TNPU: Supporting Trusted Execution with Tree-less Integrity Protection for Neural Processing Unit

Sunho Lee, Jungwoo Kim, Seonjin Na, Jongse Park and Jaehyuk Huh
School of Computing, KAIST
{*myshlee417, jwkim, sjna, jspark*}*@casys.kaist.ac.kr, jhhuh@kaist.ac.kr*

***Abstract*—As neural processing units (NPUs) for machine learning inference have been incorporated into a wide range of system-on-a-chips, NPUs are processing more and more mission-critical computations such as autonomous driving. With the increasing application scenarios, securing NPU operations from potential attacks has become crucial for the safety of the entire system. To address the security challenges of NPU operations, this study investigates how the trusted execution technology can be extended to harden the NPU execution by hardware supports. This paper proposes trusted NPU (TNPU) which supports trusted execution for NPUs integrated in a processor. For securing NPUs, a key performance challenge is in the encryption and integrity protection for external memory. This work proposes a novel tree-less integrity protection by exploiting the data flow semantics of DNN computation. The tree-less integrity protection maintains a version number for each tensor or sub-tensor inside the CPU enclave which drives the NPU computation. By exploiting the data flow of tensor updates, a per-tensor version number can efficiently verify the recency of the data in the tensor. The tree-less integrity protection eliminates performance losses by counter and hash cache misses, which are major performance overheads of hardware-based memory protection. Our evaluation with simulated NPUs shows that the performance overheads for trusted NPUs can be significantly reduced from the prior tree-based design, improving the performance of a single NPU by 10.0% and 7.5% on average over the prior one with two different NPU configurations. When the number of NPUs is increased to three, the performance gains further improve, achieving on average 13.3% and 8.7% improvements.**

***Keywords*-neural processing unit; trusted execution environment; memory protection;**

I. Introduction

The wide adoption of machine learning (ML) applications from edge devices to servers has been accelerating the integration of neural processing units (NPUs) in computer systems. For edge and mobile devices, recent SoCs (system-on-a-chips) commonly include one or multiple neural processing units along with conventional CPU and GPU cores in a single processor [1], [2], [3], [4]. However, the concern for the security of machine learning computing has been growing as it has been adopted for more and more mission-critical tasks including autonomous driving.

For such systems with critical machine learning tasks, the protection of NPUs is important in three aspects: 1) The results of inference tasks must be not maliciously altered by attacks on the system software and hardware. 2) Input and outputs can contain private data, and their confidentiality must be protected. 3) The trained machine learning parameters are important intellectual properties to be protected from stealing. However, the conventional software-based security measures cannot provide such protections against compromised operating systems or direct physical attacks on the system memory. A promising solution for such security requirements is to apply hardware-based trusted execution environments (TEEs) to NPU-based computing.

Supports for TEEs in conventional CPU-based computing have advanced with several available technologies such as Intel SGX, ARM TrustZone, RISC-V Keystone, and Sanctum [5], [6], [7], [8]. Intel SGX provides an isolated execution environment called *enclave*, which is protected by the CPU hardware mechanism. With the hardware-based access control and memory protection, SGX enables trusted computing even under compromised operating systems and physical attacks on on-board interconnects and external DRAM. The RISC-V Keystone project has also embraced a similar enclave-based TEE. Recent studies have extended the supports of trusted computing to GPU computation, when a discrete GPU has its own memory not directly accessible from CPU [9], [10], [11], [12].

However, although NPUs are integrated in a wide range of systems, the secure execution of machine learning on NPUs has not been investigated. In the recent SoCs with NPUs, NPUs and CPU cores share the external DRAM, posing a new challenge in the memory protection technique. This paper proposes a *trusted NPU (TNPU)* architecture which provides TEEs for *NPUs integrated in a processor*. TNPU must provide 1) memory isolation for NPU TEEs from the privileged software and other user applications, as well as 2) confidentiality and integrity protection for DRAM resident data. The memory isolation mechanism must block unauthorized accesses to the memory region of NPU TEEs. Against physical attacks, data from the NPU TEE must be encrypted before they are written to the external DRAM, and their integrity violation must be detected. For performance, the key challenge of supporting trusted NPUs is the protection of memory-resident data.

The hardware memory protection techniques designed for CPUs commonly employ counter-mode encryption with a

counter tree for freshness verification. Each cacheline-unit memory has an associated counter, and the counter value is increased whenever a writeback of the memory block occurs from CPU to DRAM. The counter-mode encryption generates a one-time-pad (OTP) from a secret key, address, and counter value for encryption. Fetching and decrypting a memory block requires the correct counter to pre-compute an OTP before data arrival. In addition, to verify the freshness of each memory block for possible replay attacks, the integrity of a counter is verified by maintaining an integrity tree (counter tree) for counters, securely keeping the root of the tree in the on-chip storage. To hide counter access latencies, metadata caches are used to store recently used counters and tree nodes of the counter tree.

However, the critical performance bottleneck for memory protection is the counter tree traversal caused by counter cache misses. Due to the cost of maintaining the counters and the tree, SGX has been limiting the protected memory region, called PRM (Processor Reserved Memory) to 128MB. In addition, recent updates for server models (scalable SGX) provide only confidentiality by encryption, while the hardware-based integrity protection is not supported [13]. Furthermore, the integration of CPUs and NPUs can exacerbate the pressure on the metadata caches, as they must share the limited capacity.

To address the memory protection overhead problem for NPUs, this paper proposes a novel semantic-based tree-less replay protection for NPU computing, which does not use the counter tree for replay protection. The computation on NPUs can be decomposed into tensor or sub-tensor computations driven by the software in CPU. Instead of using the current hardware-based version tracking with counters, our approach uses a software-oriented version validation with tensor or sub-tensor granularity. Considering the limited resources of edge devices, only a small region of physical DRAM is protected by the conventional hardware counter tree, which is used for the security metadata and enclave code similar to SGX. The rest of the memory directly accessed by NPUs is protected by counter-less encryption, and tree-less integrity protection with per-block message authentication codes (MACs) and tensor version numbers.

The proposed semantic-aware tree-less protection provides the same security level of memory protection as the prior pure hardware-based approach protecting the entire physical memory with a counter tree. While providing the same level of protection, the proposed one can eliminate counter access and validation overheads for the counter of each memory block, which significantly reduces the burden on the limited metadata caches for edge devices. As the number of NPUs increases and they share the same metadata caches, the tree-less protection can provide more scalable performance as the limited capacity of the metadata caches does not become a bottleneck.

We evaluate the trusted NPU with simulated NPU architectures. Our evaluation shows that the performance overheads for trusted NPUs can be significantly reduced from the prior tree-based design adopted from the CPU memory protection. It achieves on average 10.0% and 7.5% performance improvements over the prior design for small and large NPU configurations respectively. When the number of NPUs is three, the performance improvements are on average 13.3% and 8.7% for the two configurations.

The main contributions of the paper are as follows:

- This paper proposes a TEE architecture for NPU integrated in processors. It shows how the access control for memory used by NPUs can be supported when NPUs and CPU cores share the memory.
- This paper proposes a novel semantic-aware tree-less integrity protection for the memory used by NPUs. It can eliminate the need for tracking counters and validating the integrity of counters for the part of the memory.
- The paper showed that the tree-less protection can provide better scalability as the number of NPUs increases, as the metadata caches are no longer the performance bottleneck.

The rest of the paper is organized as follows. Section II presents the background of hardware-based trusted computing and heterogeneous computing with NPUs. Section III presents the motivation for challenges and opportunities of NPU memory protection. Section IV presents the architecture of trusted NPU, and Section V reports the experimental results with simulation. Section VI presents the related work and Section VII concludes the paper.

II. Background

A. Trusted Execution

Hardware-based trusted execution has enabled the hardware-enforced strong isolation of execution environments from malicious privileged software including the operating system [14]. In the trusted execution environments (TEEs), the trusted computing base (TCB) is reduced to the processor chip and the software running on the protected environments. With the processor enforced isolation, the privileged software cannot access the trusted execution environments, and the data and codes resident in the external memory are encrypted and integrity-verified by the hardware engine. Such TEE technologies have become available in commercial processors such as ARM TrustZone and Intel SGX (Software Guard Execution). In Intel SGX, a user-level isolated execution environment is called *enclave*, and the protected memory region, EPC (Enclave Page Cache), is secured by the hardware memory access control and memory encryption engine [5]. The recent Keystone project provides TEEs for the open-source RISC-V architecture. Keystone has a similar enclave concept as SGX, and in this paper, we will use the terminology in SGX.

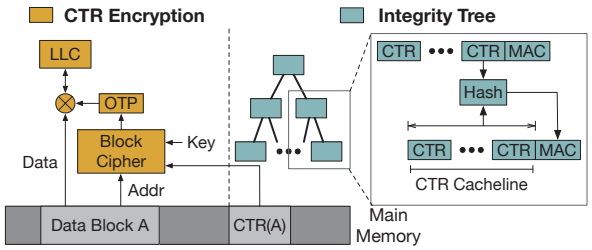


Figure 1. Counter (CTR) mode encryption and integrity tree. CTR (A) is the counter assigned for data block A.

In SGX, the page table for a user process is still maintained by the vulnerable operating system. The protected virtual address range (ELRANGE) of an enclave is mapped to the EPC region of the memory by the page table. Since the page table itself can be modified by the operating system, the hardware maintains a reverse mapping table called EPCM (Enclave Page Cache Map), which is indexed by physical address for the EPC region. The EPCM entry contains per-page security meta-data for the translation validation step. For each TLB miss for the protected memory range, the translation entry of the page table is validated by looking up the EPCM entry. The security invariant for access control is that the TLB must always contain only validated translation.

B. Memory Protection

In supporting TEEs, one of the most critical overheads is to protect external memory with a hardware encryption engine. Since the processor is the security boundary, any data coming out of the processor must be encrypted. The most popular encryption technique for hardware memory protection is counter-mode encryption. Figure 1 describes how counter-mode encryption generates ciphertexts. For each cacheline unit of memory, a dedicated counter is assigned. The counter value of a memory block is increased whenever the memory block is updated by dirty eviction from on-chip caches. The encryption of an evicted memory block is done by XOR'ing the cacheline data with a one-time-pad (OTP). The OTP is generated with a block cipher by using the secret key, address, and counter value of the memory block. Per-block counters ensure that each block is encrypted with a different OTP whenever its value changes.

In addition to the confidentiality support by encryption, the integrity of the memory block must also be protected by the hardware engine. To detect any unauthorized change of memory values, each memory block is attached with a MAC (Message Authentication Code). Since MAC does not protect against replay attacks, the integrity of per-block counter values is also validated by a counter tree. The counter for each memory block acts as a version number to validate the recency of the block. Figure 1 shows the tree organization of counters. The root node of the tree never leaves the processor chip, and the counter value fetched from the memory is validated using the counter tree [15], [16], [17], [18].

For a last-level cache miss, the corresponding counter value must be fetched before the actual ciphertext data arrives, to prepare for the OTP construction. To reduce the cost of counter fetching, a processor includes a *counter cache* that stores recently used counter values. If the required counter value does not exist in the counter cache, the counter block must be read from the memory, and its integrity must be validated by using the counter tree. Since a miss in the counter cache causes a significant delay in decrypting the data from the memory, high counter cache miss rates have been known to be one of the critical performance bottlenecks with the hardware memory encryption [12]. In addition to the counter cache, the processor also caches the internal nodes of the counter tree in a *hash cache*. The two caches for counters and hashes can be combined into a single metadata cache.

Memory encryption without integrity protection: Supporting integrity protection with hardware causes significant performance and area costs for efficiently handling counters and the counter tree. In the original SGX (*client SGX*), only 128MB of physical memory is protected by the hardware memory protection. A new model for SGX called scalable SGX no longer provides hardware-based integrity protection, but it widens the confidentiality support to the entire memory region. Scalable SGX employs AES-XTS which does not use per-block counters. Without counters and an counter tree, the metadata caches for them are not necessary. However, this new SGX memory protection does not provide data integrity and replay protection against physical attacks.

In this paper, we will adopt total memory encryption without tree-based integrity protection for NPU memory protection. However, we will add the new semantic-based version validation to support the integrity protection against physical attacks, unlike scalable SGX.

C. Heterogeneous Computing with NPUs

NPU (Neural Processing Unit) is a computing device designed to accelerate neural network computation. A typical NPU architecture uses a systolic array of processing elements (PEs) [19]. PEs consist of the computational component (multiplier and accumulator) and register file, and PEs are arranged with grid networks to transfer data. Based on the regularity of machine learning computation, NPU uses scratchpad memory (SPM) instead of block-granularity caches as an on-chip buffer. Data loading and eviction are managed by machine learning software. For common machine learning computations, a chunk of data is loaded to SPM, and computations on the loaded data are conducted. By using double buffering, the next chunk of data is transferred to SPM, while PEs are busy with computation, which parallelizes data transfer and computation [20].

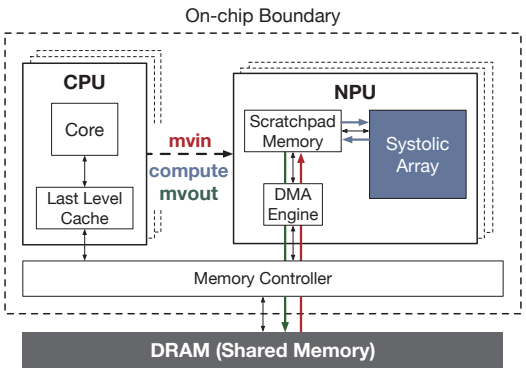


Figure 2. Integrated NPU design. The three operations, `mvin`, `mvout`, and `compute` control NPU processing.

Integrated NPUs: One of the important trends in processor designs is to incorporate multiple types of computing devices in a single chip. For example, NVIDIA Xavier has 8 CPU cores, one GPU, and two NVDLAs (NVIDIA Deep Learning Accelerator) [1]. Samsung Exynos 990 has 8 CPU cores along with a GPU, DSP, and dual-core NPU [2]. Recent Apple M1 and Tesla FSD also include one CPU, one GPU, and one or two NPUs [3], [4]. Although NPUs are also used for server-class systems using discrete designs with their own high bandwidth memory [21], this study is focused on the integrated NPUs. As machine learning tasks become common in many applications, it is expected that more and more mobile and edge processors will embrace one or more NPUs, as shown by recent the Apple M1 processor [3].

Based on the commercial designs, Figure 2 shows our target integrated NPU architecture. In this paper, we use a CPU-NPU interaction model used by Gemmini NPUs for RISC-V processors [22]. Computations on NPUs are driven by an application running in the CPU. Three key operations on NPUs are `mvin` *(move-in)*, `mvout` *(move-out)*, and `compute`. The `mvin` operation loads memory-resident data to SPM in an NPU, and the `mvout` operation writes SPM data back to the external memory. The `compute` processes the SPM-resident data.

D. Machine Learning Security

Direct system attack: Acquiring the OS privilege by various techniques for privilege escalation allows direct access to all system resources. Since the traditional memory or NPU is fully controlled by the OS, ML data and model parameters can be extracted by the OS. In addition, the integrity of computation can also be violated, producing wrong results. Another type of direct system attack is the physical attack. Unlike data center servers, edge accelerators are easily compromised by physical attacks such as bus snooping, tampering, and cold-boot attacks. A recent model architecture extraction attack uses physical or side-channel attacks to infer the ML model architecture [23], [24].

Table I
ATTACKS IN ML: CONFIDENTIALITY (C) AND INTEGRITY (I)

Attack	Weak Spot	C/I	Ours
Malicious System Software	Access Control	C, I	O
Bus snooping	Memory Protection	C	O
Tampering	Memory Protection	I	O
Cold-boot Attack	Memory protection	C	O
Model Extraction	Address Trace	C	X
Inversion Attack	Algorithm	C	X
Poisoning Attack	Algorithm	I	X

*O: supported protection, X: unsupported protection

Adversarial attack: Attacking the vulnerability of the algorithm itself, rather than the existing system, is called *adversarial attack* [25]. The model inversion and extraction attacks extract the training data and model architecture based on the result of designed queries [26], [27]. The poisoning attack maliciously manipulates training data. The evasion attack adds noises to input images to cause incorrect classification [28].

E. Threat Model

The trusted computing base (TCB) is the SoC containing CPU and NPU along with the application codes running in the TEEs. The study also assume that the data generation by sensors and transfer to the CPU TEE from the sensors are secure as discussed by the prior work [29], [30]. Privileged software such as OS and hypervisor can be compromised by attackers. In addition, attackers can launch physical attacks such as bus-snooping or modification of external DRAM contents [16], [31]. However, this paper assumes that side-channel attacks must be addressed by an orthogonal separate measure. In addition, the availability of services is not guaranteed, since the operating system controls the scheduling of CPU and NPU.

Table I shows potential attacks on ML systems, and the scope of the problem this study is addressing. This study protects ML computation on NPUs from direct confidentiality and integrity attacks by privileged software or physical attacks. Model architecture extraction via side-channel or bus snooping [23] is beyond our scope. The protection for such model extraction requires obliviousness support for hiding data transfer patterns. In addition, adversarial attacks on the ML algorithms are also out-of-scope in this study.

III. MOTIVATION

A. Trusted NPU Computing

NPUs process essential machine learning tasks, which are part of the computation cycle from the generation of input data to the final outcome. In such ML-oriented systems, the computation cycle involves sensor devices, CPUs, NPUs, and often remote clouds to complete the processing of data. To support the end-to-end security from data collection to output, the entire flow must be protected. Figure 3 shows an example of an edge system that collects data from its

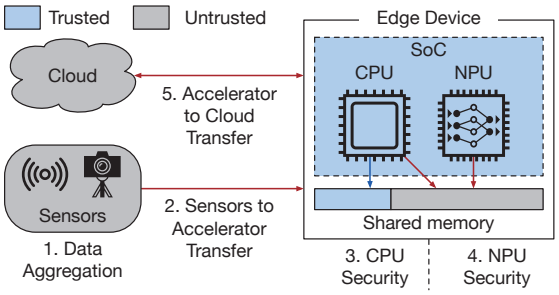


Figure 3. End-to-end security for edge ML computing from sensor data collection to NPU processing

sensors and processes the sensor data with NPUs. For the security of the entire flow, each step must be designed to support confidentiality and integrity.

Sensor data encryption and integrity protection: To securely collect data, sensors encrypt the data and securely transfer them to the CPU memory [29], [32]. The Waspmote sensor device which is used in the Libelium IoT project includes the AES encryption engine for confidentiality [29]. In addition, it can use the checksum method by using a hash algorithm such as MD5 or SHA to protect the data from integrity attacks. For sensor data, in addition to the encryption, this paper assumes that the integrity of the communication between sensors and CPUs is protected via any type of integrity protection scheme such as message authentication code (MAC).

Trusted CPU pre-processing: With the securely transferred data from sensors, CPU must execute the pre-processing of the input data in an enclave [30]. Once the pre-processing is completed, the generated input must be securely passed to the NPU. In the integrated NPU architecture, the CPU enclave uses the memory region shared with the NPU for the data transfer. In our proposed architecture, the shared memory between CPU and NPU is protected by hardware encryption and semantic-based integrity protection.

Trusted NPU processing: NPU must process ML tasks for given inputs in a trusted execution mode. The output is shared with the CPU enclave and post-processed by the CPU code in the enclave. The CPU enclave can initiate further ML tasks, or order the actuation of physical devices.

In the end-to-end security supports from sensors to NPUs, this study is focused on trusted NPU processing. Although we used the example scenarios with an edge device, this study is applicable for any integrated NPU architecture with vulnerable external DRAM.

B. Memory Protection for NPUs

For supporting TEEs, one of the key performance overheads is to provide the confidentiality and integrity of memory-resident data [15], [16]. In this section, we investigate the performance impact of memory protection for NPUs. For hardware-based memory protection, we evaluate the counter-mode encryption and counter trees for confidentiality and integrity supports. The design is based on the protection technique for CPU memory [15], [16]. We evaluate two NPU configurations with different capabilities, `Small NPU`, and `Large NPU`. The details of NPU configurations are presented in Table II. We evaluate 14 machine learning tasks. The NPUs use a 4KB counter cache and a 4KB hash cache. Each 64B counter cache entry has 64 counters using the split-counter mechanism (SC-64), and the arity of the tree is 64 [33]. More detailed methodology is presented in Section V. In the baseline design, the entire DRAM space is encrypted and protected by the hardware encryption and counter tree.

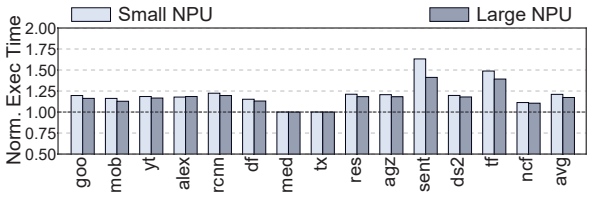


Figure 4. Execution times normalized to unsecure runs with two NPU configurations.

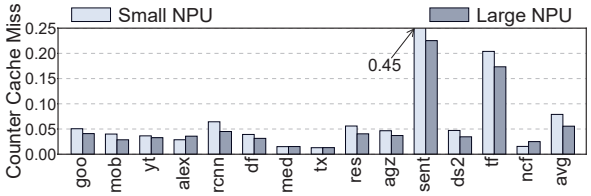


Figure 5. Counter cache miss rates of the conventional tree-based memory protection with two NPU configurations.

Performance overheads: We first evaluate the potential performance degradation by supporting hardware-based memory protection for NPU. Figure 4 presents the execution times with the two NPU types, normalized to those without any memory protection (unsecure NPUs). As shown in the figure, with a naïve adoption of CPU-oriented memory protection, the performance degradation is severe. For `sent` and `tf`, `Small NPU` can suffer from 63.2% and 48.8% execution time increases. On average, the performance degradation for 14 workloads is 21.1% and 17.3% for the `Small` and `Large NPUs`, respectively. Considering NPUs are used for performance acceleration, the security features must incur minimal performance degradation. The result shows that the CPU-oriented technique needs to be further optimized for NPUs.

Counter cache misses: The performance of tree-based integrity protection depends on the counter cache misses. When a counter cache miss happens, the integrity tree recursively verifies the counter value and missed internal tree nodes, adding significant costs. Figure 5 presents counter cache miss rates in the evaluated models. As each counter cache block can contain 64 entries with the 64-arity split-counter scheme, the miss rates look low for the workloads except for `sent` and `tf`. Although the counter cache has

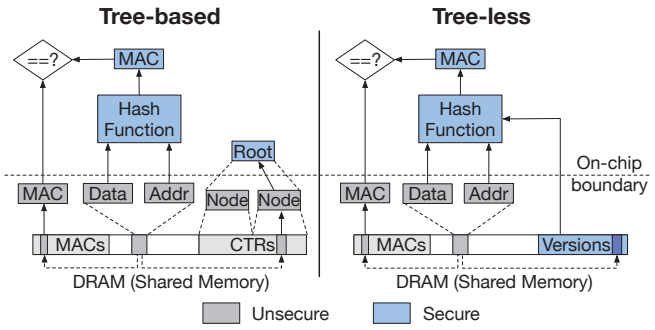


Figure 6. Tree-based vs tree-less memory protection

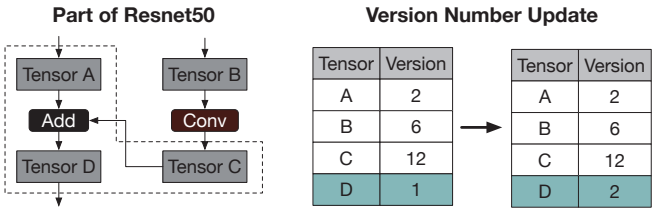


Figure 7. An example of version number updates in part of ResNet50: `Add` between tensors A and D updates the version number of tensor D.

high spatial localities for those workloads, its temporal locality is low because of streaming access to tensors. Even for the seemingly low miss rates, the performance impact is high as each counter cache miss incurs a high performance penalty. `Sent` and `tf` have low spatial localities due to their fine-grained access patterns.

Tree-based vs tree-less integrity protection: The conventional integrity protection uses a tree-based verification for per-block counters. Each memory block has an associated counter as a version number and MAC. The left part of Figure 6 shows the conventional memory protection with a tree. For the entire counters covering the total physical memory, a tree is constructed. The root of the tree is always securely kept. To reduce the cost of the tree validation, on-chip storage is required for caching counters and tree nodes.

Our approach is a semantic-aware tree-less verification, as shown in the right part of Figure 6. NPU computation can be decomposed into the data flow of tensors. For each tensor or sub-tensor (tile) which is updated as a single unit, one version number is assigned. The version numbers are stored in the CPU-side enclave memory, which uses a small fully-protected memory region in the memory. The NPU software running on the CPU is responsible for tracking the version number for each `mvin` or `mvout` operation. Since the tensor data flow is statically known, the software can manage the version numbers efficiently, unlike the per-block counters in the prior hardware-based approach.

C. Tensor-based Computation

Our tree-less replay protection relies on the tensor-based data flow of NPU computing. Figure 7 shows part of the data flow from ResNet50. The update of each tensor can

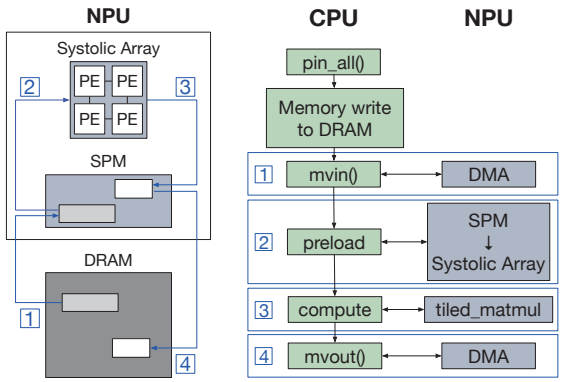


Figure 8. NPU execution model driven by the CPU-side software. It shows the data movement between the SPM and external memory controlled by `mvin` and `mvout` instructions. The `preload` instruction loads necessary parameters in the systolic array from SPM, and `compute` initiates computation.

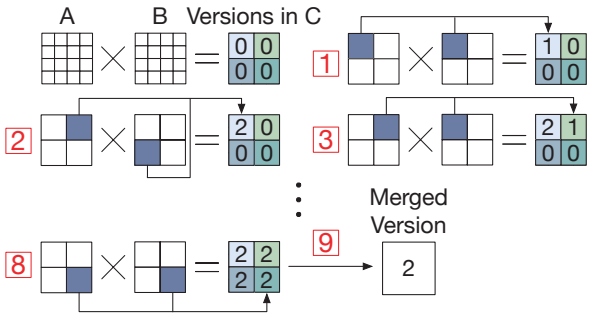


Figure 9. A example of version number management for 2×2 tiled matrix multiplication for matrices with 4×4 elements. The number in each tile is its version number, and in Step 9, they are merged into a single version number of the output matrix.

be identified in the data flow, and thus the software can increment the version number of each tensor. When a tensor is read from the memory to NPU SPM, its version number is checked whether the tensor is the latest version. The correct version number is securely stored in the CPU-side enclave memory, which is fully protected by the hardware-based integrity protection.

Figure 8 shows how the actual software handles the data movement. In the execution model, `mvin` and `mvout` are the only operations for transferring data between SPM and the memory. For `mvout`, the software updates the version number in the fully protected enclave memory, and passes the version number to the NPU hardware component to add it in the MACs of memory blocks for the tensor. For `mvin`, the software supplies the correct version number to the MAC verification unit of NPUs.

One complication is the tiling of tensors when necessary tensors cannot fit in SPM. Figure 9 describes this situation when a matrix multiplication (A x B = C) is computed. In the case, two tensors are statically decomposed into tiles. The unit for `mvin` and `mvout` is a tile. The software assigns

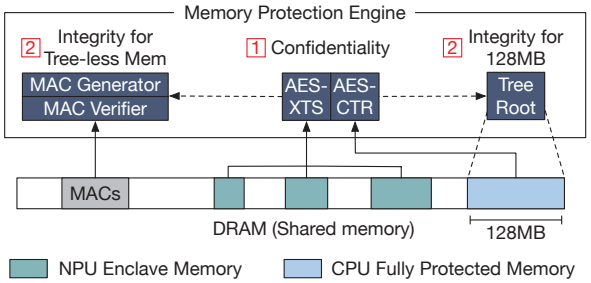


Figure 10. DRAM space and memory protection. The CPU fully protected memory uses the conventional tree-based memory protection, but the memory regions used by NPUs are protected by a new tree-less protection. A separate region is reserved for MACs.

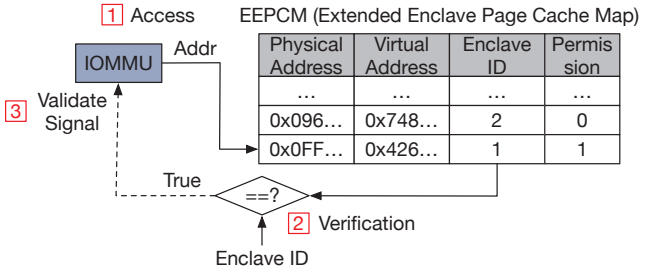


Figure 11. Translation validation for an IOMMU miss. Translation entry is verified by looking up EEPCM for the page.

a different version number for each tile. When tile-based computations for a tensor are completed, each tile has the same version number since the number of updates for tiles in the same tensor is equal for matrix multiplication. After merging, a single version number is used until tiling is necessary again in later computation.

IV. Architecture

A. Overview

TNPU (Trusted NPU) architecture provides hardware-based trusted execution for NPU computation. To support such *NPU TEE*, it extends the CPU-side trusted execution environments to NPUs, isolating the context for NPU execution. This study assumes that the baseline CPU TEE provides a similar mechanism to SGX enclaves, and we call TEEs enclaves. However, the idea is not strictly tied to SGX enclaves. The processor maintains a fixed protected memory region (PRM in SGX) in part of the memory with a small capacity of 128MB. All security metadata are stored in the protected region. There are three main components for TNPU.

Protected NPU driver: The privileged software such as the OS should be allowed to control the NPU. As proposed by the prior GPU protection [10], the NPU driver which controls NPUs must be running in a CPU driver enclave. The OS can only send requests to the protected driver. Note that supporting such driver enclaves for controlling I/O devices requires the extension of the current SGX to access MMIO regions, as suggested by the prior work [10].

Memory access control: The NPU context must be protected from OS or other user applications. Accesses to the NPU address space must be verified by a similar way to the validation step of SGX. For such validation, the processor maintains an inverse page map, called Extended EPCM (EEPCM) which is an extension of SGX EPCM to cover the entire physical memory. EEPCM is a flat inverse map indexed by physical page address. Each entry contains the security metadata for the page. For a TLB miss, EEPCM is consulted to validate the page table entry maintained by the OS.

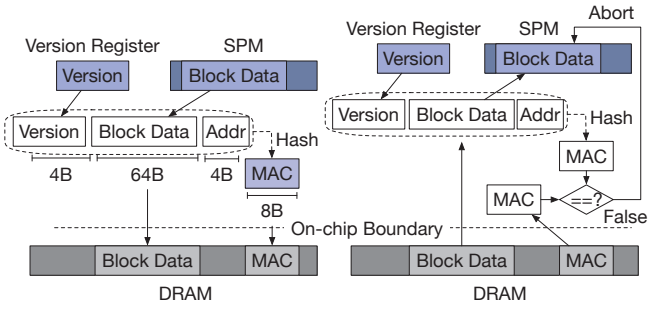
Memory encryption and integrity protection: Since physical DRAM is shared by CPU and NPU, the part of DRAM used by the NPU must be encrypted and integrity protected. As NPUs can use a large amount of memory, their memory pages are not mapped to the fixed protected region for CPU enclaves. Instead, the NPU memory is allocated during the enclave initialization, and protected by our tree-less memory protection mechanism.

NPUs are controlled by the NPU driver enclave, and the user application sends a request to the NPU driver to get permission to use the NPU. Once an NPU context is assigned to a CPU enclave, the CPU enclave shares the non-EPC part of its memory space with the NPU context and interacts with the control commands such as `mvin`, `mvout`, and `compute`. Both MMU and IOMMU consult EEPCM for validation during TLB miss handling.

In our system, in the DRAM space, 128MB fixed region is protected by the conventional tree-based scheme similar to SGX PRM. We will call this region *fully protected region*. The security metadata such as EEPCM are stored in the region, and each CPU enclave allocates the security-critical data in the region. The rest of the memory region accessed by NPUs is protected by the new tree-less protection. Note that a CPU enclave can access the tree-less region with new memory instructions, and in that way, the CPU enclave and NPU computation can share the memory. A separate fixed region is used to store MACs of the entire DRAM space. Figure 10 presents the memory space and protection mechanism.

B. Access Control for NPU Context

The memory of a trusted NPU context must be protected from any unauthorized access from other contexts or privileged software. Since the memory is shared between CPU and NPU, such access validation must be maintained both by CPU and NPU. For the entire physical memory, EEPCM contains per-page security metadata, such as owner enclave ID, virtual page number, physical page number, permission, etc. For a TLB miss in CPU MMU or NPU IOMMU, the EEPCM entry is accessed to validate the translation and permission. The page table itself is maintained by the OS, so the mapping information can be updated arbitrarily.



(a) Generation of the MAC. (b) Verification of the MAC.

Figure 12. Tree-less memory protection. (a) shows the embedding of a version number when MAC for a block is generated during `mvout`, and (b) shows the MAC verification including the version number during `mvin`.

Therefore, the system-wide page map must validate any translation attempts to the NPU memory both from CPU and NPU, since any privileged CPU process attempts to access the protected NPU memory page by changing its page table. Figure 11 shows how the access control is done with EEPCM when an IOMMU miss occurs.

EEPCM is stored in EPC pages (residing in the fully protected region), so it is protected from the OS or physical attacks. In the virtual address space of an NPU context, a contiguous region is designated as the protected address range (NELRANGE). During the initialization of an NPU context, NELRANGE is set, and pages are added to the virtual address range. When a physical page is added, the corresponding entry of EEPCM must be updated to record the ownership of the page. However, the memory pages accessible by NPUs are not mapped to the fully-protected region but assigned to the region protected by the tree-less protection. The next subsection explains how the memory pages of an NPU enclave are protected by the hardware encryption engine.

C. Tree-less Memory Protection

In the proposed architecture, the entire DRAM, except for the fully protected region, is encrypted with AES-XTS similar to Intel Total Memory Encryption [13]. The hardware-based encryption provides confidentiality for all memory blocks accessed by CPU, GPU, and NPUs. The memory encryption does not use per-block counters and thus it does not require counter and hash caches for performance.

In addition to the memory encryption, we add MAC for each memory block for integrity. The per-block MAC is used to authenticate the memory block contents. However, MAC itself cannot detect a replay attack that replaces the latest version of a data block and its MAC with an old version of them. With semantic-aware tree-less replay protection, the software tracks and assigns a version number for each `mvin` and `mvout` operation. Figure 12 (a) shows the generation of the MAC for a memory block when the memory block is written from SPM to the external memory during an `mvout` operation. For each 64B memory block, the content of the block, its block address, and version number supplied by the `mvout` operation are used to generate an 8B MAC. MAC is written to the MAC region in the memory corresponding to the block address. Note that MAC generation and verification can be selectively turned on or off, depending on the page status set in EEPCM.

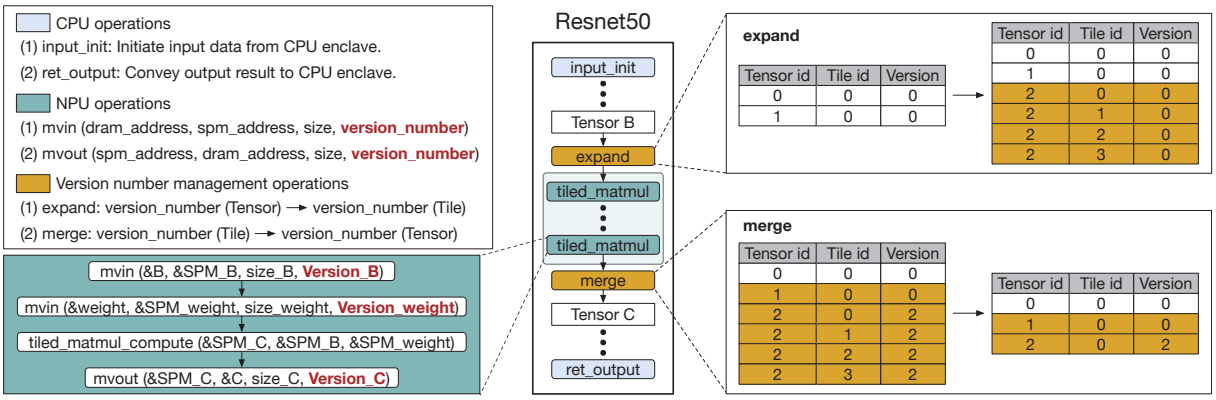
Figure 12 (b) shows the verification of data when it is fetched from the memory to the on-chip SPM. For an `mvin` operation, the CPU-side software provides the expected version number. For fetched data, the block content, address, and the expected version number are used to generate the correct MAC. If it does not match with the stored MAC, at least one of the three inputs of MAC generation (block content, address, and expected version number) is invalid. By setting and verifying a version number for each memory block, a replay attack can be detected. The version number is assigned to a tensor or a tile, so the memory blocks belonging to the same tensor or tile use the same version number.

A small region of memory (fully-protected region) is still protected by the hardware-based integrity tree, but NPUs do not access the memory directly. The fully-protected memory region is used to support CPU enclaves, as used by SGX. The application software running in the CPU is protected inside an enclave. Therefore, the codes and CPU-only data including the version numbers of all tensors are stored in the fully-protected memory region. Compared to the main NPU data for tensors, the required data in the fully-protected memory region is small.

Hardware changes: To support the tree-less replay protection, NPU APIs are extended. For each `mvin` or `mvout` operation, a version number is added as a parameter. For `mvin`, the version number is the expected version number for the tensor data, and for `mvout`, the version number is used to generate MACs. The user application running in a CPU enclave is responsible for managing version numbers for tensors. When it executes `mvin` or `mvout`, it supplies an appropriate version number, which is stored in the fully protected memory region.

The hardware mechanism is straightforward. For `mvin`, the DMA engine will read data from the memory. The expected version number is passed to the MAC verifier. Since a single `mvin` can generate many 64B block accesses, all blocks are verified by the MAC verifier with the same supplied version number. For `mvout`, the MAC generator uses the supplied version number to create new MACs. MACs and data blocks are written to the external memory via the DMA engine.

CPU-side accesses to tensors: A CPU enclave driving NPUs also needs to access the tensor region with the tree-less memory protection during the initialization of the tensors and the finalization of NPU computation. To support the CPU-side access to the region covered by the tree-less



(a) Extended code with version number.

(b) Version number management.

Figure 13. Software changes and version number management operations for a ResNet50 layer. The code is extended with version numbers in `mvin` and `mvout`. In addition, the version number can be expanded and merged for tiled operations. The table for version numbers is stored in the fully protected memory.

protection, special memory read and write instructions are added. Memory accesses by the new instructions from CPU must follow the encryption and tree-less integrity protection scheme used for tensors, so that the data written by CPU can be read correctly by the NPU, and vice versa. To do that, the new memory instructions have a version number in their parameters, which are similar to `mvin` and `mvout`. A difference from `mvin` and `mvout` is that the granularity of access is not a tensor or tile, but a block or word.

One important restriction for the new memory operations by a CPU enclave is that they need to bypass on-chip CPU caches, since the caches cannot store version numbers. If caching is allowed for the data, CPU caches must carry the version number for each block, incurring area overheads. In our design, we opted for forcing uncacheable writes with version numbers when CPU writes to tensor data, since the initialization and CPU-side update of tensors are infrequent. In addition, the streaming nature of common tensor accesses by the CPU software (for example, initialization) does not efficiently utilize the caches due to its low temporal locality. To do that, the tensor memory must be marked as uncacheable pages, as supported by x86. Since a MAC is created for each block unit of data, two small block buffers (64B) for read and write are added to temporarily store data for each core. For efficiency, two new instructions (`ts_read_block` and `ts_write_block`) fetch and write tensor data between the memory and the buffer at 64B block granularity. An additional `ts_read_byte` reads specified bytes from the read buffer filled by `ts_read_block`. A new instruction `ts_write_byte` fills the write buffer, which will be flushed to the memory by `ts_write_block`.

D. Tracking Version Number in SW

The compiler for NPUs and library writers add the code for tracking version numbers. Since the data flow is statically analyzed in the NPU software, the extra effort is minor and it can be automatically inserted by the compiler. We use a ResNet50 example from the Gemmini project to show the required changes for the software. Figure 13 (a) shows part of the code in ResNet50. In this example, `mvin` and `mvout` are changed to new APIs with a version number. For the CPU-side access for *initialization*, `ts_write_byte` and `ts_write_block` operations are used.

Version number management: Version numbers are stored in a table whose indices are *tensor_id* and *tile_id*. The table for each NPU context is stored in the fully protected region. To initiate `mvin`, the corresponding version number must be read from the table, and passed as the instruction parameter. When `mvout` is initiated, the CPU-side application code updates the version number in the table, and the updated value is passed as the instruction parameter.

Figure 13 (b) shows the version number management in the fully protected region. As shown in Figure 9, since a tile-unit version number must be maintained within a layer operation, the tensor-unit version table is expanded. While the input tensor uses a tensor-unit version number as it is not updated, the version number of the output tensor is divided into a set of tile-unit version numbers. After all tile operations in a layer are completed, they are merged into one entry as they have the same values in the end of the operations. According to our analysis, the storage for version numbers is only 1.3KB on average and up to 7.5KB with the `tf` which is the most memory-intensive workload.

E. Initialization and Attestation

Initialization: With the integrated NPU with CPU, the secure initialization is done by the CPU TEE mechanism. The secure NPU execution always has an associated CPU enclave. The CPU enclave initiates the secure NPU computation. During the initialization, the CPU enclave allocates its own EPC memory page, and it also allocates non-EPC

memory for NPUs, which is access-controlled by hardware, but protected by tree-less protection. System booting does not need to be secure, since SGX and Keystone provide dynamic enclave creation.

Remote attestation: Since NPU is integrated into the processor, the remote attestation is provided by the CPU-side enclave attestation mechanism. The processor chip itself including CPU and NPU is validated by the CPU-side (processor-wide) validation protocol similar to the SGX remote attestation. The binary for the CPU enclave software which includes NPU instructions is attested with the CPU enclave attestation. The NPU driver enclave also needs to be attested.

V. Evaluation

A. Methodology

Simulation infrastructure: We use an in-house cycle-level simulator, which is built upon the widely-used open-source, SCALE-Sim simulator [34]. As SCALE-Sim provides layer-wise modeling without the support for inter-layer connections, we added the support to our simulator. The simulator contains the following NPU characteristics: 1) scratchpad memory as on-chip memory, 2) double buffering for data transfer, and 3) systolic array to utilize PE efficiently. To reflect data transfer overheads between NPU and off-chip memory, we use a simple memory bandwidth model, which limits the maximum bandwidth. We assume 100 cycles for DRAM latency based on the prior work [35]. The simulator can process convolution, fully-connected, matrix-matrix multiplication, and matrix-vector multiplication computations. The simulated NPU has a hardware *im2col* block, and thus it can perform *im2col* on the fly.

We augmented the necessary components for TNPU to our simulator. For the baseline, we model counter-mode encryption for confidentiality and MAC with a 64-arity split-counter tree (SC-64) for integrity protection. The entire memory of the baseline system is protected by the counter-mode encryption and the counter tree. Creating an OTP takes 10 cycles with 1 cycle for XOR operation in counter-mode encryption. We use a 4KB counter cache and a 4KB hash cache. As SGX supports an 8-arity tree with a total 64KB counter and hash cache, our setup with our 64-arity tree has a similar data coverage to SGX. To reduce the overheads of MAC accesses, an 8KB MAC cache is used to keep recently accessed MACs. A 64B MAC block contains 8 MACs and the MAC cache reduces MAC read and write traffic by exploiting the locality [36].

To simulate our tree-less mechanism, we reduce the fully protected area to 128MB which includes the security metadata such as version numbers. Except for the 128MB region, the rest of the memory uses AES-XTS encoding and HMAC with a version number. As AES-XTS requires 10 cycles for two parallel AES and 3 cycles for two addition and one XOR operation, we simulate a latency of 13 cycles for confidentiality. To consider the cost of storing version numbers in the fully protected memory, access requests to the fully protected memory are generated, when the version number is needed. TNPU also use an 8KB MAC cache.

Table II
Simulation environments [2], [37]

Small NPU (Samsung Exynos 990)	
PE	32 x 32, systolic array
Bandwidth	11 GB/s with 4 channels
Frequency	2.75 GHz both processor/memory
Scratchpad Memory	480KB in total
Precision	Float16, 2B per element
Large NPU (ARM Ethos N77)	
PE	45 x 45, systolic array
Bandwidth	22 GB/s with 4 channels
Frequency	1 GHz both processor/memory
Scratchpad Memory	1MB in total
Precision	Float16, 2B per element

Table III
Evaluated benchmark models

Model	Mem Footprint
Googlenet (`goo`), Mobilenet (`mob`)	15.2MB, 11.4MB
Yolo-tiny (`yt`), Alexnet (`alex`)	18.9MB, 11.7MB
FasterRCNN (`rcnn`), DeepFace (`df`)	29.3MB, 2.2MB
Resnet50 (`res`), MelodyExtractionDetection (`med`)	41.4MB, 34.8MB
Text-generation (`tx`), AlphaGoZero (`agz`)	21.7MB, 2.2MB
Sentimental-seqCNN (`sent`), DeepSpeech2 (`ds2`)	58.8MB, 15.6MB
Transformer (`tf`), NCF-recommendation (`ncf`)	75.6MB, 11.6MB

Hardware configuration: As this paper targets edge-level SoC platforms with integrated NPUs, we refer to the specifications of SoCs in the Samsung Exynos 990 and ARM Ethos N77, and use the NPU architecture configurations of Exynos 990 (`Small NPU`) and Ethos N77 (`Large NPU`) [2], [37]. As shown in Table II, `Small NPU` is equipped with 1024 (32×32) PEs with 11 GB/s bandwidth, and `Large NPU` comes with 2025 (45×45) PEs with 22 GB/s bandwidth. The `Small NPU` and `Large NPU` configurations also have 480KB and 1MB of scratchpad memories, respectively. Both configurations employ 16-bit data width.

Benchmarks: Table III lists the evaluated benchmarks, which are selected from a wide range of machine learning algorithm domains that include computer vision, speech recognition, natural language processing, and computer game [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53]. Each benchmark has compute-intensive and memory-intensive layers. The convolution layer is the most compute-intensive layer and the embedding layer is the most memory-intensive layer. Therefore, `sent` and `tf` which have embedding layers are more memory intensive than the others. The column, *Mem Footprint*, reports the total size of memory footprint requirement, which includes ifmap, ofmap, and model parameters.

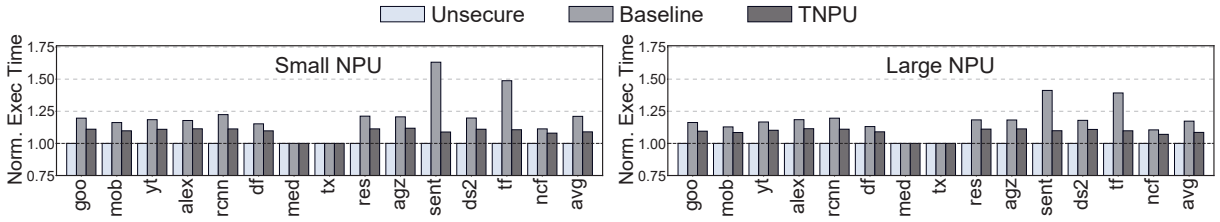


Figure 14. Execution times for unsecure, baseline, and TNPU, normalized to the unsecure configuration.

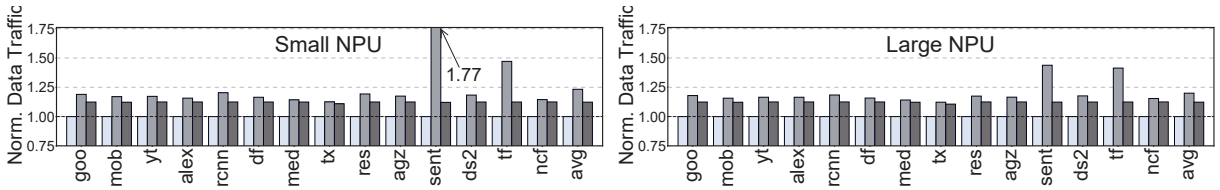


Figure 15. Data traffic volume normalized to the unsecure configuration.

B. Performance Improvement

Figure 14 presents the execution times of our tree-less protection. Each result is normalized to the unsecure configuration (`Unsecure`), which represents the execution of NPU without any memory protection scheme. The second bars (`Baseline`) of each model show the execution times with the conventional tree-based memory protection. The execution times of the proposed architecture are presented at the rightmost bars (`TNPU`). The elimination of the counter traffic and recursive integrity-tree processing reduces the performance overheads significantly from `Baseline` to `TNPU`.

For `Small NPU`, the tree-less protection (TNPU) improves the performance of the baseline by 10.0% on average. Compared to the unsecure run, the tree-less protection only incurs 9.0% performance degradation, while the baseline has 21.1% deterioration. For `Large NPU`, `TNPU` has 7.5% performance improvement over the baseline, and TNPU has 8.6% performance degradation for memory protection over the unsecure run.

The amount of data traffic partly explains the performance improvement. Figure 15 presents the memory traffic from the simulation, which matches the execution time trends. Confidentiality support and integrity protection generate significant extra memory traffics in the baseline. In this result, we can infer the volume of security metadata traffic based on the difference from the unsecure run. In `Small NPU`, the amount of extra data traffic is reduced from 23.3% in the baseline to 12.3% in TNPU, compared to the unsecure run. `Large NPU` also shows similar trends.

Memory-intensive benchmarks, `sent` and `tf`, which contain embedding layers, have high performance degradations with the baseline, 52.2% and 44.0% respectively. Since these embedding layers have multiple large one-hot vectors that create many tiles, these tiles cause relatively sparse memory accesses with low spatial locality, which prevents the efficient caching of counters. TNPU reduces the performance degradation to 9.4% and 10.2% for `sent` and `tf`.

The remaining overhead compared to the unsecure configuration is due to the extra memory traffic generated for MAC reads and writes. 8B MAC is used for each 64B memory block, and thus $\frac{8B}{64B} = 12.5\%$ of extra traffic occur for the MAC accesses. Although the MAC cache reduces the memory traffic, MAC accesses still cause non-negligible overheads for secure DNN acceleration.

C. Scalability

As the number of NPUs sharing the counter and hash caches increases, the performance of the baseline can further degrade [54]. In this subsection, we perform the scalability study by increasing the number of NPUs from one to three. Figure 16 presents the execution times of the baseline and TNPU, which are normalized to the unsecure run for each NPU count. For example, for 3 NPUs, the baseline execution time is normalized to the unsecure run with the same 3 NPUs. In multi-NPU runs, each NPU has a separate IOMMU while the memory controller and security engine are shared, sharing memory bandwidth and the capacity of metadata caches. The same inference models are running in each NPU.

As shown in the Figure 16, as the number of NPUs increases, the performance gap between the baseline and TNPU increases. On average, for `small NPU`, TNPU improves the baseline by 10.0% for 1 NPU, and the improvements increase to 12.9%, and 13.3% for 2, and 3 NPUs, respectively. The performance gain by TNPU increases over the baseline since counter and hash cache misses increase with the increasing NPU count in the baseline.

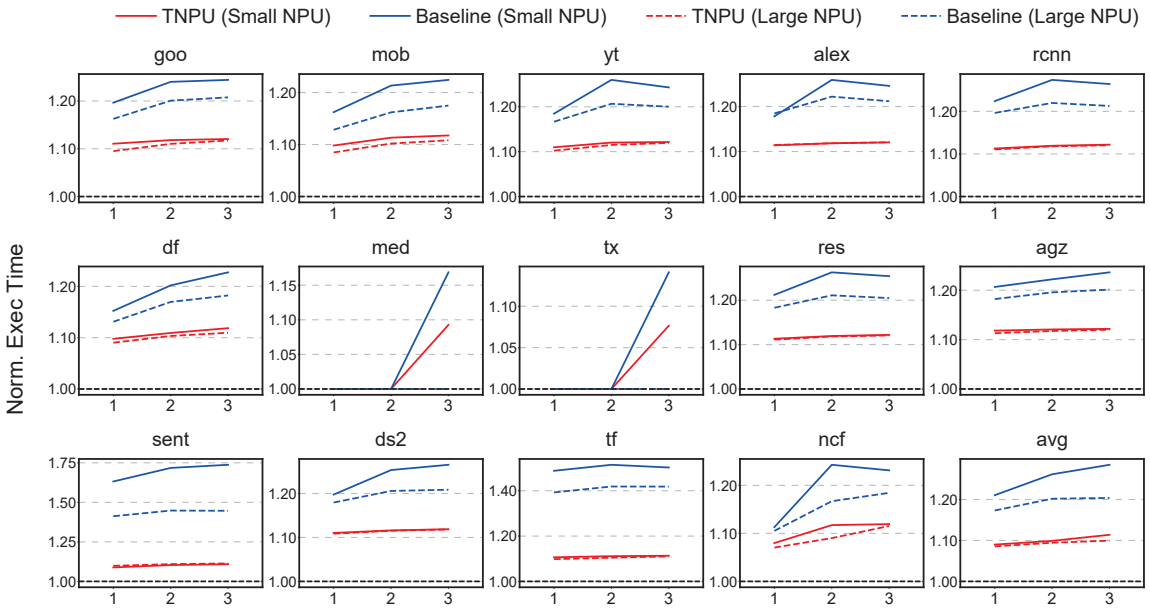


Figure 16. Execution times when one to three NPUs are used. They are normalized to the unsecure run of the same NPU count.

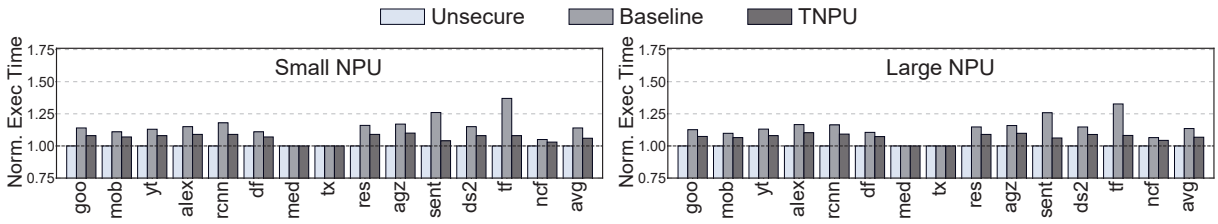


Figure 17. End-to-end execution time of unsecure, baseline, and TNPU, normalized to the unsecure configuration.

The performance gap between the baseline and TNPU is smaller with `Large NPU` than `Small NPU`. It is due to the bigger SPM capacity of `Large NPU`, which reduces counter cache access. In `med` and `tx` with `small NPU`, the baseline and TNPU do not have performance degradation for 1 NPU, but more NPUs incur overheads due to counter cache misses and metadata traffic. The performance gap between the baseline and TNPU slightly reduced for `yt`, `alex`, `rcnn`, `res`, and `ncf` with 3 NPUs compared to 2 NPUs in `Small NPU`. It is because the limited memory bandwidth becomes the primary bottleneck for those cases.

D. End-to-End Performance

To evaluate the entire execution latency including the CPU-side operation, we combine the Gemmini model [22] with our simulator. With the evaluation model with CPU and NPU interaction, the end-to-end latency is the time delay from the completion of data transfer from the sensor, to the return of the inference output from NPU to CPU. Except for the NPU computation time, the dominant extra latency is for the initial transfer of model parameters to the memory region of the NPU context. As an NPU context commonly processes many inference requests for a loaded model, the amortized cost of the initialization can be significantly reduced. However, in this result, for conservative evaluation, the parameter initialization time is added for processing a single request. Figure 17 shows the end-to-end execution times for three configurations. While the baseline has 14.1% and 12.6% performance overheads on average over the unsecure run for `Small` and `Large NPUs`, TNPU has only 6.4% and 5.6% overheads over the unsecure run. Therefore, even for the end-to-end evaluation, TNPU improves the performance of the baseline by 8.2% and 6.2% for the two NPU configurations.

E. Hardware Overhead

For TNPU, the memory encryption engine for tree-less protection is added as an extra hardware component. The memory encryption engine consists of AES-XTS and HMAC hardwares. They need three AES-engines and 512B small additional storage for a tweak value and intermediate values. In addition, we use a 8KB MAC cache to exploit MAC locality. The 8KB MAC cache is used for both of the baseline tree-based design and the tree-less TNPU. Including

the MAC cache, TNPU requires 0.03632mm^2 which is only 0.035% of Exynos 990 and 17.73mW at the highest performance point [55], [56].

VI. Related Work

Secure ML execution on accelerators: There have been several proposals to support TEEs on GPUs [9], [10]. Graviton modifies GPUs to isolate the critical operations on GPU contexts within each GPU [9]. HIX isolates I/O paths to GPUs to support trusted controls of GPUs from the protected GPU driver running in a CPU enclave [10]. HETEE encapsulates GPUs and accelerators in a special hardened chassis with a secure control module to serve computation offloading from servers through PCIe fabrics [57]. It allows to use any commodity GPUs and accelerators in the chassis. Alternatively, instead of supporting full TEEs on GPUs or NPUs, several studies adopted homomorphic encryption. Cheetah optimizes homomorphic encryption for inference with parameter tuning and operator scheduling [58]. It also proposes changes in accelerator designs for better support of homomorphic encryption. For an ML task, DarKnight offloads linear operations with encrypted data to unsecure GPUs, while processing the rest of the task in a secure CPU enclave [59]. There have been recent studies to reduce the security overheads of accelerators. SEAL proposes ML optimized encryption techniques by skipping encryption engine for partial data and co-locating data and counters [60]. GuardNN proposes a secure NPU architecture with application-specific version number management [36]. It exploits the static data-flow of DNN workloads to eliminate per-block counters, sharing a similar application-managed approach as TNPU.

Memory protection: For the confidentiality and integrity protection of memory, there have been many studies to mitigate the performance overhead of memory protection in CPU and GPU systems [12], [18], [17]. Since integrity verification is one of the major performance bottlenecks for memory protection, the studies reduce the performance overhead of integrity tree verification in CPU systems. VAULT uses different arities in each level of the counter integrity tree [18]. Morphable counters provides a compact counter representation packing more counters per counter node [17]. Common counters investigated the performance overhead of supporting memory protection for GPUs and proposed an efficient counter representation tuned for GPU applications [12]. Lee et al. investigated the reduction of bandwidth consumption for security metadata in multi-core systems [54].

VII. Conclusion

Trusted NPU (TNPU) extends the trusted execution support to the integrated NPU architectures. It allows an application in a CPU enclave to securely execute ML workloads on NPUs. On top of the trusted execution support, the paper investigated the hardware-based memory protection for NPU computation. The study showed that a naïve adoption of the prior CPU hardware memory encryption for the NPU memory protection can degrade its performance significantly. To address the performance degradation by the hardware memory protection, the study proposed a novel semantic-aware tree-less integrity protection. Our evaluation showed that the support for NPU TEEs is feasible with a minor extra area overhead and small performance degradation over unsecure NPU execution.

Acknowledgment

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grants funded by the Ministry of Science and ICT, Korea (IITP2017-0-00466, IITP2021-0-01817). This work was also partly supported by Samsung Electronics Co., Ltd. (IO201209-07864-01).

References

[1] M. Ditty, A. Karandikar, and D. Reed, "NVIDIA's Xavier SoC," in *Hot Chips: A Symposium on High Performance Chips*, 2018.

[2] J. Song, Y. Cho, J.-S. Park, J.-W. Jang, S. Lee, J.-H. Song, J.-G. Lee, and I. Kang, "An 11.5 tops/w 1024-mac butterfly structure dual-core sparsity-aware neural processing unit in 8nm flagship mobile SoC," in *International Solid-State Circuits Conference (ISSCC)*, 2019.

[3] Apple. (2020) Apple unleashes M1. [Online]. Available: https://www.apple.com/newsroom/2020/11/apple-unleashes-m1

[4] E. Talpes, D. D. Sarma, G. Venkataramanan, P. Bannon, B. McGee, B. Floering, A. Jalote, C. Hsiong, S. Arora, A. Gorti, and G. S. Sachdev, "Compute solution for Tesla's full self-driving computer," in *IEEE Micro*, vol. 40, no. 2, 2020, pp. 25–35.

[5] V. Costan and S. Devadas, "Intel SGX explained," in *IACR Cryptology ePrint Archive*, vol. 2016, no. 86, 2016, pp. 1–118.

[6] T. Alves and D. Felton, "TrustZone: Integrated hardware and software security," in *Information Quarterly*, 2004.

[7] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovi, and D. Song, "Keystone: An open framework for architecting trusted execution environment," in *European Conference on Computer Systems (EuroSys)*, 2020.

[8] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *USENIX Security Symposium (USENIX Security)*, 2016.

[9] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on GPUs," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[10] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity GPUs," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[11] O. Kwon, Y. Kim, J. Huh, and H. Yoon, "Zerokernel: Secure context-isolated execution on commodity GPUs," in *Transactions on Dependable and Secure Computing (TDSC)*, 2019.

[12] S. Na, S. Lee, Y. Kim, J. Park, and J. Huh, "Common counters: Compressed encryption counters for secure GPU memory," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2021.

[13] Intel, "Intel trust domain extensions," Intel, Tech. Rep., 2021.

[14] S. Jin, J. Ahn, J. Seol, S. Cha, J. Huh, and S. Maeng, "H-SVM: Hardware-assisted secure virtual machines under a vulnerable hypervisor," in *Transactions on Computers (TC)*, vol. 64, no. 10, 2015, pp. 2833–2846.

[15] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *International Symposium on Microarchitecture (MICRO)*, 2003.

[16] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and Bonsai merkle trees to make secure processors OS-and performance-friendly," in *International Symposium on Microarchitecture (MICRO)*, 2007.

[17] G. Saileshwar, P. Nair, P. Ramrakhyani, W. Elsasser, J. Joao, and M. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *International Symposium on Microarchitecture (MICRO)*, 2018.

[18] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: Reducing paging overheads in SGX with efficient integrity verification structures," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[19] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[20] Y.-H. Oh, S. Kim, J. Y, S. Son, J. Bae, J. Lee, Y. Park, D. U. Kim, T. J. Ham, and J. W. Lee, "Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2021.

[21] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *International Symposium on Computer Architecture (ISCA)*, 2017.

[22] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Design Automation Conference (DAC)*, 2021.

[23] X. Hu, L. Liang, S. Li, L. Deng, P. Zuo, Y. Ji, X. Xie, Y. Ding, C. Liu, T. Sherwood, and Y. Xie, "DeepSniffer: A DNN model extraction framework based on learning architectural hints," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[24] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures," in *USENIX Security Symposium (USENIX Security)*, 2020.

[25] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust physical-world attacks on deep learning visual classification," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[26] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[27] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction APIs," in *USENIX Security Symposium (USENIX Security)*, 2016.

[28] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations (ICLR)*, 2015.

[29] D. Gascon, "IoT security infographic privacy, authenticity, confidentiality and integrity of the sensor data: The invisible asset," Libelium, Tech. Rep., 2015.

[30] S. Hu, Q. A. Chen, J. Joung, C. Carlak, Y. Feng, Z. M. Mao, and H. X. Liu, "CVShield: Guarding sensor data in connected vehicle with trusted execution environment," in *ACM Workshop on Automotive and Aerial Vehicle Security*, 2020.

[31] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2003.

[32] S. Ozdemir and Y. Xiao, "Secure data aggregation in wireless sensor networks: A comprehensive overview," in *Computer Networks*, vol. 53, no. 12, 2009, pp. 2022–2037.

[33] C. Yan, B. Rogers, D. Englender, Y. Solihin, and M. prvulovic, "Improving cost, performance, and security of memory encryption and authentication," in *International Symposium on Computer Architecture (ISCA)*, 2006.

[34] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "A systematic methodology for characterizing scalability of DNN accelerators using SCALE-Sim," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.

[35] B. Hyun, Y. Kwon, Y. Choi, J. Kim, and M. Rhu, "NeuMMU: Architectural support for efficient address translations in neural processing units," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[36] W. Hua, M. Umar, Z. Zhang, and G. E. Suh, "GuardNN: Secure DNN accelerator for privacy-preserving deep learning," in *Secure and Private Systems for machine Learning Workshop (SPSL)*, 2021.

[37] ARM, "Powering the edge: Driving optimal performance with Ethos-N77 processor," ARM, Tech. Rep., 2019.

[38] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[39] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," in *ArXiv Preprint ArXiv:1704.04861*, 2017.

[40] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[41] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Conference on Neural Information Processing Systems (NIPS)*, 2012.

[42] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Conference on Neural Information Processing Systems (NIPS)*, 2015.

[43] S. Narang and G. Diamos. (2017) DeepBench: Benchmarking deep learning operations on different hardware. [Online]. Available: https://github.com/baidu-research/DeepBench

[44] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[45] H. Park and C. D. Yoo, "Melody extraction and detection through LSTM-RNN with harmonic sum loss," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017.

[46] A. Graves, "Generating sequences with recurrent neural networks," in *ArXiv Preprint ArXiv:1308.0850*, 2013.

[47] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of Go without human knowledge," in *Nature*, vol. 550, no. 7676, 2017, pp. 354–359.

[48] R. Johnson and T. Zhang, "Effective use of word order for text categorization with convolutional neural networks," in *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2015.

[49] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, J. Chen *et al.*, "Deep Speech 2: End-to-end speech recognition in English and Mandarin," in *International Conference on Machine Learning (ICML)*, 2016.

[50] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Conference on Neural Information Processing Systems (NIPS)*, 2017.

[51] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural collaborative filtering," in *International Conference on World Wide Web (WWW)*, 2017.

[52] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka *et al.*, "MLPerf inference benchmark," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[53] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks *et al.*, "MLPerf training benchmark," in *Conference on Machine Learning and Systems (MLSys)*, 2020.

[54] J. Lee, T. Kim, and J. Huh, "Reducing the memory bandwidth overheads of hardware security support for multi-core processors," in *Transactions on Computers (TC)*, vol. 65, no. 11, 2016, pp. 3384–3397.

[55] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," HP laboratories, Tech. Rep., 2009.

[56] Y. Zhang, K. Yang, M. Saligane, D. Blaauw, and D. Sylvester, "A compact 446 Gbps/W AES accelerator for mobile SoC and IoT in 40nm," in *Symposium on VLSI Circuits (VLSI-circuits)*, 2016.

[57] J. Zhu, R. Hou, X. Wang, W. Wang, J. Cao, B. Zhao, Z. Wang, Y. Zhang, J. Ying, L. Zhang, and D. Meng, "Enabling rack-scale confidential computing using heterogeneous trusted execution environment," in *Symposium on Security and Privacy (S&P)*, 2020.

[58] B. Reagen, W.-S. Choi, Y. Ko, V. T. Lee, H.-H. S. Lee, G.-Y. Wei, and D. Brooks, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[59] H. Hashemi, Y. Wang, and M. Annavaram, "DarKnight: An accelerated framework for privacy and integrity preserving deep learning using trusted hardware," in *International Symposium on Microarchitecture (MICRO)*, 2021.

[60] P. Zuo, Y. Hua, L. Liang, X. Xie, X. Hu, and Y. Xie, "SEALing neural network models in secure deep learning accelerators," in *Design Automation Conference (DAC)*, 2021.