Unified Memory Protection with Multi-granular MAC and Integrity Tree for Heterogeneous Processors

Sunho Lee
KAIST
Daejeon, Republic of Korea
myshlee417@casys.kaist.ac.kr

Seonjin Na
Georgia Institute of Technology
Atlanta, GA, USA
seonjin.na@gatech.edu

Jeongwon Choi
KAIST
Daejeon, Republic of Korea
jwchoi@casys.kaist.ac.kr

Jinwon Pyo
KAIST
Daejeon, Republic of Korea
jpyo0803@casys.kaist.ac.kr

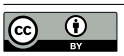
Jaehyuk Huh
KAIST
Daejeon, Republic of Korea
jhhuh@kaist.ac.kr

Abstract

Recent system-on-a-chip (SoC) architectures for edge systems incorporate a variety of processing units, such as CPUs, GPUs, and NPUs. Although hardware-based memory protection is crucial for the security of edge systems, conventional mechanisms experience a significant performance degradation in such heterogeneous SoCs due to the increased memory traffic with diverse access patterns from different processing units. To mitigate the overheads, recent studies, targeting a specific domain such as machine learning software or accelerator, proposed techniques based on custom granularities applicable either to counters or MACs, but not both. In response to this challenge, we propose a unified mechanism to support both multi-granular MACs and counters in a device-independent way. It supports a granularity-aware integrity tree to make it adaptable to various access patterns. The multi-granular tree architecture stores both coarse-grained and fine-grained counters at different levels in the tree. Combined with the multi-granularity technique for MACs. Our optimization technique, termed *multi-granular MAC&tree*, supports four different levels of granularity. Its dynamic detection mechanism can select the most appropriate granularity for different memory regions accessed by heterogeneous processing units. In addition, we combine the multi-granularity support with the prior subtree approaches to further reduce the overheads. Our simulation-based evaluation results show that the multi-granular MAC and tree reduce the execution time by 14.2% from the conventional fixed-granular MAC&tree. By combining prior sub-tree techniques, the multi-granular MAC and tree finally reduce the execution time by 21.1% compared to the conventional fixed-granular MAC&tree.

CCS Concepts

• **Security and privacy** → *Hardware security implementation*; • **Computer systems organization** → *Heterogeneous (hybrid) systems.*



ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731066

Keywords

Hardware security, memory protection, heterogeneous processors

ACM Reference Format:
Sunho Lee, Seonjin Na, Jeongwon Choi, Jinwon Pyo, and Jaehyuk Huh. 2025. Unified Memory Protection with Multi-granular MAC and Integrity Tree for Heterogeneous Processors. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731066

1 Introduction

In edge-level systems, diverse processing units such as CPU, GPU, and NPU (Neural Processing Unit) are integrated into an SoC (System-on-a-Chip) architecture with the shared external memory [3, 5, 12, 45]. For critical operations, edge-level systems must ensure both confidentiality and integrity of memory-resident data with a minimum performance degradation against physical attacks. For confidentiality, all data must reside in the external memory only in encrypted form. For integrity, a hashed value called MAC (Message Authentication Code) is computed and stored in the memory for each 64B cacheline. In addition, to detect replay attacks, each cacheline is paired with its own designated counter to track the version number of the memory block. An integrity tree of the counters is maintained to validate the version numbers of all memory blocks with the root node securely stored only in the on-chip storage. However, the critical memory protection overhead occurs due to the fine-grained cacheline granularity. A fine granularity significantly increases data traffic for both counters and MACs, miss rates of security metadata caches, and the cost of updating the integrity tree.

To reduce the performance overhead of memory protection, recent studies on GPUs and NPUs leverage bulk data transfer characteristics and increase the granularity of security metadata (counters or MACs) [20, 23, 24, 29, 30, 35, 56]. For GPUs, the dual-granular MAC utilizes a coarse-grained MAC to reduce the security metadata overhead [56], while *Common Counters* proposed a shared counter for contiguous memory regions of data without integrity tree modification [35]. However, the prior approaches allow only dual granularity, but more fine-grained levels of granularity are required to efficiently represent diverse workloads in heterogeneous processors. Furthermore, they can improve only either MACs or tree, but not both: coarse-grained MACs without improving the

integrity tree [56], or coarse-grained counters to reduce counter cache misses without improving MAC overheads [35]. To support both, the integrity tree must be changed.

For NPUs, recent studies utilize the software-managed tensor-granularity of the NPU execution [23, 24, 29]. As each tensor shares a single counter based on the compiler detection, significantly fewer counters need to be stored in the on-chip region, removing the necessity of the integrity tree. However, the prior studies focus either on a specific domain, or on the coarse granularity only one of the MACs and counters. The NPU memory protection provides a tree-less mechanism tailored for tensor operations in machine learning, with a limited number of version numbers [20, 23, 24, 29, 52]. However, this approach cannot be applied to general applications.

To address the limitations, this paper proposes a unified memory protection scheme for heterogeneous processors with multi-granularity support for MACs and an integrity tree. The multi-granular tree allows selective node reduction, unlike the prior complete counter tree. The proposed technique can detect the suitable granularity among four sizes (64B, 512B, 4KB, and 32KB), and support the multi-granular management not only for MACs but also for the integrity tree of counters. When the granularity is promoted, the integrity tree delegates the responsibilities of fine-grained child nodes to the parent node. For MACs, multiple cachelines of fine-grained MACs are merged into a single cacheline of coarse-grained MAC for compacting MAC spaces. The paper proposes the address computation mechanism for the modified addresses of promoted counters and merged MACs. Figure 1 (a) presents the conventional integrity tree with a fixed granularity for MACs and the integrity tree, while Figure 1 (b) shows the multi-granular tree which reduces tree traversal latencies and bandwidth consumption with coarse-grained MACs.

The *multi-granular MAC&tree* requires dynamic granularity detection based on an access pattern. As memory accesses occur, the access tracker keeps track of accesses in a one-hot vector entry of a corresponding memory chunk to detect a granularity dynamically. The determined granularity and its position are encoded and stored in the protected granularity table followed by reorganization of the integrity tree. With a promotion, multiple data requests belonging to the coarse granularity share a single counter and a MAC, improving performance by reducing traffic.

A recent alternative approach for reducing the overheads of integrity tree traversal is to decompose the single integrity tree into multiple subtrees. In *Bonsai Merkle Forest* (*BMF*) [17] and *PENGLAI* [16], subtrees with lower heights can reduce the tree traversal latencies if the roots of the subtrees can be stored in the safe on-chip storage. Such a pruning approach can be combined with our multi-granular tree, which can reduce both tree traversal times and counter cache misses, in addition to the decreased MAC overheads. Figure 1 (c) shows the combined scheme.

We evaluate the heterogeneous system of one CPU, one GPU, and two NPUs on the combined simulator of three other simulators: ChampSim (CPU) [18], mGPUsim (GPU) [48], and mNPUsim (NPU) [25]. The evaluation results show that the multi-granular MAC and tree reduce the execution time by 14.2% from the conventional fixed-granular MAC&tree. By combining the sub-tree technique, the multi-granular MAC and tree finally reduce the execution time by 21.1% compared to the conventional scheme.

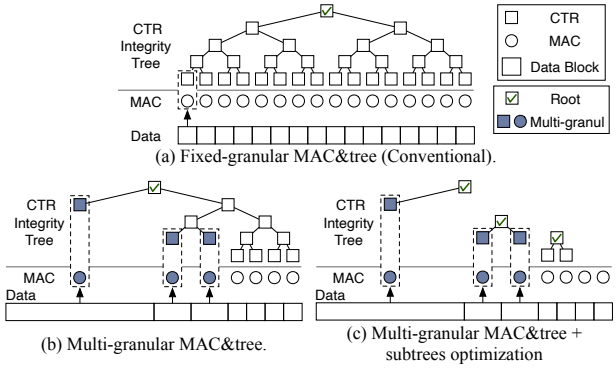


Figure 1: The required counters and MACs in conditions of (a) fixed-granularity, (b) multi-granularity, and (c) multi-granularity with subtree optimization methods.

The main contributions of the paper are as follows:

- The study highlights that a unified multi-granular memory protection scheme is necessary to handle the diverse access patterns in heterogeneous processors.
- It proposes a granularity-aware tree and merged MACs to support multi-granularity both for counters and MACs.
- It proposes a dynamic multi-granular memory protection scheme by detecting the chunk size of memory access.
- It combines the proposed method with existing integrity tree optimization techniques for further improvement.

2 Background

2.1 Heterogeneous Processor

Architecture of heterogeneous processor: There are several commercial products with integrated-NPU [3, 5, 12, 36, 45, 50]. Heterogeneous processors contain three different processing units (CPU, GPU, and NPU). CPU is for general-purpose operations. GPU is designed to accelerate repeated computations by increasing the number of compute units. NPU is a more efficient and powerful processing unit for computing ML workloads by adopting a systolic array and a software-managed on-chip storage (scratchpad memory). All the processing units share the off-chip memory.
Execution model: GPU programming model, called *NVIDIA CUDA*, supports the hierarchical organization of threads and blocks. A group of blocks is assigned to a single streaming multiprocessor (SM). NVIDIA Orin contains an integrated Ampere GPU with up to 16 SMs. Without a dedicated memory, the integrated GPU relies on the system memory being shared by other processing units [12]. NVIDIA NPUs, also known as deep learning accelerators (DLAs), play a key role in accelerating ML tasks as specialized hardware devices. The CPU issues instructions and control signals to manage and drive the GPU and the NPUs to perform distributed tasks and retrieve the results.

2.2 Counter-mode Memory Protection

As shown in Figure 2, modern processors commonly adopt counter-based memory protection to ensure confidentiality, integrity, and freshness [4, 39]. A one-time-pad (OTP) is generated as a function of a secret key, an address, and a counter value. Each 64B memory

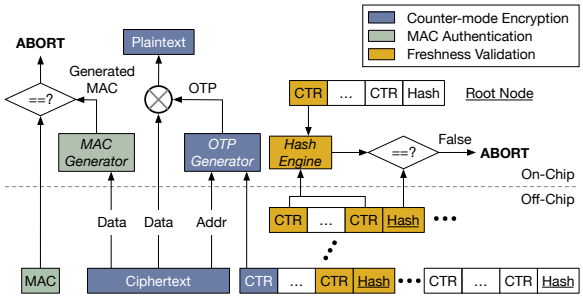


Figure 2: Counter-mode encryption and integrity protection.

block has its own counter, which assures the uniqueness of an OTP and updates only upon dirty eviction. The generated OTP is then XORed with the plaintext to produce a ciphertext or vice versa.

To prevent off-chip data leaks, it is essential to ensure the integrity of the off-chip data. A secure processor generates and sends the message authentication code (MAC), a keyed hash value, along with data to off-chip memory for future data integrity verification. Typically, an 8B MAC is allocated to a 64B memory block. Integrity verification occurs when a cache miss occurs. During this process, a pair of the data and the MAC stored in off-chip memory is retrieved. By comparing the newly generated MAC based on the data with the retrieved MAC, the integrity of the retrieved data can be verified.

Furthermore, freshness validation of counters is necessary to protect from replay attacks, and utilizes the integrity tree traversal. When an LLC miss occurs, the decryption of the ciphertext fetched from off-chip initially involves a sequence of comparisons from the fetched counter (the leaf node) to the root node using the integrity tree. One common approach to mitigate this problem is to maintain on-chip caches for counters (*counter cache*) and the intermediate tree nodes (*hash cache*). In some designs, these two caches are integrated into a unified metadata cache (*metadata cache*).

2.3 Granularity-managed Memory Protection

In a conventional counter-mode encryption scheme, one counter and one MAC are maintained for each cacheline data. Since coarse-grained memory accesses are common in GPUs or NPUs, the multiple counters and MACs represent the same values. As shown in Table 1, several studies of coarse-granular counters and MACs are conducted to remove redundant secure metadata. These studies prove that the accurate coarser-grained counter or MAC improves the counter cache efficiency and reduces the memory bandwidth.

However, prior studies are inappropriate for heterogeneous systems due to the need for advanced designs that should 1) target both counters and MACs, 2) support diverse access patterns resulting from the interplay of CPUs, GPUs, and NPUs, and 3) accommodate a wide variety of workloads. Furthermore, employing separate individual memory protections is impractical for the scalability of heterogeneous processors. This approach leads to several drawbacks: 1) The under-utilization of on-chip resources dedicated to separate security units, 2) an increase in processor design complexity, and 3) the need to develop suitable hardware and software solutions for every new computing device. In response, we propose a unified solution based on a novel integrity tree that dynamically manages both multi-granular counters (of integrity tree) and MACs.

Table 1: Comparisons with the prior studies

Study	Target System	Multi CTR	Integrity Tree Opt.	Multi MAC	Dynamic Granul.	Target App.
SoftVN [52]	CPU	✓	✗	✗	✗ (S/W)	ML-specific
Common Counters [35]	GPU	▲	✗	✗	✗ (Kernel)	General
Yuan. et. al. [56]	GPU	✗	✗	▲	✓	General
TNPU [29]	CPU+NPU	✓	✗	✗	✗ (S/W)	ML-specific
MGX [24]	NPU	✓	✗	✓	✗ (S/W)	ML-specific
GuardNN [23]	NPU	✓	✗	✗	✗ (S/W)	ML-specific
TensorTEE [20]	CPU+NPU	✓	✗	✓	✓	ML-specific
Bonsai Merkle Forests [17]	CPU	✗	Subtrees	✗	✗	General
PENGLAI [16]	CPU	✗	Subtrees	✗	✗	General
Migratable Merkle Tree [15]	CPU+GPU	✗	Subtrees	✗	✗	General
Data Enclave [54]	CPU	✗	Subtrees	✗	✗	General
Ours	CPU+GPU +NPU	✓	CoarseCTR (+ Subtrees)	✓	✓	General

(▲: Dual CTR or MAC), (CoarseCTR: Tree opt. by coarse-grained CTRs)
(Subtrees: Tree opt. by caching roots of subtree based on hotness)

Coarse-grained techniques: *SoftVN* utilizes software hints for CPU memory-intensive workloads to provide only a single counter for a bulk data request [52]. *Common Counters* proposes the shared counter for coarse-grained segments in a GPU environment. When all of the counter values in a particular segment are found to be equivalent in the scanning step, the segment is treated as a coarse-grained unit [35]. The study of a dual-granular MAC uses an access tracker to dynamically track access patterns in GPU scenarios [56].

In NPU cases, *TNPU*, *MGX*, *GuardNN*, and *TensorTEE* propose tensor-based version numbers and MACs [20, 23, 24, 29]. As software can obtain knowledge of the tensor computation in advance during the compilation time, version numbers are stored in on-chip storage and conveyed with NPU commands. These studies exploit the tree-less integrity protection by leveraging the reduced storage for secure metadata. Therefore, the device-specific solutions use the software-detected tensor-granularity scheme to reduce the security metadata (counters and MACs).

ML-specific version number techniques: Several prior studies leverage the tree-less scheme with on-chip stored version numbers [20, 23, 24, 29, 52]. However, this scheme is neither general nor scalable since the number of coarse-grained address ranges is limited. For this reason, studies of on-chip stored version numbers are for tensor-based machine learning workloads. Therefore, for general workloads, integrity tree optimization techniques are necessary, rather than relying on tree-less approaches.

Limitations of previous general solutions: *Common Counters* maintains a limited set of 16 shared counters and falls back to a conventional integrity tree when these counters are insufficient [35]. Since *Common Counters* does not directly modify the integrity tree, it requires dedicated storage and a mandatory scanning step upon kernel termination. The limited number of shared counters makes it challenging to accommodate general patterns of heterogeneous processors (multi-granularity and a lot of coarse regions). Additionally, the scanning step is particularly suited to GPU kernel-based execution patterns. Also, *Common Counters* does not consider any optimization for MACs. A prior study on coarse-grained MAC [56] has focused solely on dual-granular MAC. In this study, we modify the integrity tree to integrate both multi-granular counters and MACs. Furthermore, we achieve additional reductions in switching overhead through lazy switching.

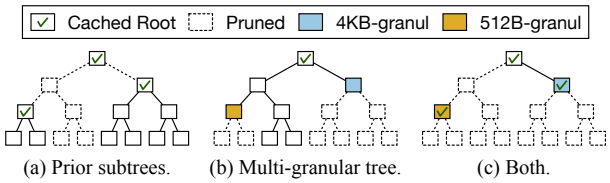


Figure 3: Multi-granular tree compared to prior subtree optimization techniques. The number of pruned nodes increases when both optimization schemes are adopted.

2.4 Integrity Tree Optimization

As shown in Table 1 and Figure 3 (a), several prior studies were able to mitigate the recursive integrity tree traversal overhead with the reduced tree height by dividing [15–17, 54]. In *Bonsai Merkle Forests*, roots of subtrees are stored in on-chip processor storage based on the hotness of access [17]. The root of the integrity subtree for the hot memory region is retained in the on-chip cache to shorten the validation path of the integrity tree. *PENGLAI* suggests mountable trees to prune the roots of subtrees related to unused memory regions [16]. These subtree approaches reduce the height of integrity trees using an orthogonal way to our multi-granular tree. Figure 3 (b) represents the multi-granular tree, pruning tree nodes for coarse-grained data. As shown in Figure 3 (c), two orthogonal techniques can be combined to further reduce the height of integrity trees and yield significant performance improvements.

2.5 Threat Model

In this study, we assume strong attackers capable of controlling operating system and hypervisor, and having physical access to the whole system. Such attackers can examine and modify data in the off-chip memory, and they can manipulate the system address mapping to reroute data to the memory region of a compromised process. Moreover, attackers can launch physical attacks such as bus-probing to intercept or modify the data transmitted between a memory controller and an off-chip memory across the bus. Our trusted computing base (TCB) includes each processor chip (CPU, GPU, and NPU), and application codes running in the trusted execution environments (TEEs).

However, this study does not address the availability of SoC computation such as denial-of-service (DoS) or side-channel attacks that exploit cache-timing, memory access patterns, or speculative execution as in prior studies [1, 9, 20, 24, 27, 29, 35, 53, 57].

3 Motivation

3.1 Diversity of Chunk Access

To investigate access patterns, we measure the ratio of the coarse-grained access via simulation. We use ChampSim [18], MGPUSim [48], and mNPUsim [25]. Our detailed simulation setup is described in Section 5. When all memory blocks that belong to the 64B, 512B, 4KB, or 32KB memory chunk are loaded within a short period (16K cycles), the memory chunk is regarded as a *stream* chunk.

Single processing unit: Figure 4 shows the proportion of streaming chunk granularity when running individual workloads on the CPU, GPU, and NPU. CPU presents that the 64B streaming chunk size is dominant over coarse-grained accesses. Instead of `xal` with 19.5% of 512B granularity, the other CPU workloads show a high

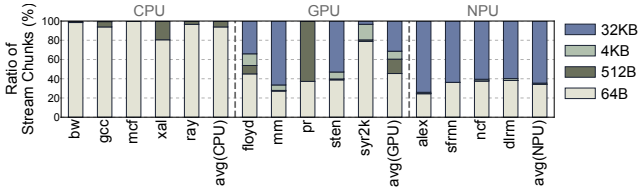


Figure 4: The ratio of stream chunks. *Stream* chunk refers to a memory chunk where all memory blocks are accessed within a specificed time period.

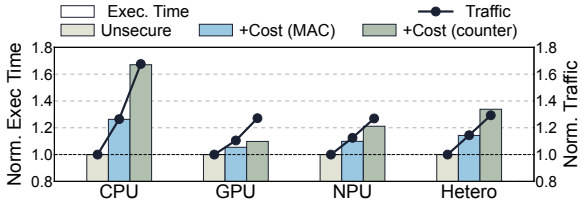


Figure 5: Performance breakdown results of a conventional memory protection technique on each hardware unit.

ratio of 64B fine-grained accesses. GPU shows more diverse memory access patterns. While `syr2k` and `pr` show fine-grained access patterns, `mm` and `sten` show coarse-grained access patterns. In addition, `floyd` shows a relatively diverse access pattern. In cases of NPU, 64.5% of its memory requests use a 32KB coarse-grained streaming chunk size. Notably, `alex` shows relatively higher 32KB coarse-grained accesses (74.1%) than others. From this observation, we summarize the access pattern of the workloads in Table 4.

Heterogeneous processing units: In a heterogeneous system, fine-grained and coarse-grained data are mixed up, displaying diverse granular access patterns. The access pattern and memory traffic intensity of individual processing units influence the overall access patterns. As shown in Table 4, the most coarse-grained heterogeneous scenarios (*cc*) consist of the coarse-grained GPU workload (`mm`) and the coarse-grained NPU workload (`alex`). Despite their coarse-grained access patterns, the NPU workloads `ncf` and `dlrm` are typically categorized under fine-grained scenarios due to their relatively small traffic intensity. As shown in Figure 19 (b), in our heterogeneous scenarios, the ratio of 64B stream chunks is from 22.1% to 60.7%, the ratio of 512B stream chunks is from 0.2% to 6.7%, the ratio of 4KB stream chunks is from 2.1% to 12.1%, and the ratio of 32KB stream chunks is from 34.8% to 71.9%.

3.2 Performance Overhead Breakdown

To identify the factors that degrade performance in memory protection techniques, we conduct a performance breakdown analysis. As shown in Figure 5, we analyze the conventional 64B fixed-granular memory protection overhead by dividing it into the overhead due to MACs for data integrity validation, which is represented as *+Cost (MAC)*, and counters for data encryption and integrity tree verification, which is marked as *+Cost (counter)*.

Single processing unit: While MACs degrade performance by 26.3% (CPU), 5.4% (GPU), and 9.9% (NPU) compared to the unsecured scheme, the performance overhead by encryption and integrity tree verification shows an additional 40.7% (CPU), 4.4% (GPU), and 11.3% (NPU) of performance degradation compared to the unsecured scheme. To sum up, the performance degradation

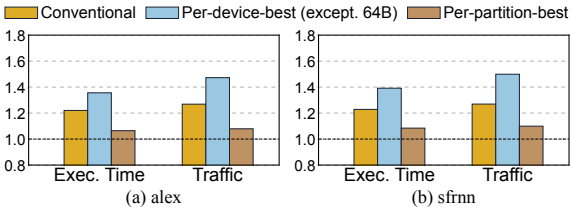


Figure 6: Drawbacks of the per-device granularity: The case of `alex` and `sfrnn`. We define a 512B data block as a *partition*.

caused by MACs and counters reaches 67.0% (CPU), 9.8% (GPU), and 21.1% (NPU) compared to the unsecured scheme. The performance degradation is caused by 67.6% (CPU), 27.1% (GPU), and 26.9% (NPU) of traffic increment compared to the unsecured scheme. It highlights that both counters and MACs are closely related to the performance degradation of the memory protection scheme.

Heterogeneous processing units: A similar pattern is shown in a heterogeneous system. The overhead of MACs degrades the execution time by 14.3% and the overhead of counters causes more delays by 19.5% compared to the unsecured scheme. Therefore, security metadata aggravates the execution time by 33.8%. The execution time of a heterogeneous system highly increases more than that of the traffic increment. When the amount of traffic significantly exceeds the memory bandwidth, stalled memory requests recursively delay subsequent memory requests. Although 8-arity counter cachelines and 8B MAC cachelines include the same 8 number of counters and MACs, counters cause severe extra overhead due to the integrity tree traversal. However, counters are closely associated with the integrity tree, unlike MACs, making it more challenging to readily apply a multi-granularity scheme. To handle this challenge, prior works either focus on dual-granularity counters or MACs instead of the multi-granularity counters and MACs.

3.3 Limitation of Prior Granularity Studies

Domain-specific techniques: Existing studies on coarse-grained granularity rely on the characteristics of specific devices or specific applications [20, 23, 24, 29, 52]. For example, a version number-based tree-less mechanism is a solution for ML workloads with a limited number of tensors. While this approach significantly improves a performance in a specific domain, it cannot be broadly applied to general domains. Therefore, for heterogeneous processors, multiple separate solutions must be independently provided. However, providing separate solutions simultaneously introduces substantial overhead. It increases the design complexity of processors and results in under-utilization due to dedicated hardware. This problem is especially aggravated by the increasing variety of specialized accelerators and growing heterogeneity. Therefore, we focus on the unified solution for heterogeneous processors.

Per-device granularity: The simplest unified solution, per-device granularity, results in the performance degradation. This approach sets the static per-device granularity before execution. Figure 6 presents the per-device granularity and per-partition (512B) granularity. *Per-device-best* means the best per-device granularity.

However, the per-device granularity only reflects the majority of data accesses, causing mispredictions on the other accesses (up to 50% of all accesses). We analyze two workloads (`alex` and `sfrnn`) that exhibit significant overhead in *Per-device-best* scenario.

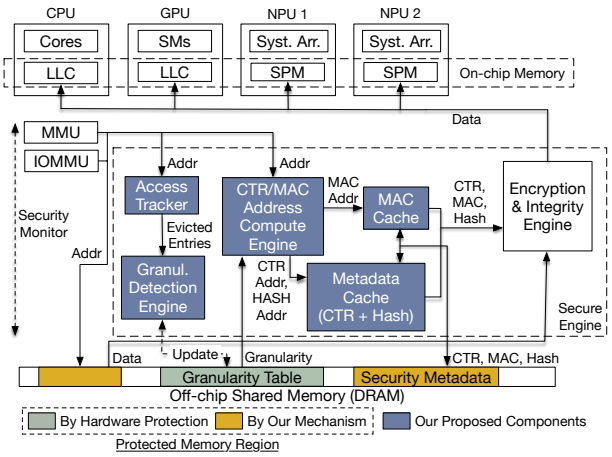


Figure 7: The overview of our proposed architecture.

Unlike *Per-device-best*, which demonstrates performance degradation of 13.6% (`alex`) and 16.3% (`sfrnn`) compared to the conventional scheme along with a traffic increase of 20.4% (`alex`) and 23.0% (`sfrnn`) compared to the conventional scheme, the more fine-grained per 512B partition granularity (*Per-partition-best*) shows improved performance of 15.6% (`alex`) and 14.4% (`sfrnn`) compared to the conventional scheme based on a traffic reduction of 19.0% (`alex`) and 17.0% (`sfrnn`) compared to the conventional scheme. When it extends to the heterogeneous processor, the diverse requests result in an increase in the number of incorrectly classified partitions when using per-device granularity. Additionally, since the exhaustive search for per-partition granularity is impractical due to the vast search space, a dynamic per-partition (512B) granularity detection mechanism is essential.

Dual-granularity techniques: Some prior studies utilizing dynamic granularity detection only support dual-granularity [35, 56]. However, as discussed in Section 3.1, the access pattern of heterogeneous processors shows diverse stream chunks. Therefore, our unified technique should support the multi-granularity.

Only counter or MAC optimization: Several prior studies only optimize either counters or MACs [23, 29, 35, 52, 56]. However, as discussed in Section 3.2, the performance is improved by both counters and MACs. Thus, our mechanism should optimize both counters and MACs.

Given the limitations of prior studies, we propose the dynamic per-partition multi-granular unified memory protection scheme with both counter and MAC optimizations.

4 Architecture

4.1 Overview

To reduce the significant overhead of memory protection on heterogeneous systems, we propose multi-granular MACs and tree-based unified memory protection to exploit the appropriate security metadata granularity. Our mechanism compacts the fine-grained MACs to remove fragmentation between coarse-grained MACs. In addition, our technique reduces the time required to perform the integrity verification process by cutting down the height of the integrity tree, called *multi-granular tree*. Figure 7 presents the proposed architecture overview. To detect the appropriate granularity,

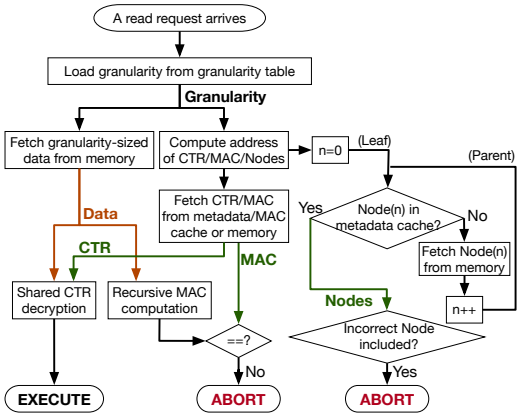


Figure 8: Flowchart of MAC verification and decryption.

we monitor the access pattern per 32KB chunk. The detected access granularity is stored in a granularity table within a protected memory region secured by the discrete fixed-64B integrity tree. Since the multi-granular MACs and tree dynamically relocate security metadata, an address computing engine resolves the addresses of counters and MACs using the address and the granularity.

Figure 8 presents the flow of our memory protection. When a memory access request arrives, the granularity is determined by referencing the granularity table. Afterwards, the data as much as granularity is fetched. Concurrently, the addresses of the counters (both the leaf and the tree nodes) and the MAC are computed and loaded. After the counters and the MAC are fetched from metadata/MAC caches or off-chip memory, the secure engine performs the en/decryption and integrity validation.

4.2 Baseline System

Trusted execution: To ensure the isolated execution from untrusted privileged software, such as an operating system, various trusted execution environments (TEEs) like Intel SGX, Intel TDX, Arm TrustZone, and RISC-V Keystone can be employed [9, 10, 26, 28, 38]. Our study requires a hardware-guaranteed secure memory region to store the granularity table, which can be provided by a TEE with the discrete conventional integrity tree. Furthermore, the proposed method extends existing the memory protection engine (MEE) and is thus executed by a high-privileged security monitor.
Baseline integrity tree: Among various designs of integrity tree, we adopt the 8-arity counter tree design as a baseline integrity tree. In this configuration, each individual intermediate node has 8 (arity) child nodes, and a single counter cacheline includes 8 (arity) counters. This assumption suggests our granularity candidates as 64B, 512B, 4KB, and 32KB, each 8 times coarser than the previous one. Without loss of generality, our mechanism can readily use the other integrity tree design by properly choosing granularity candidates. When the read or write request is issued, the security monitor searches the granularity table stored in the secure memory region. Then, the security monitor provides the memory protection engine with an address and a granularity.

4.3 Multi-granular Integrity Tree

The heterogeneous system requires multi-granular memory protection and the dynamic scaling of granularity during execution. As

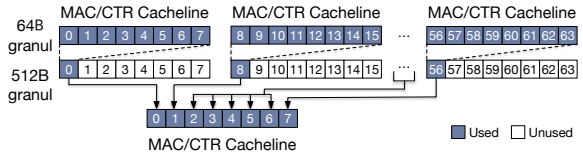


Figure 9: Compaction of course-grained counters and MACs to solve fragmentations. While MACs are merged to the front-most cacheline, counters are promoted to the parent node.

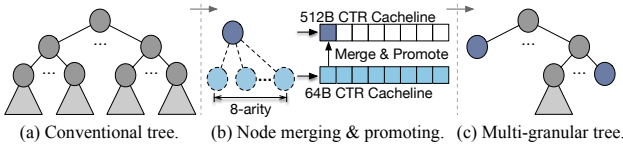


Figure 10: An overview of a multi-granular tree. The diagram describes how tree nodes are merged and promoted.

shown in Figure 9, the increase in granularity leads to a decrease in the number of MACs and counters, which results in memory under-utilization: a cacheline fragmentation. To solve this problem, we fill course-grained counters and MACs from the beginning of the first counter and MAC cacheline one by one to get rid of the empty spaces. For instance, after the merge of the MACs of blocks 0-7 and 8-15 into two coarse-grained MACs, each of which is placed at positions 0 and 8, results in fragmentations in positions 1-7 and 9-15. The two coarse-grained MACs should be moved to positions 0 and 1. To optimize further, we propose the tree node promotion with which the write-back location is promoted to the parent of the integrity tree node. Due to changes in the addresses of counters and MACs caused by the merging and the tree node promotion, we propose an address computation technique based on the granularity. Lastly, we introduce the counter sharing and nested hash function for coarse-grained encryption and authentication.
Multi-granular tree mechanism: Multi-granular tree mechanism shortens the height of an integrity tree by removing fragmentations of course-grained counters. Figure 10 shows the multi-granular integrity tree technique when multiple 64B fine-grained counters are promoted to a single coarse-grained counter. As finer-grained counters are merged into the counter of the parent node, all child nodes are pruned. When the 512B granular stream partition is detected, the responsibilities of eight individual counters are delegated to their parent node. This idea can similarly applied to other cases such as 512B-4KB and 4KB-32KB cases. That is, it reduces the metadata overhead which in turn reduces the recursive validation path of the integrity tree and improves metadata/MAC cache utilization.
Counter/MAC addressing for multi-granularity: We propose the address computation technique to support multi-granular counters and MACs. As shown in Figure 9, several coarse-granular counter/MAC cachelines are merged into one cacheline. Similar to Figure 10, the merged counter cachelines are promoted to the parent nodes. We need a granularity-aware address computation of counters and MACs due to merged MACs and promoted counters.

An address of a counter or a MAC is computed by 32KB chunks, considering that every granularity of security metadata in previous chunks is finest-grained (64B). As the 32KB data chunk is covered by the $\frac{size_{chunk}}{64B}$ number of fine-grained counters or MACs, the

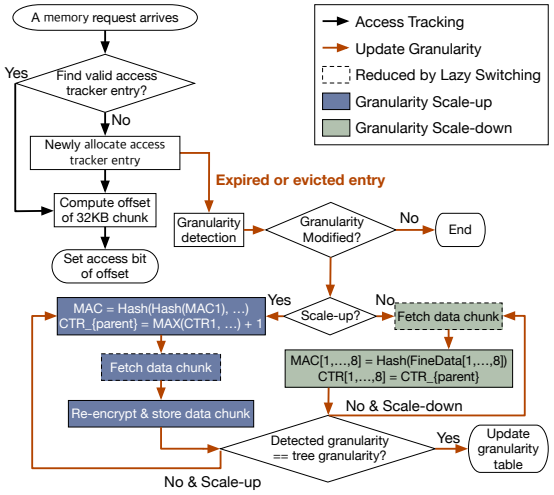


Figure 11: Flowchart of dynamic granularity management.

index of the leaf counter or the MAC is easily computed.

$$Addr_{MAC} = BaseAddr_{MAC} + Idx_{MAC} \times size_{MAC} \quad (1)$$

As shown in Equation 1, we compute the MAC address by adding the base address with the product of the index and MAC size (8B).

$$Parents = \log_{Arity}\left(\frac{granul_{Addr}}{64B}\right) \quad (2)$$

$$Idx_{CTR} = Ancestor^{Parents}(Idx_{CTR,leaf}) \quad (3)$$

$$Addr_{CTR} = BaseAddr_{CTR} + \lfloor \frac{Idx_{CTR}}{Arity} \rfloor \times 64B \quad (4)$$

For the counter, its address is modified when the integrity tree is promoted. The number of pruning steps increases with the granularity while decreasing with the arity of the integrity tree. For example, in an 8-arity tree design, 512B results in a 1-level shorter tree while 4KB or 32KB granularity reduces the 2 or 3 levels of the tree respectively. Therefore, Equation 2 presents the number of parents to reach the correct address which is the logarithm, base $Arity$, of the granularity divided by 64B. As shown in Equation 3, the index of the counter is computed by recursive calls to the ancestor from the index of the leaf counter. Finally, Equation 4 represents the counter address as counter cachelines cover the $Arity$ counters.

Multi-granular encryption and authentication: As the memory protection engine is designed based on a 64B cacheline granularity, we propose the transition between a pair of a coarse-grained counter and a MAC and a pair of a fine-grained counter and a MAC.

$$MAC_{coarse} = \mathbf{Hash}(\mathbf{Hash}(\mathbf{Hash}(MAC_{fine1}), MAC_{fine2}), \ldots) \quad (5)$$

In the case of multiple cachelines sharing a single counter, a multi-granular MAC can be obtained by performing the nested hash computation of fine-grained MACs as in Equation 5. When a coarse-grained data chunk access occurs, the coarse-grained counter and the MAC are loaded along with the data. The loaded data chunk is divided into 64B-sized parts (the finest-grained size), and then each split data is encrypted using its shared counter similar to the prior work [30]. Afterward, the new MAC, generated by nested hash computation from split fine-grained data, is compared to the loaded MAC for integrity authentication.

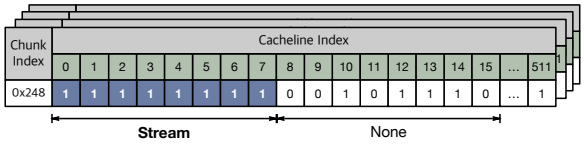


Figure 12: An access tracker to detect a granularity.

Algorithm 1: Granularity Detection Algorithm

```
1  GRANULARITY_DETECTION(access_bits_evict):
2  stream_part ← 0
3  p_0, p_1, ..., p_{num_p − 1} ← SPLIT(access_bits_evict)
4  for i ∈ {0, 1, ..., num_p − 1} do
5      if ISALLSET(p_i) then
6          stream_part ← stream_part + 1
7      end
8      stream_part ← stream_part ≪ 1
9  end
10 return stream_part
```

4.4 Dynamic Granularity Management

Since the appropriate granularity can be determined based on the access pattern, we add an access tracker and a granularity detection engine. The detected granularity is stored in a granularity table in a stream partition format ($stream_{part}$). This format represents locations of 512B the second fine-grained memory blocks. Lastly, we suggest additional bulk requests for misprediction handling. Figure 11 shows the entire flow of a dynamic granularity management.

Access tracker: An access tracker consists of multiple entries, and each entry records accessed cachelines within a chunk to detect a stream pattern. Figure 12 presents the access tracker with multiple entries. Each entry records a *chunk_index* and one-hot access bits. To detect the coarse-grained 32KB granularity, we define 32KB as one *chunk*. We also define an index of a chunk as the upper 49 bits among 64 bits, and the lower 15 bits are used as an offset (*cacheline_index*). When the memory access occurs, the access tracker sets the (*cacheline_index*)-th bit in the corresponding *chunk_index* entry. The entry is evicted either when the number of chunk accesses exceeds the number of cachelines in the chunk ($\frac{32KB}{64B} = 512$) or the lifetime of the entry expires (16K cycles). Also, when there is no available entry for a new memory request for allocation, the entry for eviction is selected according to the least recently used (LRU) policy. Here, We set $3 \times (\#\ of\ processing\ units) = 12$ access tracker entries. This assumption is based on the same on-chip memory requirement for access tracker entries as in the prior study [56].

Access granularity detection: To determine the granularity of the security metadata, the granularity detection algorithm is performed after the access tracker entry is evicted. Algorithm 1 describes the procedure of granularity detection. We define a memory block with a second fine-grained granularity (512B) as a *partition*. We define partition as *stream* partition in which all cachelines (64B) are loaded or stored within a short time window. (16K cycles) Since we store the granularity information into a position map of stream partitions ($stream_{part}$), the granularity detection algorithm tracks $stream_{part}$. The granularity detector scans the evicted entry. Then, its access bits are divided into several partitions. (*line 3*) If every access bit in the partition is set, the partition is classified as a stream

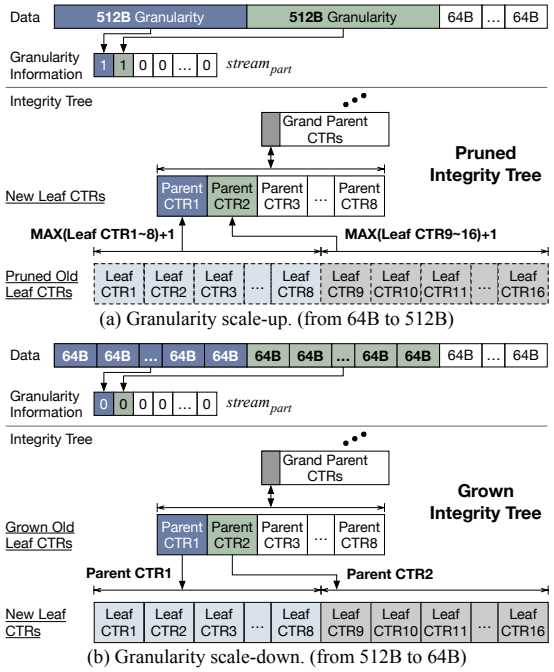


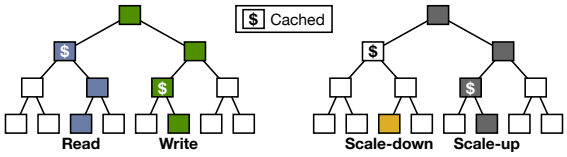
Figure 13: Integrity tree node promoting during a granularity switching. (a) When the granularity of the first two 512B partitions changes from 64B to 512B, leaf counters are pruned. (b) When the granularity is scaled-down, leaf counters are generated and assigned the same value as the parent counter.

partition. (*line 5*) Representing the location of the stream partition, $stream_{part}$ increases by one for recording location and shifts-left by 1 (or multiplied by 2) for tracking the next partition. (*line 4-9*)

Granularity switching: For granularity switching, new MACs and counters corresponding to the new granularity need to be generated. Figure 13 (a) describes the promotion scenario of the first two 512B partitions from 64B to 512B granularity. Therefore, the first 16 fine-grained MACs can be represented with 2 coarse-grained MACs after the promotion. These 2 MACs are computed by recursive hashing of old MACs. Then, the maximum values of leaf counters are stored in parent counters after added by one. Finally, the granularity table is updated to prune leaf counters and MACs.

As shown in Figure 13 (b), the reverse process (granularity scale-down) occurs in a similar manner. Each 64B block is replaced with fine-grained MACs, and the parent counter is divided into eight leaf counters. However, a key difference lies in how the counter values are handled. In granularity scale-up, the parent counter is updated to a new value that has not been previously used. In contrast, in granularity scale-down, the same counter value is retained. This is because, in a coarse-grained setting, all data blocks of child nodes already share the highest counter.

Granularity switching is a costly operation requiring extra memory read/writes, re-encryption, and MAC re-calculation for a corresponding data block. Granularity switching is performed when a misprediction occurs in a memory region that has been accessed more than once. The probability of misprediction is 26.5%. To minimize such overhead, we employ *lazy granularity switching*. To



(a) Read/write requests. (b) Granularity switching requests.

Figure 14: Required integrity tree nodes for read/write requests and granularity switching requests.

support lazy granularity switching, both the current (old) and next (new) granularity are stored in the granularity table.

Granularity switching overhead: Similar to the prior study [56], we analyze overhead in terms of additional data traffic. Granularity switching fetches additional tree nodes, MACs, or data chunks.

Figure 14 shows the integrity tree nodes required for read/write requests and granularity switching procedures. For a read request, nodes only be fetched up to the metadata cache, while for a write request, the fetch operation extends to the root node. Granularity scale-down requires fetching only the leaf node, whereas granularity scale-up requires fetching all nodes up to the root similar to a write request. This distinction arises because, in a granularity scale-down, the counter value is reused in child nodes, whereas in a granularity scale-up, the counter value is updated.

Based on this, Table 2 lists the additional fetch overhead for granularity switching. In the case of scale-down (coarse to fine), lazy switching eliminates the need to fetch the leaf node due to following read/write requests. Conversely, in scale-up (fine to coarse), write-after-read (WAR) and write-after-write (WAW) requests do not incur additional overhead, as the subsequent write operation inherently fetches all nodes up to the root. However, in read-after-read (RAR) and read-after-write (RAW) requests, the tree nodes should be fetched from the node in a metadata cache to the root. Nevertheless, since RAW requests benefit from prior write requests, the probability of a metadata cache hit is relatively high. Consequently, additional tree node fetches due to granularity switching predominantly occur in RAR requests (only 8.8% of requests).

To support the multi-granular MACs, we combine a similar overhead handling technique from the prior study [56] with lazy switching as shown in Table 2. By storing the fine-grained MACs only for read-only data in the unprotected memory region, the overhead of granularity scale-down on the read-only data falls down to a negligible overhead. Thanks to lazy switching, granularity scale-up also be trivial. Only granularity scale-down for non read-only data makes additional data chunk accesses (only 2.8% of requests).

Granularity table: We store the current and next granularity information ($stream_{part}$) in the granularity table. The granularity table is stored in a secure memory region to defend malicious accesses by modifying the granularity and its address computation. Each granularity table entry requires 64 bits to keep track of the current granularity per chunk. The bits of $stream_{part}$ are set if the corresponding partition is determined to be a stream partition. For example, *0b101000...* means the first and the third 512B partitions of the chunk are 512B granularity and the other partitions are 64B granularity, and *0b111...1* represents the 32KB granularity.

The granularity table is located within a protected memory region of 128-256MB, provided by the existing protection mechanism [9]. This region is secured by a fixed 64B-granular counter,

Table 2: Additional fetch overhead of granularity switching

From	To	Type	Counter and Integrity Tree Additional Fetch Overhead	Reason	Ratio
Coarse	Fine	All	Zero	Lazy switching	4.4%
Fine	Coarse	WAR	Zero	Lazy switching	5.1%
Fine	Coarse	WAW	Zero	Lazy switching	3.0%
Fine	Coarse	RAR	Low (Fetch parent to root)	-	8.8%
Fine	Coarse	RAW	Negligible (Fetch parent to root)	Metadata cache	5.2%
Correct prediction (Fine-Fine, Coarse-Coarse)					73.5%

WAR: Write-After-Read, WAW: Write-After-Write
RAR: Read-After-Read, RAW: Read-After-Write

From	To	R/O	Message Authentication Code (MAC) Additional Fetch Overhead	Reason	Ratio
Coarse	Fine	Yes	Negligible (Fetch fine MACs)	Constant fine MACs	1.6%
Coarse	Fine	No	Moderate (Fetch whole data chunk)	-	2.8%
Fine	Coarse	All	Zero	Lazy switching	22.1%
Correct prediction (Fine-Fine, Coarse-Coarse)					73.5%

R/O: Read-Only, Moderate: Similar to [56]

MAC, and integrity tree. For a 4GB memory system, the size of the granularity table is approximately 2MB, comprising 1MB for storing the current granularity ($4GB \times \frac{1bit}{512B}$) and 1MB for the next granularity, being too large to be stored on-chip storage. Each granularity table entry is 16B (8B for the current granularity and 8B for the next granularity), which corresponds to 16B per 32KB of data. Considering the conventional Intel SGX architecture with a 4KB counter cache, the high locality of granularity table entry only increases 0.3% overhead compared to data access overhead even if the granularity table is on a protected memory region.

Misprediction handler: The multi-granularity mechanism manages the granularity based on the previous access pattern. Thus, there is a possibility of misprediction when the actual access is different from the predicted access. As for convolution operations, the access patterns of tensors can vary significantly depending on where the *im2col* operation (either in CPU or NPU) is performed. If misprediction occurs, our mechanism provides a lazy granularity switching procedure, loading all required data similar to Table 2.

4.5 Hardware Overhead

Our mechanism adopts access tracking components and granularity switching logics. Since the prior study utilized 96 4KB-covered access tracker entries [56], our mechanism adopts 12 32KB-covered access tracker entries to track the same area. Single entry tracking a 32KB region requires $\frac{32KB}{64B} = 512bits$. With the chunk index shifted by 15 bits in a 64-bit address space, an additional 49 bits are necessary. Consequently, the total hardware overhead for the access tracker amounts to $12 \times 561bits = 842B$ of on-chip storage. In the granularity detection process, a buffer of $\frac{32KB}{512B} \times 1bit = 64bits = 8B$ is required to store a temporal granularity value ($stream_{part}$). Additionally, $bitshift$ and $add1$ operations are needed. For granularity switching, decryption, re-encryption, and MAC recalculation can be handled by the existing security engine. The address of MACs or counters like Equation 1-4, only needs an additional ALU.

Thus, a total of 850B on-chip storage and an ALU capable of arithmetic, $bitshift$ and $add1$ operations, are essential. Using CACTI, 850B on-chip storage requires 0.013mm^2 of area and 0.04mW of power [33]. Prior research indicates that the ALU requires 0.09mm^2 of area and 213mW of power [37]. Considering NVIDIA Xavier [32] with 350mm^2 of area and 30W of power, the prior version of NVIDIA Orin, our proposed design only consumes 0.029% of area and 0.71% of power overheads.

Table 3: Simulated heterogeneous system configuration [12]

	CPU	GPU	NPU
Cores	8-core	14 SMs	45 × 45 Systolic array
Frequency	2.2 GHz	1 GHz	1 GHz
On-chip Memory (L1-L2)	Cache, 64 KB-2 MB	Cache, 192 KB-4 MB	Scratchpad memory, 2.2MB (in total)
Memory System	LPDDR4, 2.4 GHz frequency, 17 GB/s bandwidth		

5 Evaluation

5.1 Methodology

Accelerator-integrated heterogeneous simulator: We model the heterogeneous system using execution cycles and access traces obtained from three open-source simulators, ChampSim [18] (CPU), MGPUSim [48] (GPU), and mNPUsim [25] (NPU). ChampSim creates a CPU trace and simulates a CPU execution. MGPUSim generates a GPU trace and an execution log of GPU operations. Implemented based on DRAMsim, mNPUsim simulates a multi-NPU environment and provides information about off-chip traffic and execution latency. We designed the heterogeneous system by adding memory requests of MGPUsim and ChampSim to mNPUsim and delaying the GPU warp computation or CPU operation with the memory requests from NPU and other processing units.

Memory protection engine: The secure engine is integrated into the original simulator for memory protection. We use hyperparameters for the secure engine from the prior work [29]. To model the counter-mode protection, the 8-arity counter tree design with 8B MAC is adopted. The latency overhead of OTP generation is fixed to 10 cycles and XOR operation to 1 cycle. We adopt the 8KB metadata cache and 4KB MAC cache for efficient memory protection.

Our additional security design: We construct the additional components for the multi-granular MACs and an integrity tree. For granularity detection, we employ 12 number of access tracker entries. Each access tracker entry is evicted by the LRU policy when the number of accesses to a specific 32KB chunk exceeds 512 ($\frac{32KB}{64B}$), the number of tracked chunks exceeds 12, or the lifetime of the access tracker entry exceeds 16K cycles.

Hardware configuration: As shown in Table 3, we set the simulator environment emulating the NVIDIA Orin system [12]. It consists of an 8-core Orin Arm Cortex, an Orin Ampere GPU, two NVDLAs, and the LPDDR memory system. We use the 2.2GHz 8-core CPU with L1 caches of 64KB and L2 caches of 2MB, and 1GHz clock frequency 14-SMs GPU with L1 cache of 192KB and L2 cache of 4MB. Also, we used a 45 × 45 systolic array, scratchpad on-chip memory of 2.2MB, and INT8 precision NPU with 1GHz clock frequency. We also simulated the 17GB/s bandwidth (2 channels × 8.5GB/s) LPDDR4 memory system with 2.4GHz frequency.

Workloads and scenarios: As shown in Table 4, we used 5 CPU benchmarks, 5 GPU benchmarks, and 4 NPU benchmarks. Therefore, the total number of Orin scenarios is $5 \times 5 \times \binom{4+2-1}{2} = 250$. We select the confined, but diverse workloads regarding the access pattern, benchmark suite, and memory traffic ratio. Among the

Table 4: Simulated workloads & scenarios

Workloads Access Pattern: Fine - *ff* - *f* - *c* - *cc* - Coarse | Diverse (*d*) Traffic per Cycles: Small (*s*) - Medium (*m*) - Large (*l*)			
CPU	**SPEC2017**	*ff-s*	Fluid-Dynamics (bw)
		ff-s	C-Compiler (gcc)
		ff-m	Route-Planning (mcf)
		f-m	XML-HTML-Conversion (xal)
	PARSEC	*ff-s*	Ray-Tracing (ray)
GPU	**AMD APP SDK**	*d-s*	Floyd-Warshall (floyd)
		cc-m	Matrix-Multiplication (mm)
	Pannotia	*f-m*	Page-Rank (pr)
	SHOC	*c-l*	Stencil2d (sten)
	Polybench	*ff-m*	Symmetric-Rank-2k (syr2k)
NPU	**Recommendation**	*c-s*	NCF-Recommendation (ncf)
		c-s	DL-Recommendation (dlrm)
	CNN	*cc-m*	Alexnet (alex)
	RNN	*c-l*	Selfish-RNN (sfrnn)

Selected Scenarios (Only for Section 5.4)
Access Pattern: Fine - *ff* - *f* - *c* - *cc* - Coarse

ID	CPU	GPU	NPU1	NPU2
ff1	bw	syr2k	ncf	dlrm
ff2	mcf	syr2k	sfrnn	dlrm
ff3	gcc	floyd	sfrnn	ncf
f1	xal	pr	sfrnn	ncf
f2	xal	pr	ncf	ncf
c1	gcc	sten	alex	dlrm
c2	bw	sten	ncf	ncf
c3	mcf	sten	sfrnn	sfrnn
cc1	xal	mm	alex	dlrm
cc2	ray	mm	alex	alex
cc3	ray	floyd	alex	alex

Table 5: Simulation schemes

	Scheme	Memory Protection Description
Prior Work Comp.	Adaptive [56]	64B-granular CTRs, dual-granular MACs
	CommonCTR [35]	Dual-granular CTRs, 64B-granular MACs
	BMF&Unused [17]	64B-granular CTRs/MACs w/ subtree opt.
	BMF&Unused+Ours	**Multi-granular CTRs/MACs w/ subtree opt.**
Perf. Analyze	Unsecure	w/o memory protection
	Conventional	64B-granular CTRs/MACs
	Static-device-best	Best per-device static-granular CTRs/MACs
	Multi(CTR)-only	Multi-granular CTRs, 64B-granular MACs
	Ours	**Multi-granular CTRs/MACs**

widely used SPEC2017 and PARSEC [6, 46], we select 5 CPU benchmarks depending on access patterns. For GPU, we take 5 workloads from AMD APP SDK, Pannotia, SHOC, and Polybench [2, 7, 11, 19]. We include NPU workloads from CNN, RNN, and recommendation systems [21, 22, 31, 40]. Each workload is categorized depending on the access granularity (From fine to coarse, *ff*, *f*, *c*, *cc*, and diverse *d*) and memory traffic per second (From small to large, *s*, *m*, *l*).

From these workloads, we set up every case of scenarios including one CPU workload, one GPU workload, and two NPU workloads for the heterogeneous system. Only for a detailed analysis in Section 5.4, we choose 11 scenarios. The selected scenarios are named based on the proportion of stream chunk sizes: *ff*, *f*, *c*, and *cc* for scenarios with a relatively high proportion of 64B, 512B, 4KB, and 32KB stream chunks compared to other scenarios, respectively. These scenarios are configured with a similar distribution of workloads.

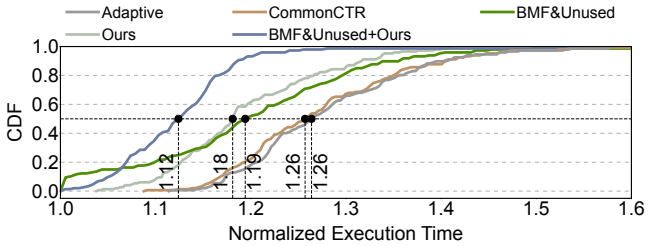


Figure 15: Execution time comparison with prior studies by cumulative distribution functions (CDF). It is measured by averaging the execution time of four processing units.

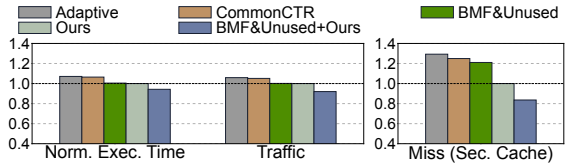


Figure 16: Execution time, amount of traffic, and number of security cache misses compared to prior studies.

Simulation schemes: Table 5 lists the simulated schemes. The first four schemes are presented for performance improvement compared to and combined with prior studies. `Adaptive` and `CommonCTR` model memory protection with dual-granular MACs and dual-granular counters, respectively [35, 56]. `BMF&Unused` refers to prior subtree-based optimization schemes including subtree root caching of *Bonsai Merkle Forests* and unused memory region pruning of *PENGLAI* [16, 17]. `BMF&Unused+Ours` represents the performance when both our multi-granular techniques and the previous subtree-based optimizations are integrated. The following five schemes illustrate the performance breakdown. As a baseline, `Conventional` represents the traditional memory protection, using fixed 64B-granular counters and fixed 64B-granular MACs. `Static-device-best` denotes the static per-device exhaustive search method. This scheme involves exploring all possible granularities for each device and selecting the most optimal granularity. `Multi(CTR)-only` demonstrates the performance of optimized dynamic multi-granular counters, paired with unoptimized 64B MACs. `Ours` presents our proposed architecture, which integrates multi-granular counters and MACs with dynamic modification via *multi-granular MAC&tree*.

5.2 Performance Improvement

Figure 15 and Figure 16 show the normalized execution times of 250 scenarios. To measure the normalized execution times of four processing units (1 CPU, 1 GPU, and 2 NPUs), the execution time of each processing unit is divided into the execution time of the unsecured version. Then, we average these four normalized latencies.

Figure 15 shows the performance comparison between the study of dual-granular MACs (`Adaptive`) and the study of dual-granular counters (`CommonCTR`). Our mechanism achieves 8.5% and 7.7% higher performance compared to `Adaptive` and `CommonCTR` on average, respectively. `Adaptive` employs a 4KB dual-granular MAC. From a scalability perspective, `Adaptive` simultaneously stores both coarse and fine granular MACs in memory and lacks an optimization mechanism for counters, making it non-scalable in heterogeneous processors. `CommonCTR` proposes a 32KB dual-granularity

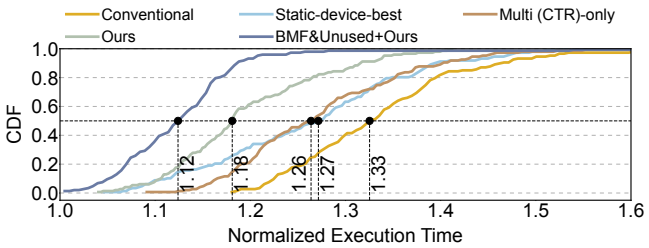


Figure 17: Performance breakdown of our multi-granular design by cumulative distribution functions (CDF) of the normalized execution time.

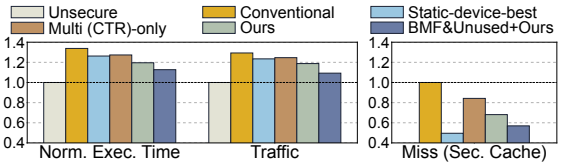


Figure 18: Performance improvement, data traffic reduction, and the number of security cache misses reduction with each optimization adopted from the conventional system. Execution time and data traffic are normalized to the unsecured scheme, and the number of security cache misses is normalized to the conventional scheme.

only for counters, thus limiting the performance improvement. Additionally, it is not scalable in a heterogeneous environment due to the scanning overhead for 32KB segments, the table for recording common counters, and a limited set of common counters (16 number of common counters). By effectively optimizing both counters and MACs based on a single granularity detection, our proposed mechanism provides a scalable solution for heterogeneous systems.

Figure 15 shows the normalized execution time of applying *Bonsai Merkle Forests* with unused memory region pruning of *PENGAI* (`BMF&Unused`). Compared to standalone `BMF&Unused` and `Ours`, we show performance improvements by 7.4% and 6.9%, respectively, when prior subtree-based optimization techniques are incorporated (`BMF&Unused+Ours`). However, there are scenarios where the performance of `BMF&Unused` is lower than that of `BMF&Unused+Ours`, which can be attributed to an increased penalty from mispredictions due to the overall reduction in execution time. This indicates that our approach provides a unified memory protection mechanism based on access patterns of heterogeneous processors, with only a 12.7% performance overhead compared to the unsecured scheme.

Since heterogeneous processing units impose an excessive burden on the memory bandwidth, the amount of traffic is critically related to the execution time. Figure 16 shows the amount of traffic being normalized to `Ours`. The trend of data traffic amount is similar to that of the execution time. While 7.0%, 6.1%, and 0.2% more amount of data traffic is transferred by `Adaptive`, `CommonCTR`, `BMF&Unused` compared to `Ours`, the amount of data traffic in `BMF&Unused+Ours` reduced by 9.5% compared to `Ours`. Therefore, `BMF&Unused+Ours` show only a 9.3% traffic increment compared to the unsecured scheme. From this result, we confirm that the performance of our proposed method is caused by the reduced security metadata.

The amount of security metadata is closely related to the security cache misses from a metadata cache and a MAC cache. Figure 16 presents the number of security cache misses normalized to `Ours`. `Ours` shows 19.9%, 17.0%, and 14.3% of reduced security cache misses compared to `Adaptive`, `CommonCTR`, and `BMF&Unused`. It further reduces 11.2% security cache misses in `BMF&Unused+Ours` scheme compared to `Ours`. Therefore, our significant reduction of security cache misses improves the utilization of security caches and reduces the data traffic and the additional security computation overhead.

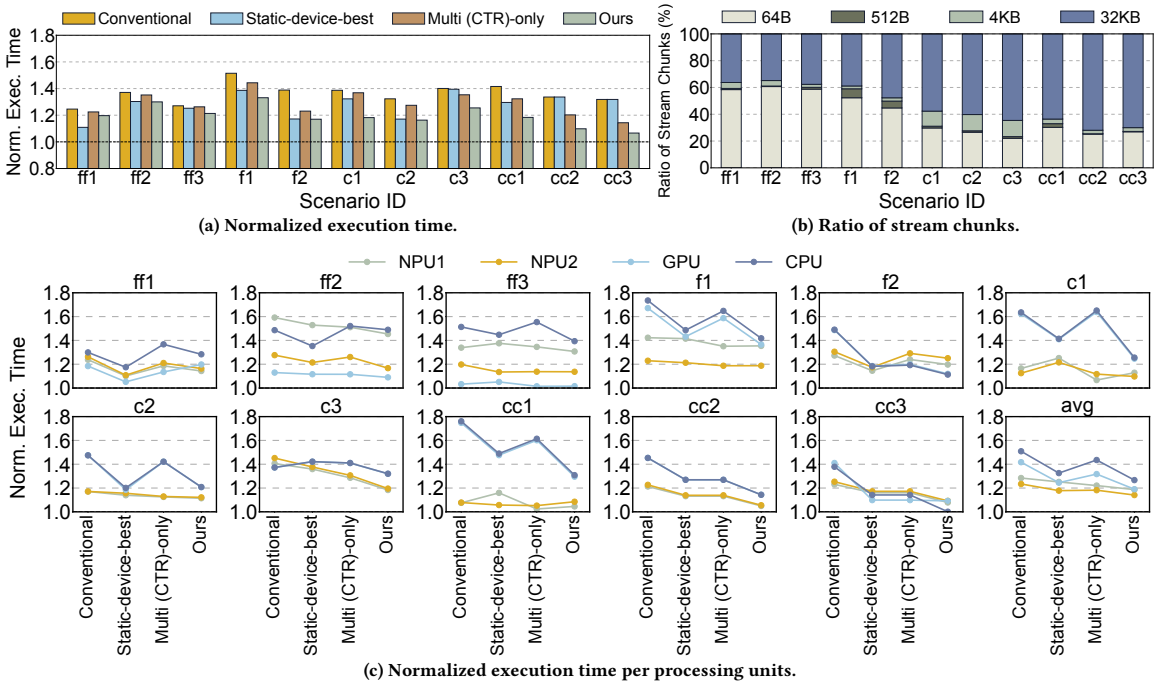
5.3 Performance Breakdown

Figure 17 and Figure 18 show a performance breakdown from the conventional scheme. Similar to Section 5.2, the normalized execution time of each processing unit is computed as the division of its execution time by that of the unsecured scheme. Also, we define the normalized execution time of each scenario as the average of the normalized execution time of four processing units. Compared to the conventional scheme, `Ours` reduces the security overhead in normalized execution time from 33.9% to 19.6% on average. By adopting the subtree-based optimization schemes (`BMF&Unused+Ours`), it is further reduced to 12.7%.

A per-device exhaustive search (`Static-device-best`) presents the fixed granularity per device during the execution, only improving 7.5% of execution time compared to the conventional scheme. Even if the best per-device fixed granularities are selected among 64B, 512B, 4KB, and 32KB, `Static-device-best` shows inefficiency, requiring the dynamic per-partition (512B) memory protection scheme. Furthermore, a per-device granularity technique requires an expensive warmup process for each execution to find the best granularities. Compared to `Static-device-best`, our dynamic and multi-granularity technique accomplishes 14.3% performance improvement, 6.8% more performance improvement than a per-device granularity technique.

The only multi-granular counters (`Multi(CTR)-only`) improve only 6.5% of execution time, compared to the conventional scheme, requiring the necessity for MAC optimization. The proposed approach in this study allows both the counter and MAC to share a single granularity, thereby providing both multi-granular counters and MACs. With both of these optimizations presented at the same time, our method achieves 14.3% performance improvement, 7.8% better performance improvement than `Multi(CTR)-only`.

Figure 18 presents the amount of data traffic and the number of security cache misses. While only 4.7% of data traffic is reduced compared to the conventional scheme by the optimization of only counters, it is significantly reduced with both counters and MACs by 10.5%. Also, we observe that only 5.9% of data traffic compared to the conventional scheme is reduced by the best per-device fixed granularity. The number of security cache misses represents a similar trend instead of `Static-device-best`. While only 15.8% of cache misses compared to the conventional scheme are reduced from multi-granular counters, a further 16.1% reduction of cache misses occurs in our design with both multi-granular counters and MACs. Although `Static-device-best` shows an 18.5% lower cache miss ratio compared to `Ours`, `Static-device-best` sets the aggressively coarser granularity, causing additional bulk data accesses due to mispredictions. By combining subtree-based tree optimization techniques, `BMF&Unused+Ours` finally reduces the amount of data traffic by 9.3% and the number of security cache misses by 56.9% compared to the conventional scheme.



(a) Normalized execution time.

(b) Ratio of stream chunks.

(c) Normalized execution time per processing units.

Figure 19: Performance analysis of selected scenarios.

5.4 Analysis of Selected Scenarios

Among 250 scenarios, we select 11 scenarios and classify selected scenarios as 4 groups (*ff*, *f*, *c*, and *cc*) by the access pattern.

Figure 19 (a) shows the normalized execution time of the selected scenarios. From left (*ff*) to right (*cc*), the ratio of possible improvement from coarse-grained chunks increases. While the finest group (*ff*) shows a smaller improvement (5.9%), the coarsest group (*cc*) shows a high performance gain (24.1%) compared to the conventional scheme. As shown in Figure 19 (b), these gains come from the reduced number of security metadata by capturing stream chunks with the appropriate granularity. Coarse-grained scenarios (from *c1* to *cc3*) show 29.3% higher ratio of coarse-grained stream chunks (4KB or 32KB) than fine-grained scenarios (from *ff1* to *f2*). Therefore, the performance improvement of multi-granularity is higher in coarse-grained scenarios.

As shown in Figure 19 (c), each device shows a different execution time variation. *c1* and *cc1* represent cases where CPU and GPU are significantly improved. Their performance gains were achieved even at the expense of NPU performance. Execution times of NPUs, as shown in cases like *c3*, *cc2*, and *cc3*, are quite reduced since the NPU granularity is relatively large. As the coarse-grained scenarios (from *c1* to *cc3*) usually show high performance improvements access all processing units. From this observation, we confirm that our mechanism effectively detects coarse-grained patterns. However, in the fine-grained scenarios (from *ff1* to *f2*), the performance improvement is relatively modest. These scenarios show a zero-sum game of each device or only a specific device gets a small gain.

The execution time of CPU or GPU is relatively more reduced than that of NPUs. This feature is attributed to the bursty nature of an NPU, which transfers bulk data requests in a short period.

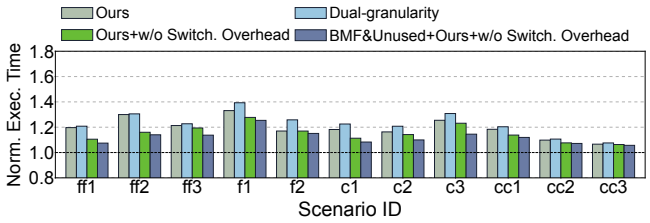


Figure 20: Normalized execution time adopting dual-granularity or eliminating switching overhead.

The large volume of data requests generated by the NPU blocks subsequent memory requests from the CPU and GPU. Through our mechanism, bursty NPU requests are rapidly handled, processing stalled CPU and GPU requests more quickly than a conventional technique. Consequently, on average, our technique improves a performance by 24.2% (CPU), 22.7% (GPU), and 9.5% (NPU).

Figure 20 shows the performance degradation of dual-granularity and switching overhead. Overall, dual-granularity exhibits a slight performance variation, yielding an average performance delay of 3.3% compared to `Ours`. Notably, for workloads containing a mix of 512B and 4KB chunks (from *f1* to *c3*), a performance degradation of 5.8% compared to `Ours` is observed. Given the increasing diversity of workloads in future processing units, even greater performance improvements are expected. Furthermore, as evidenced by the difference between the first and third bars, eliminating switching overhead presents an additional opportunity for an average performance gain of 4.4%. By combining with subtree-based prior optimizations (`BMF&Unused+Ours+w/o Switch.Overhead`), perfect prediction (same as removing switching overhead) results in only a 12.1% performance delay compared to the unsecured scheme.

Table 6: Scenarios of real-world applications

Finance [14, 43, 51, 58]	Data movement: GPU (pr) → CPU (mcf) → NPU (dlrm)		
	GPU	Page-Rank (pr)	Financial risk/commodity network
	CPU	Route-Planning (mcf)	Optimal asset allocation
	NPU	DL-Recomm. (dlrm)	Investment recommendation
AutoDrive [42, 47, 55]	Data movement: GPU (sten) → NPU (yt) → CPU (sc)		
	GPU	Stencil2d (sten)	Camera data filtering/preprocessing
	NPU	Yolo-Tiny (yt)	Obstacle detection
	CPU	Stream-Clustering (sc)	Obstacle K-means clustering

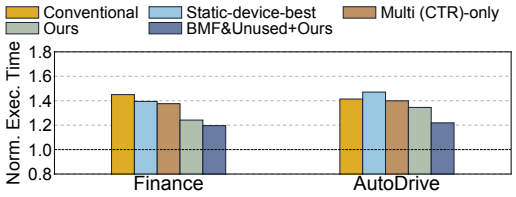


Figure 21: Normalized execution time improvement of our multi-granularity design in real-world applications.

5.5 Real-world Applications

Table 6 represents a real-world application. In *Finance*, the GPU measures risks and identifies financial networks between stocks and commodities using Page-Rank (pr) [51, 58], subsequently transferring the output data to the CPU. The CPU then utilizes Route-Planning (mcf) [14] to compute optimized asset allocation, which is subsequently transmitted to the NPU. The NPU employs DL-Recommendation (dlrm) [43] to generate investment recommendations by integrating the received asset allocation with customer profiles and previous investment records. In *AutoDrive*, the GPU processes 2D camera data using Stencil2d (sten) [55] to perform noise reduction before transferring the refined data to the NPU. The NPU then applies Yolo-Tiny (yt) [42] for obstacle detection. Once obstacles are identified, the CPU performs Stream-Clustering (sc) [47] to group detected obstacles, and this clustered information is utilized for path planning.

Figure 21 shows the performance improvement of real-world scenarios. In *Finance* scenario, the conventional approach resulted in a performance degradation of up to 45.0% compared to the unsecured scheme. However, by employing *Multi-granular MAC&tree*, the degradation is reduced to 24.2% compared to the unsecured scheme, and further optimization through prior subtree-based techniques lowered it to 19.6%. For *AutoDrive* scenario, the conventional approach results in a 41.4% performance degradation compared to the unsecured scheme, which is mitigated to 34.5% through *Multi-granular MAC&tree*. By integrating this approach with prior subtree-based optimizations, the performance degradation was further reduced to 21.9% compared to the unsecured scheme, demonstrating a substantial improvement. Notably, *AutoDrive* exhibits worse performance of the static granularity scheme than the conventional scheme, highlighting the advantages of dynamic selection.

6 Related Work

Trusted execution environment for accelerators: There have been several studies [24, 27, 29, 35, 53, 56, 57] that extend CPU-based trusted execution environments (TEEs) such as Intel SGX [9] or ARM TrustZone [38] to accelerators such as GPUs or NPUs. *Graviton* [53] and *HIX* [27] have been proposed for GPU TEEs. *HIX* extends the TEE to GPUs by protecting the I/O path between the CPU and GPU without hardware modifications [27], whereas *Graviton* provides a GPU TEE by substituting untrusted GPU driver operations with a trusted command processor [53].

Efficient memory protection mechanisms: There have been many studies to minimize the performance impact of secure memory on CPU, and accelerators including GPUs, and NPUs. [1, 34, 35, 41, 49, 56, 57]. *VAULT* [49] and *Morphable Counters* [41] propose compact integrity tree structures compared to intel SGX by increasing the arity of each tree node. Since replay-attack protection requires additional memory requests up to the root of an integrity tree, lowering the height of the tree reduces the overhead. *Common Counters* [35] proposes compressed encryption counters by exploiting the uniformity of memory updates during GPU application execution. *PSSM* [57] proposes partitioned security metadata structures tailored for GPU sectored cache design to prevent fetching unnecessary data for the security check. *Plutus* [1] finds the data similarity and proposes a value-based verification mechanism to effectively reduce the MAC traffic. Recent work [34] proposes a dynamic dual-granular MAC management to minimize the overhead of secure GPU communication on multi-GPU systems.

With the increasing demand for memory expansion, various memory protection studies have been proposed for the CXL (Compute Express Link) memory environment [8, 13, 44]. *ShieldCXL* supports data protection for CXL memory by employing tamper-responding memory sealing and CXL flit-granular protection [8]. *Toleo* addresses replay-attack protection in a large CXL memory domain by leveraging smart memory to reduce integrity tree traversal [13]. *TAROT* mitigates row-hammer attacks in CXL memory by utilizing multi-bit error detection and offloading this mitigation solution to a smartNIC (smart network interface card) device [44].

7 Conclusion

The paper proposes a unified and efficient memory protection technique for the SoC heterogeneous system with CPU, GPU, NPU, and shared memory. Workloads in each processing unit are mixed in the shared security engine, aggravating the memory protection overhead. However, prior studies are not independently used due to the scalability weakness. The paper proposes the merged MACs and their relocation to remove fragmentations. The paper also suggests the multi-granular tree mechanism to store the multi-granular counters with shortened integrity validation paths. With the multi-granular MACs and the multi-granular integrity tree, the paper explains the unified memory protection scheme to support dynamically detected multi-granularity. Our proposed mechanism tightly reduces the security metadata burden, improving the performance of the secure SoC heterogeneous system.

Acknowledgments

This work was supported by National Research Foundation of Korea (NRF; RS-2024-00347114), and Institute of Information & communications Technology Planning & Evaluation (IITP) funded by the Ministry of Science and ICT, Korea (RS-2024-00402898). This work was also partly supported by Samsung Electronics Co., Ltd. (IO201209-07864-01).

References

[1] Rahaf Abdullah, Huiyang Zhou, and Amro Awad. 2023. Plutus: Bandwidth-efficient memory security for GPUs. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[2] AMD. 2017. AMD APP SDK 3.0 getting started.

[3] Apple. 2020. *Apple unleashes M1.* https://www.apple.com/newsroom/2020/11/apple-unleashes-m1

[4] Apple. 2024. *Apple platform security: Secure enclave.* https://support.apple.com/et-ee/guide/security/sec59b0b31ff/web

[5] Arm. 2019. *Powering the edge: Driving optimal performance with Ethos-N77 processor.* Technical Report.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[7] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *International Symposium on Workload Characterization (IISWC)*.

[8] Kwanghoon Choi, Igjae Kim, Sunho Lee, and Jaehyuk Huh. 2025. ShieldCXL: A practical obliviousness support with sealed CXL memory. In *Transactions on Architecture and Code Optimization (TACO)*.

[9] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. In *IACR Cryptology ePrint Archive*.

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

[11] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Workshop on General Purpose Computation on Graphics Processing Units (GPGPU)*.

[12] Michael Ditty. 2022. Nvidia Orin system-on-chip. In *Symposium on High Performance Chips (Hot Chips)*.

[13] Juechu Dong, Jonah Rosenblum, and Satish Narayanasamy. 2024. Toleo: Scaling freshness to tera-scale memory using CXL and PIM. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[14] Ergun Eroglu. 2013. An application of network simplex method for minimum cost flow problems. In *Balkan Journal of Mathematics*.

[15] Erhu Feng, Dong Du, Yubin Xia, and Haibo Chen. 2023. Efficient distributed secure memory with migratable merkle tree. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[16] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable memory protection in the PENGLAI enclave. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[17] Alexander Freij, Huiyang Zhou, and Yan Solihin. 2021. Bonsai Merkle Forests: Efficiently achieving crash consistency in secure persistent memory. In *International Symposium on Microarchitecture (MICRO)*.

[18] Nathan Gober, Gino Chacon, Lei Wang, Paul V Gratz, Daniel A Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The championship simulator: Architectural simulation for education and competition. In *ArXiv Preprint arXiv:2210.14324*.

[19] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar)*.

[20] Husheng Han, Xinyao Zheng, Yuanbo Wen, Yifan Hao, Erhu Feng, Ling Liang, Jianan Mu, Xiaqing Li, Tianyun Ma, Pengwei Jin, Xinkai Song, Zidong Du, Qi Guo, and Xing Hu. 2024. TensorTEE: Unifying heterogeneous TEE granularity for efficient secure collaborative tensor computing. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[22] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *International Conference on World Wide Web (WWW)*.

[23] Weizhe Hua, Muhammad Umar, Zhiru Zhang, and G Edward Suh. 2020. GuardNN: Secure accelerator architecture for privacy-preserving deep learning. In *Design Automation Conference (DAC)*.

[24] Weizhe Hua, Muhammad Umar, Zhiru Zhang, and G Edward Suh. 2022. MGX: Near-zero overhead memory protection for data-intensive accelerators. In *International Symposium on Computer Architecture (ISCA)*.

[25] Soojin Hwang, Sunho Lee, Jungwoo Kim, Hongbeen Kim, and Jaehyuk Huh. 2023. mNPUsim: Evaluating the effect of sharing resources in multi-core NPUs. In *International Symposium on Workload Characterization (IISWC)*.

[26] Intel. 2022. *Intel trust domain extensions.* Technical Report.

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

[28] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *European Conference on Computer Systems (EuroSys)*.

[29] Sunho Lee, Jungwoo Kim, Seonjin Na, Jongse Park, and Jaehyuk Huh. 2022. TNPU: Supporting trusted execution with tree-less integrity protection for neural processing unit. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[30] Sunho Lee, Seonjin Na, Jungwoo Kim, Jongse Park, and Jaehyuk Huh. 2022. Tunable memory protection for secure neural processing units. In *International Conference on Computer Design (ICCD)*.

[31] Shiwei Liu, Decebal Constantin Mocanu, Yulong Pei, and Mykola Pechenizkiy. 2021. Selfish sparse RNN training. In *International Conference of Machine Learning (ICML)*.

[32] Ditty Michael, Karandikar Ashish, and Reed David. 2018. NVIDIA's Xavier SoC. In *Symposium on High Performance Chips (Hot Chips)*.

[33] Naveen Muralimanohar, Balasubramonian Rajeev, and Norman P. Jouppi. 2009. *CACTI 6.0: A tool to model large caches.* Technical Report.

[34] Seonjin Na, Jungwoo Kim, Sunho Lee, and Jaehyuk Huh. 2024. Supporting secure multi-GPU computing with dynamic and batched metadata management. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[35] Seonjin Na, Sunho Lee, Yeonjae Kim, Jongse Park, and Jaehyuk Huh. 2021. Common Counters: Compressed encryption counters for secure GPU memory. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[36] NVIDIA. 2018. *NVIDIA primer.* http://nvdla.org/primer.html

[37] R Dave Omkar and M Aarthy. 2014. ASIC implementation of 32 and 64 bit floating point ALU using pipelining. In *International Journal of Computer Applications (IJCA)*.

[38] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A comprehensive survey. In *ACM Computing Surveys (CSUR)*.

[39] Qualcomm. 2022. *Qualcomm SPU270 security target lite.* https://www.tuv-nederland.nl/assets/files/cerfiticaten/2023/02/nscib-cc-0569293-st.pdf

[40] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: Better, faster, stronger. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[41] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, Jose A Joao, and Moinuddin K Qureshi. 2018. Morphable Counters: Enabling compact integrity trees for low-overhead secure memories. In *International Symposium on Microarchitecture (MICRO)*.

[42] Abhishek Sarda, Shubhra Dixit, and Anupama Bhan. 2021. Object detection for autonomous driving using YOLO algorithm. In *International Conference on Intelligent Engineering and Management (ICIEM)*.

[43] Marwa Sharaf, Ezz El-Din Hemdan, Ayman El-Sayed, and Nirmeen A. El-Bahnasawy. 2022. A survey on recommendation systems for financial services. In *Multimedia Tools and Applications*.

[44] Chihun Song, Michael Jaemin Kim, Tianchen Wang, Houxiang Ji, Jinghan Huang, Ipoom Jeong, Jaehyun Park, Hwayong Nam, Minbok Wi, Jung Ho Ahn, and Nam Sung Kim. 2024. TAROT: A CXL SmartNIC-based defense against multi-bit errors by row-hammer attacks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[45] Jinook Song, Yunkyo Cho, Jun-Seok Park, Jun-Woo Jang, Sehwan Lee, Joon-Ho Song, Jae-Gon Lee, and Inyup Kang. 2019. 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)*.

[46] Standard Performance Evaluation Corporation (SPEC). 2020. *SPEC CPU 2017.* https://www.spec.org/cpu2017

[47] Tomasz Stęclik, Rafał Cupek, and Marek Drewniak. 2022. Stream data clustering for engineering applications a use case of autonomous guided vehicles. In *International Conference on Big Data (BigData)*.

[48] Yifan Sun, Trinayan Baruah, Saiful A Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, et al. 2019. MGPUSim: Enabling multi-GPU performance modeling and optimization. In *International Symposium on Computer Architecture (ISCA)*.

[49] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[50] Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, and Gagandeep Sachdev, S. 2020. Compute solution for Tesla's full self-driving computer. In *IEEE Micro*.

[51] Artur F Tomeczek and Tomasz M Napiórkowski. 2024. PageRank and regression as a two-step approach to analysing a network of Nasdaq firms during a recession: Insights from minimum spanning tree topology. In *Gospodarka Narodowa. The Polish Journal of Economics*.

[52] Muhammad Umar, Weizhe Hua, Zhiru Zhang, and G Edward Suh. 2022. Softvn: Efficient memory protection via software-provided version numbers. In *International Symposium on Computer Architecture (ISCA)*.

[53] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted execution environments on GPUs. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[54] Yuanchao Xu, James Pangia, Chencheng Ye, Yan Solihin, and Xipeng Shen. 2024. Data Enclave: A data-centric trusted execution environment. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[55] Bo Yu, Wei Hu, Leimeng Xu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. 2020. Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations. In *International Symposium on Microarchitecture (MICRO)*.

[56] Shougang Yuan, Amro Awad, Ardhi Wiratama Baskara Yudha, Yan Solihin, and Huiyang Zhou. 2022. Adaptive security support for heterogeneous memory on GPUs. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[57] Shougang Yuan, Yan Solihin, and Huiyang Zhou. 2021. PSSM: Achieving secure memory for GPUs with partitioned and sectored security metadata. In *International Conference on Supercomputing (ICS)*.

[58] Taesub Yun, Deokjong Jeong, and Sunyoung Park. 2019. "Too central to fail" systemic risk measure using PageRank algorithm. In *Journal of Economic Behavior & Organization*.