootloader is deployed. The firmware package is encrypted with a symmetric key, which is provisioned to a hardware security module (HSM).

The encrypted firmware package and the HSM are used by the board manufacturer to deploy CCU firmware during

Threat	Mitigation
Host (Software, Physical)	
IPU Memory Access, e.g., host software uses MMIO and PCI BARs, physical attacker tampers with on-chip memory	MMIO blacklist prevents CPU from accessing code/data in IPU; access via interfaces JTAG prohibited; IPU memory cannot be physically accessed w/o breaking into the package.
Host CPU, Memory, and PCIe bus, e.g., read/write, replay, or re-ordering of code/data in host memory or in transit, including DMA buffers, PCIe bus	Code/data are encrypted with AES-GCM with explicit IVs, and keys not shared with the host; uniqueness and integrity of IVs are ensured by trusted code executed on tiles.
IPU Binary Malleability, e.g., host replaces model encryption key or encrypted code	Bootloader computes hash of the tile binary; hash accumulated and checked against expected measurement in the job manifest. (Not implemented.)
IPU Connectivity, ICU-CCU or ICU-IPU Tampering on the development board	**no**; attacker can mount a physical attack to (1) retrieve the key(s) sent to IPU, and (2) tamper with ICU FW measurement sent to CCU
IPU-IPU Tampering	**no**; attacker can mount physical attacks against multi-IPU tasks; tamper with data sent across IPUs.
Supply Chain, Firmware	
Primary Bootloader Provisioning Tampering	The IPU manufacturer checks whether the signed bootloader manifest includes the expected nonce provisioned into the CCU primary bootloader.
Using non-genuine, known vulnerable TCB components	Firmware authorization; hardened measurement protocol outlined in Appendix A.4.
Side-channels	
IPU Memory	IPU memory access patterns cannot be observed by co-located attacker as the IPU is entirely assigned to one job at a time.
Host Memory and PCIe Bus	**no**: attacker can observe access patterns to host memory and on PCIe bus. However, these patterns do not leak much information in the BSP model, e.g., the size and number of minibatches, but not their contents.
Power and Timing	**no**: attacker can measure power consumption and/or execution time of a superstep. Similarly, this does not leak much information for typical ML tasks.

Table 1: Potential threats and how ITX mitigates them. Physical access attacks on the CCU-ICU-IPU and the IPU-IPU channels can be mitigated once the CCU is integrated on the IPU and AES-GCM is utilized to protect the IPU-IPU channels.

the manufacturing and testing. The chip tester implements a multi-stage protocol between the CCU secure bootloader and the HSM, during which the HSM authenticates the certificate issued by the CCU and wraps its firmware encryption key using the certified public key. This enables the CCU secure bootloader to decrypt the firmware package, to install the firmware, and to configure the `OPTION` bytes based on the requested configuration.

While this SFI process guarantees confidentiality of the firmware, it does not directly protect its integrity: provisioning may be subject to supply-chain attacks that would replace CCU parts provisioned using SFI with CCU parts containing malicious firmware. We extend SFI with protection against such attacks by injecting a secret known only to the IPU manufacturer into the primary bootloader. Once the CCU has been integrated onto an IPU board, a challenger can ask the primary bootloader to prove possession of the secret.

This process entails the following three steps. First, the IPU manufacturer generates a fresh secret for every batch of CCUs. The secret is injected to the primary bootloader of the CCU firmware. Second, the IPU manufacturer derives from the secret an asymmetric batch-specific bootloader manifest signing key. After deriving this key, the IPU manufacturer keeps only the public part. Third, the IPU manufacturer issues a certificate for the public bootloader manifest signing key. The certificate is signed by the IPU manufacturer Firmware certificate authority (CA). This certificate contains a batch number, and is valid till the production date of the batch.

Device Certification. In order to certify its device identity keys, the board tester resets the board and harvests the CIK and PIK CSRs generated for the board and platform identity keys, as well as the bootloader manifest. The command to harvest the bootloader manifest includes a fresh nonce, to be echoed in the signed bootloader manifest.

In response, the IPU manufacturer verifies the CSRs received by the card manufacturer and issues CIK and PIK certificates that are signed by the CIK and PIK CAs of the IPU manufacturer. In addition, the IPU manufacturer validates the bootloader manifest against the bootloader manifest signing key certificate specific to the batch to which the CCU belongs, and ensures that the nonce matches the expected one.

A.3 Firmware Updates

The CCU firmware includes a secondary bootloader and a CCE, both authenticated by the primary bootloader and possibly updated after the card has been deployed in production.

Updating the Secondary Bootloader. The secondary bootloader involves relatively complex cryptographic operations, and may need to be updated in the field. As discussed in Section 5, the platform identity key (PIK) is derived from UDS depending on the hash of the secondary bootloader. Therefore, any updates to the secondary bootloader changes the platform

identity, and PIK certificates issued by the manufacturer are no longer valid, requiring device re-certification.

Unfortunately, re-certification of a remote device by the IPU manufacturer can be a complex and lengthy operation as the manufacturer (by design) does not retain unique device keys. Thus, it requires collection of CSRs from the device, and more importantly an authentication mechanism to ensure that the manufacturer signs only PIK certificates exported from devices in the cloud provider's datacenters.

We overcome this challenge via a protocol that enables updates to the secondary bootloader without invalidating manufacturer-issued certificates.

Prior to updating the secondary bootloader (say to version Y), the cloud provider's IPU Firmware CA issues a TCB update certificate capturing the measurement of the new version of the secondary bootloader and revokes previous certificates for versions of the secondary bootloader that should no longer be deployed.

After a firmware update has been deployed, the primary bootloader generates a new CDI (CDI^Y). The secondary bootloader generates platform identity and attestation keys specific to this version of firmware (PIK^Y and AK^Y). However, the card identity key (CIK) stays the same as it does not depend on the measurement of the secondary bootloader. The PIK^Y certificate, hence, is signed by the original CIK, which has been certified by the manufacturer.

Subsequently, a remote challenger can combine the TCB update certificate with the CIK certificate originally issued by the manufacturer to verify the PIK^Y certificate is issued by the device using the original CIK, and that the measurement of the new secondary bootloader in the PIK^Y certificate matches the measurement of the secondary bootloader in the TCB update certificate.

Updating the CCE. Updates can be applied at any point without the need for any additional certification from the manufacturer. When a device boots with a new version of CCE, it generates a new attestation key with a signature over the public AK along with a hash of the CCE using the PIK. Quotes generated by the updated version of CCE firmware can be validated using a valid PIK certificate.

A.4 Measured Boot Protocol

The protocol discussed in Section 5.1 is still susceptible to advanced chosen-firmware attacks: a malicious secondary bootloader could impersonate another version of the firmware by using CIK to endorse a PIK certificate for the corresponding firmware measurement. Firmware authorization provides a strong defense against such attacks—the malicious firmware would need to be correctly signed by the IPU manufacturer to run as secondary bootloader. We can harden the boot protocol further by moving CIK and PIK generation into the primary bootloader (as shown in Figure 10) without revealing the private CIK to the secondary bootloader.

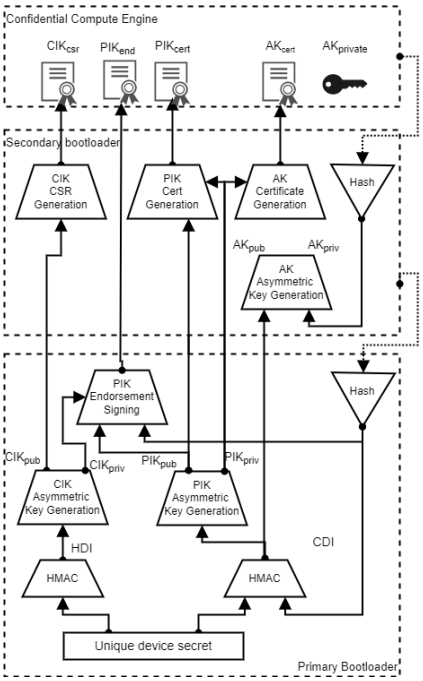

Figure 10: Hardened boot protocol that protects against bootloader impersonation attacks.

In this protocol, the primary bootloader generates CIK from UDS, and generates PIK using CDI and the measurement of the secondary bootloader. To allow a relying party (e.g., IPU manufacturer CA) to attest that the PIK was indeed generated by the primary bootloader, the primary bootloader creates a custom structure, known as PIK endorsement, containing the PIK public key along with a measurement of the secondary bootloader, and a signature over these two attributes using the CIK. The bootloader then scrubs the CIK private key and passes public CIK and PIK keys along with the private PIK and the PIK endorsement to secondary bootloader. During manufacturing, the IPU manufacturer PKI issues a PIK certificate only after validating the PIK endorsement structure. (Our prototype CCU does not implement this protocol to keep the primary bootloader simple.)

A.5 Compiled Manifests and Bootloader

Job Manifest. The compiler-generated job manifest includes all the information required by the IPU runtime and CCU to create and launch a new TEE, which will host the ML task. The manifest contains the hash digest of the application binary loaded into each IPU. It lists the synchronization points at which the IPU needs to synchronize with the host, and for each synchronization point, it keeps the following information:

- the key region identifier assigned to each stream that will be read or written following the end of synchronization (i.e., the mapping between a stream identifier j to a key region identifier);

- the ring buffer region (i.e., Tile PCI space in the ring buffer) assigned to each key region (key region definition registers);
- the part of each stream that has been mapped to the ring buffer region (stream offset);
- the set of physical key contexts to which the stream key needs to be loaded;
- the physical key context assigned to each exchange block context (exchange block context map registers); and
- the key region to which each physical key context is assigned (physical key map registers).

Secure Code Bootstrapping. The code snippet below illustrates the bootloader code that fetches the application binary frames and confirms the integrity of the IV of each frame.

```
def bootloader():
  IPU_id = get_current_ipu_id()
  tile_id = get_current_tile_id()
  num_frames = TOTAL_TILE_MEMORY / (MAX_FRAME_SIZE
       - IV_SIZE - TAG_SIZE)
  for index in range(1, num_frames):
    expected_iv = StreamType::CODE | ipu_id |
        tile_id | index
    frame = read_next_frame_from_host()
    if expected_iv != get_iv(frame):
      raise_security_exception()
    strip_iv_and_tag(frame)
  compute_hash()
```

A.6 Attestation

Cryptographic Operations. Table 2 details the keys sampled, derived, and exchanged at the start of a run in trusted mode. We rely on standard algorithms: Elliptic Curve Diffie-Hellman for establishing shared secrets, a KDF for deriving keys, and an AES-based authenticated key-wrapping scheme. These operations rely on the attestation of the manifest and runtime parameters, including all public keyshares. Each party provides its own random nonce, and the CCU combines them to deterministically derive keys for checkpoints and the final model; these keys are fresh secrets as long as one party is honest. To resume from a checkpoint saved in a previous run, the attested runtime parameters ensure that all parties agree on the epoch counter and checkpoint identifiers, and the parties provide their nonces for the previous and new run.

Remote Attestation. During TEE creation, the CCU generates an attestation report that captures security-critical attributes about the IPU and runtime configuration, including:

- the measurement of configuration registers;
- the measurement of the IPU bootloader used for loading application binaries onto IPUs;
- the measurement of the job manifest;

Key or secret		**Provider**	**CCU**
public/private keyshare for each relying party p	X_p, x_p	fresh EC share	receive X_p
encryption key for each input stream j	k_j	fresh key	unwrapped
public/private keyshare for the CCU in this run	Y, y	receive Y	fresh EC share
nonce for p in this run	$s_{p,Y}$	fresh secret	unwrapped
wrapping key for p, Y with salt $a = X^p \|\| Y \|\| M$	w_p	$\mathsf{KDF}[x_p \cdot Y](a)$	$\mathsf{KDF}[y \cdot X_p](a)$
key to load checkpoints saved by prior run Z	k_{load}	N/A	$\mathsf{KDF}[\vec{s}_{p,Z}]('ck')$
key to save checkpoints	k_{save}	N/A	$\mathsf{KDF}[\vec{s}_{p,Y}]('ck')$
key to save final model	k_m	unwrapped	$\mathsf{KDF}[\vec{s}_{p,Y}]('m')$

Table 2: Keying for a workload with manifest M between relying parties identified by their public keyshares $\vec{X}_p$ and a CCU identified by its fresh CCU public keyshare Y for this run. After attestation, an ECDH shared secret w_p is used for wrapping k_j, $s_{p,Y}$, and $s_{p,Z}$ when resuming from Z from p to the CCU, and optionally for wrapping k_m from CCU to any party p designated as a receiver of the final model. The keys used for encrypting checkpoints and the final model are derived from nonces from all relying parties, ensuring these keys are fresh (as long as one party is honest) and require agreement from all parties to be released.

- the hash digest of the attributes for this run, including:
  - the public keyshare of the CCU for this run (Y);
  - the epoch e and checkpoint counter c from which the job is restarted (if any);
  - the certificate fingerprints of all parties ($\vec{X}_p$);
  - a stream assignment, specifying a party for each input, and parties (model receivers) that receive the model key.

The host collects the attestation report, along with a CCU-issued certificate chain, which includes the AK, PIK and CIK certificates, and is rooted at the self-signed CIK certificate. These are presented to relying parties along with: the original CIK and PIK certificates, the TCB update certificates for the secondary bootloader and ICU firmware, and any intermediate CA certificates.

A relying party can verify the attestation report as follows:

1. Validate the CCU-issued certificate chain and auxiliary certificates; and check for certificate revocation.
2. Confirm that public key of the CIK certificate issued by IPU manufacturer matches the public key in the CIK certificate obtained from the CCU.
3. Confirm that any updates to the secondary bootloader and ICU firmware are rooted to a valid certificate chain. Two checks are required: (i) if there exists a TCB update certificate issued for the secondary bootloader with a

hash digest matching the hash digest in the CCU-issued PIK certificate; (ii) if there exists a TCB update certificate issued for ICU firmware with a hash digest matching the hash digest in the CCU-issued PIK certificate.

4. Review the attested manifest and attributes for this run.

Secure Key Exchange. For each run, each party p derives a fresh wrapping key w_p using its private keyshare x_p and the public keyshare of the attested CCU Y. This key is used to wrap a key package containing the streams identifiers assigned to the party and the party's key for these streams k_j, and the nonce(s) $s_{p,Y}$ for the current run (and $s_{p,Z}$ for the previous run if the current run is resuming from a checkpoint saved in run Z.) The CCU can derive the wrapping key for party p using its private keyshare y and the party's public keyshare X_p. In possession of $\vec{w}_p$, the attested CCU can unwrap the key packages of all parties, which are made available during the TEE launch stage.

The parameters of the model are encrypted using the final-model key k_m that has been derived by the CCU using the nonces obtained from all parties. The parties engage in a protocol for exchanging their nonces so they can derive the key once they possess all nonces. The CCU can additionally release the final-model key to model receivers listed in the attestation report using the wrapping key shared between itself and each model receiver.

A.7 Secure Checkpointing

Each IPU periodically checkpoints its state to enable recovery from failures. A checkpoint is created by writing the weights of the model to an output stream. The checkpoint also includes metadata, such as the current offset for all confidential data streams. These offsets are also written in plaintext, so that the IPU runtime can restart the job and resume loading of confidential data streams at the correct offset. Conversely, a checkpoint is restored by reading the weights using an input stream and resuming confidential streams from the checkpointed offsets. A checkpoint along with the job manifest and binaries suffice to resume an application from the checkpoint instead of restarting from the beginning.

In trusted mode, checkpoints are encrypted and integrity protected. In particular, tiles enforce the integrity of the process of restoring state from a previously created checkpoint. This includes protecting against attacks, such as tampering a checkpoint or loading a wrong checkpoint onto an IPU. (Guaranteeing freshness, e.g., resumption from the latest checkpoint, would involve some form of trusted persistent storage and is out of scope in this paper.)

Checkpoints are implemented using confidential streams. The code generated to read a checkpoint stream generates a sequence of expected IVs, checks that the IVs returned in the frames match the expected IV, and strips the IV and authentication tag from the frames. Conversely, the code generated to write a checkpoint stream generates a sequence of IVs and places them in the header of the frame. The IV for each frame uniquely encodes the checkpoint type, the epoch counter (incremented at each resumption), the checkpoint identifier (incremented at each saved checkpoint), the IPU and tile IDs, and the frame index. The CCU uses a separate key for each epoch; it installs the key of the epoch of the checkpoint it is resuming from, if any, and the key of the current epoch for writing all its checkpoints.

The tiles read and write checkpoints as follows:

1. Tiles obtain initial values of the epoch counter and checkpoint identifier (assigned by the CCU along with the bootloader code) from pre-determined locations in tile memory. If the epoch counter is not null, the tiles use it (with the checkpoint identifier) to compute their expected IVs and read part of their corresponding checkpoint.

2. Each tile increments their local epoch counter and start (or resume) the application.

3. At regular intervals, the tiles checkpoint their part of the state, using IVs computed from their current values, and then increment their local checkpoint identifier.

A.8 Sample Training Scenario

Figure 11 shows a sample training scenario with three parties. Given the job manifest generated by the compiler, IPU runtime, CCU, and IPU synchronize at various points where the IPU runtime populate the ring buffer with the data expected by the IPU, and the CCU loads keys to the IPU SXPs.

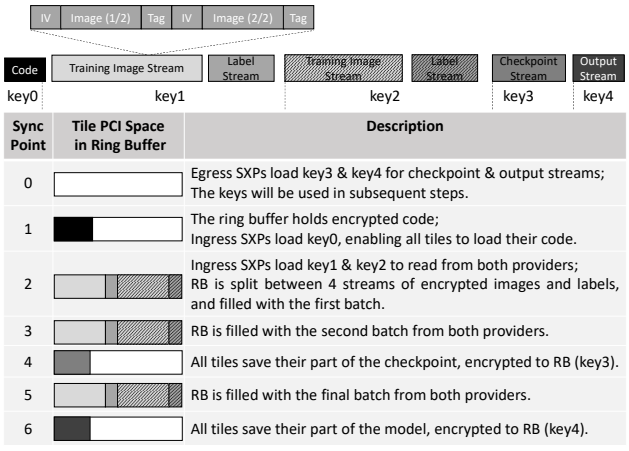

Figure 11: Sample training scenario with 3 parties: one providing model code (using key0) and the others (using key1 and key2) each providing their own streams of training images and labels; this task saves checkpoints (using key3) and a final model (using key4). The compiler emits a job manifest that indicates, for each synchronization point of the task, which part of each stream is mapped to the ring buffer (1..6) and which keys the CCU should load for ingress. The keys for egress streams are programmed in the start of the job (0).