ABSTRACT

YUAN, SHOUGANG. Bandwidth-Efficient Secure Memory Designs for GPUs. (Under the direction of Dr. Huiyang Zhou).

Wide adoption of cloud computing makes privacy and security a primary concern. Although recent CPUs have integrated secure memory architecture, such support is still missing for GPUs, a key accelerator in data centers.

In the first work, we explore two secure memory architectures, counter-mode encryption and direct encryption, for GPUs, and show that we need to architect secure memory differently from it for CPUs. Our in-depth study reveals the following insights. First, as GPUs are designed for high-throughput computation, its secure memory needs to deliver high bandwidth. Second, with counter-mode encryption, the memory traffic resulting from the metadata, i.e., the counters, MACs (message-authentication codes), and integrity tree, may cause significant performance degradation, even in the presence of metadata caches. Third, the sectored cache structure adopted by GPUs leads to multiple sequential accesses to the same metadata cache line, which necessitates the use of MSHRs (miss-status handling registers) for meta-data caches. Fourth, unlike CPUs, separate/partitioned metadata caches perform better than unified metadata caches on GPUs. The reason is that GPU workloads feature streaming accesses, which cause severe contention in the unified metadata cache and the cached counters and integrity tree nodes may be evicted before being reused. Fifth, the massive-threaded nature of GPUs make them latency-tolerant and the performance impact due to the extra encryption/decryption latency is limited. As a result, direct encryption can be a promising alternative for GPU secure memory. The challenge, however, lies in memory integrity verification as the integrity tree may incur high storage overhead and metadata traffic.

In the second work, we point out that conventional CPU secure memory architecture can not be directly adopted to the GPUs. The key reasons include: (1) accessing the security metadata, including encryption counters, message authentication codes (MACs) and integrity trees, requires significant memory bandwidth, which may lead to severe bandwidth competition with normal data accesses and degrade the GPU performance; (2) contemporary GPUs use partitioned memory organization, which results in storage and coherence problems for encryption counters and integrity trees since different partitions may need to update the same counter/integrity tree blocks; and (3) the existing split-counter block organization is not friendly to sectored caches, which are commonly used in GPU for band-

width savings. Based on these observations, we propose partitioned and sectored security metadata (PSSM), which has two components: (a) using the offset addresses (referred to as local addresses) within each partition, instead of the virtual or physical addresses, to generate the metadata so as to solve the counter or integrity tree storage and coherence problems and (b) reorganizing the security metadata to make them friendly to the sectored cache structure so as to reduce the memory bandwidth consumption of metadata accesses. With these proposed schemes, the performance overhead of secure GPU memory is reduced from 59.22% to 16.84% on average. If only memory encryption is required, the performance overhead is reduced from 29.53% to 5.18%.

In the third work, we analyze the security guarantees that used to defend against physical attacks, and make the observation that heterogeneous GPU memory system may not always need all the security mechanisms to achieve the security guarantees. Based on the memory types as well as memory access patterns either explicitly specified in the GPU programming model or implicitly detected at run time, we propose adaptive security memory support for heterogeneous memory on GPUs. Specifically, we first identify the read-only data and propose to only use MAC (Message Authentication Code) to protect their integrity. By eliminating the freshness checks on read-only data, we can use a single counter for such data regions and remove the corresponding part in the Bonsai Merkel Tree (BMT), thereby reducing the traffic due to counters and BMT. Second, we detect the common streaming data access patterns and propose coarse-grain MACs for such stream data to reduce the MAC access bandwidth. With the hardware-based detection of memory type (read-only or not) and memory access patterns (streaming or not), our proposed approach adapts the security support to significantly reduce the performance overhead. Our evaluation using both memory-intensive and computation-intensive workloads shows that our scheme can achieve secure memory on GPUs with low overheads for memory-intensive workloads while not affecting computation-intensive workloads. Among the fourteen memory-intensive workloads in our evaluation, our design reduces the performance overheads of secure GPU memory from 53.9% to 10.2% on average. Compared to the state-of-the-art secure memory designs for GPUs [Na21; Yua21a], our scheme outperforms PSSM by up to 36.8% and 10.4% on average and outperforms Common counters by 77.5% on average for memory-intensive workloads.

In the fourth work, we focus on encryption counters given their impact on the counter and BMT traffic while leveraging prior schemes [Sai18; Taa18a] to address the MAC traffic. We first analyze the characteristics of the encryption counters from a wide range of GPGPU benchmarks and make two key observations. (1) With the split counter scheme, the cache

blocks in a large portion of the memory space, sometimes the entire GPU memory space, share the same major counter value. (2) The difference among minor counters is fairly limited. We then propose a novel scheme to reduce the encryption counter traffic. Our design includes (a) a highly compact way of counter representation and (b) a verification scheme to determine the correct minor counter values. In our design, we use a few on-chip registers to hold the major counters and use a (7-bit) base value along with two small (2-bit) deltas to represent the minor counters in a large memory chunk, one delta for the most frequent delta between minor counters and the base, the other delta for the maximal difference between a minor counter and the base. This way, for a large memory chunk (e.g., 16kB), the counter overhead becomes nearly negligible (less than 2B). We then leverage the existing MAC verification logic to verify the minor counters computed from the base and deltas. Our approach essentially trades off decryption and integrity-check latency for reduced counter-data traffic to take advantage of the latency-hiding nature of GPUs. Compared to prior works on reducing counter traffic [Na21], our scheme handles more counter value patterns (as we don't restrict the counters to be the same in a memory chunk) and is more effective in reducing counter traffic. Our study also reveals that the GPU memory data are typically compressible. As a result, we can co-locate the MACs with the compressed cache blocks, similar to [Taa18a]. Our experimental results show that our proposed delta counter scheme significantly reduces the storage and bandwidth overheads of encryption counters and achieves secure GPU memory with an average performance overhead of 2.01% compared to GPU without security support. Our delta scheme is also compatible with SYNERGY [Sai18], which leverages ECC chips to store MACs, and our achieved performance overhead is 2.83%.

© Copyright 2023 by Shougang Yuan

All Rights Reserved

Bandwidth-Efficient Secure Memory Designs for GPUs

by
Shougang Yuan

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina
2023

APPROVED BY:

Dr. James Tuck	Dr. Xu Liu
Dr. Amro Awad	Dr. Huiyang Zhou Chair of Advisory Committee

BIOGRAPHY

The author received his bachelor's degree of Information Management and Information System from Northwest A&F University in 2015, and his master's degree of Computer Science from Xi'an Jiaotong University in 2018. He started his Ph.D. program in North Carolina State University in 2018. His research mainly focused on secure memory architectures for non-volatile memory and GPU memory.

ACKNOWLEDGEMENTS

I would like to express my gratitude to Dr. Huiyang Zhou, Dr. Amro Awad, Dr. James Tuck, and Dr. Xu Liu for serving on my committee, and thank Dr. Yan Solihin for his help and guidance during my PhD study.

TABLE OF CONTENTS

List of Tables . . . . . **vii**

List of Figures . . . . . **viii**

Chapter 1 INTRODUCTION AND BACKGROUNDS **1**
1.1 Introductions 1
1.2 Background 2
1.2.1 Threat Model and Scope of Work 2
1.2.2 Security Mechanisms 3
1.3 GPU Security 5

Chapter 2 Analyzing Secure Memory Architecture for GPUs **6**
2.1 Introduction 6
2.2 Methodology 8
2.3 Counter-Mode Encryption 10
2.3.1 Performance Overhead 10
2.3.2 MSHRs for Metadata Caches 13
2.3.3 Metadata Cache Size 14
2.3.4 Unified vs. Separate Metadata Caches 16
2.3.5 AES Engine Throughput 18
2.3.6 Die Area 19
2.4 Direct Encryption 22
2.4.1 Performance Overheads of Direction Encryption 22
2.4.2 Direct Encryption **vs.** Counter-mode Encryption 24
2.4.3 Integrity Protection 24
2.5 Conclusions 26

Chapter 3 PSSM: Achieving Secure Memory for GPUs with Partitioned and Sectored Security Metadata **27**
3.1 Motivations 27
3.1.1 Performance Impacts of Naive Design 27
3.1.2 Problem Diagnosis 29
3.1.3 Coarse-Grain Interleaving 30
3.1.4 Sectored MDC 31
3.1.5 Sectored Data Cache and MAC Verification 33
3.2 Architecture Design 35
3.2.1 Overall Architecture 35
3.2.2 Using Local Addresses for Security Metadata 36
3.2.3 Making Metadata Friendly to Sectored Caches 37
3.2.4 Encryption and MAC Engine 39
3.2.5 Bandwidth for Accessing MACs 40

3.3 Evaluation . . . 41
3.3.1 Methodology . . . 41
3.3.2 Performance . . . 42
3.4 Conclusions . . . 47

Chapter 4 SHM: Adaptive Security Support for Heterogeneous Memory on GPUs 48
4.1 Motivation and Design Principles . . . 48
4.1.1 Heterogeneous Memory on GPUs . . . 48
4.1.2 Seed Generation in Counter-Mode Encryption . . . 49
4.1.3 Overhead of MAC Accesses . . . 51
4.2 Architecture Design . . . 51
4.2.1 Overall Architecture . . . 51
4.2.2 Detecting Read-only Regions . . . 53
4.2.3 Detecting Streaming Accessed Chunks . . . 56
4.2.4 Using L2 as Victim Cache for Security Metadata . . . 59
4.3 Methodology . . . 60
4.3.1 Hardware Overheads . . . 63
4.4 Evaluation . . . 63
4.4.1 Read Only Prediction . . . 63
4.4.2 Streaming Access Pattern Detection . . . 64
4.4.3 Overall Performance . . . 65
4.4.4 Performance Breakdown . . . 66
4.4.5 Bandwidth Saving . . . 67
4.4.6 Power Saving . . . 69
4.4.7 Using L2 as a Victim Cache . . . 69
4.5 Conclusions . . . 70

Chapter 5 Delta Counter: Bandwidth-Efficient Encryption Counter Representation for Secure GPU Memory . . . 71
5.1 Summary . . . 71
5.2 Motivations . . . 72
5.2.1 Major Counter Analysis . . . 73
5.2.2 Minor Counter Analysis . . . 74
5.2.3 MAC Bandwidth Requirement . . . 76
5.3 Architecture Design . . . 77
5.3.1 Overall Architecture . . . 77
5.3.2 Management of Major Counter Registers . . . 77
5.3.3 Delta Counter Entry . . . 81
5.3.4 Dataflow of DRAM Read and Write . . . 82
5.3.5 Delta Counter Cache Management . . . 82
5.3.6 Counter Cache and BMT Verification . . . 83
5.3.7 Data Compressibility . . . 84
5.4 Methodology . . . 86

5.5 Evaluation 88
5.5.1 Overall Performance 88
5.5.2 Performance breakdown 88
5.5.3 Upper Bound Analysis 92
5.6 Conclusions 92

Chapter 6 CONCLUSIONS AND FUTURE WORKS 94

LIST OF TABLES

Table 2.1 Baseline GPU configuration 8
Table 2.2 Metadata organization and storage 9
Table 2.3 Metadata cache organization 9
Table 2.4 Benchmarks 10
Table 2.5 Evaluated designs for counter-mode encryption 11
Table 2.6 Die area of the AES engine 20
Table 2.7 Scaled down die area of the AES engine and caches 20
Table 2.8 Evaluated designs for direct encryption 22

Table 3.1 Baseline GPU Configuration 41
Table 3.2 MDC and MEE Organization 41
Table 3.3 Benchmarks 42
Table 3.4 Evaluated designs for GPU secure memory with both memory encryption and integrity verification. 43
Table 3.5 Evaluated designs for GPU memory encryption. 43

Table 4.1 Security Mechanisms for GPU Heterogeneous Memory 49
Table 4.2 Security Mechanisms for Application Data 49
Table 4.3 Handling Streaming Predictions for Read Accesses 59
Table 4.4 Handling Streaming Predictions for Write Accesses 60
Table 4.5 Baseline GPU Configuration 61
Table 4.6 MDC and MEE Organization 61
Table 4.7 Benchmarks 61
Table 4.8 Evaluated designs for GPU secure memory with both memory encryption and integrity verification. 62
Table 4.9 Hardware Overhead 63

Table 5.1 Baseline GPU Configuration 86
Table 5.2 MDC and MEE Organization 87
Table 5.3 Benchmarks 87
Table 5.4 Evaluated designs for GPU secure memory with both memory encryption and integrity verification. 89

LIST OF FIGURES

Figure 1.1 Counter-mode encryption and direct encryption 3

Figure 2.1 GPU architecture with secure memory support 7
Figure 2.2 Normalized IPC of counter mode encryption with Bonsai Merkle tree. 12
Figure 2.3 Distribution of different types of memory requests. 13
Figure 2.4 Amount of secondary misses in metadata caches. 14
Figure 2.5 Normalized IPC of secure memory with different numbers of MSHRs in metadata caches. 15
Figure 2.6 Normalized IPC for different metadata cache sizes. 15
Figure 2.7 Normalized IPC of unified metadata caches vs separate metadata caches. 17
Figure 2.8 Miss rates for different types of metadata in unified vs. separate metadata caches. 17
Figure 2.9 Reuse distance of counters of the benchmark fdtd2d 18
Figure 2.10 Reuse distance of MACs of the benchmark fdtd2d 18
Figure 2.11 Normalized IPC with different numbers of AES engines in each memory partition 19
Figure 2.12 Normalized IPC with different L2 cache capacities 21
Figure 2.13 L2 cache miss rate 21
Figure 2.14 Normalized IPC of direct encryption with different encryption latencies. 22
Figure 2.15 Normalized IPC of direct encryption and counter-mode encryption 23
Figure 2.16 Normalized IPC of direct encryption and counter-mode encryption with integrity protection 25

Figure 3.1 Normalized performance of secure memory designs to the baseline GPU without secure memory support. 28
Figure 3.2 A single (128B) counter block corresponds to 128 data blocks in 32 partitions. Similarly, one BMT node requires multiple counter blocks. 29
Figure 3.3 The IPC of different page interleaving granularities, normalized to the baseline GPU with 256B interleaving and without secure memory support. 31
Figure 3.4 Performance comparison of secure GPU with non-sectored MDCs (labeled 'secureMem') and sectored MDCs (labeled 'sec_mdc'). 32
Figure 3.5 A Split-counter block of 128B, containing 1 × 128-bit major counter and 128 × 7-bit minor counters. When split into 4 sectors, the first sector contains the major counter and some minor counters. 33
Figure 3.6 The IPC of a non-sectored L2 cache with different numbers of request merges in an L2 MSHR normalized to the baseline sectored L2. 34
Figure 3.7 The L2 MSHRs. (a) The structure of an L2 MSHR. (b) The MSHR state of a sectored L2 after the access sequence A1-A5. (c) The MSHR state of a non-sectored L2 cache after the same access sequence A1-A5. . . 35

Figure 3.8 Secure GPU architecture with the trust boundary as the GPU chip . . 36
Figure 3.9 Physical and local addresses in partitioned memory organization. . . 37
Figure 3.10 Sectored split-counter design: each sector has 1×32-bit major counter and 32 × 7-bit minor counters. 38
Figure 3.11 Numbers of stores per kilo instructions (SPKI). 38
Figure 3.12 The encryption/decryption process in PSSM. The input to the AES encryption engine ensures encryption seed uniqueness, both temporally and spatially. 39
Figure 3.13 The MAC generation process in PSSM. The MAC computation output is truncated to 64/32 bits. Sector id is used when a MAC is generated for each sector. 40
Figure 3.14 Normalized IPC of different secure GPU memory designs. 44
Figure 3.15 MDC miss rates (in MPKI) of the PSSM_nL2_4B_sMdc design. 45
Figure 3.16 Normalized IPC of the PSSM_nL2_4B_sMdc design with different ideal MDCs. 46
Figure 3.17 Normalized IPC of different GPU memory encryption schemes. . . . 46

Figure 4.1 Seed generation for (a) not-read-only data and (b) read-only data. . . 49
Figure 4.2 Integrity tree with read-only regions excluded. 50
Figure 4.3 The ratio of memory accesses (i.e., L2 misses and L2 write backs) accessing streaming data as well as read-only data in various GPU workloads. 52
Figure 4.4 Overall architecture. 53
Figure 4.5 The read-only detector and streaming detector in a memory partition. Their inputs are the LLC misses and write backs. 56
Figure 4.6 An example showing the propagation from the shared counter to the per block counters. 56
Figure 4.7 The process of shared counter update when using the read only reset API. 56
Figure 4.8 Breakdown of read-only predictions. 64
Figure 4.9 Breakdown of streaming pattern predictions 65
Figure 4.10 Normalized IPC of different secure GPU memory designs. 66
Figure 4.11 Performance impacts of different optimizations. 67
Figure 4.12 Bandwidth overheads due to security metadata, normalized to regular data bandwidth, of different designs. 68
Figure 4.13 Normalized Energy Consumption per Instruction for different designs. 69
Figure 4.14 Normalized IPC when enabling L2 as a victim cache for security metadata. 70

Figure 5.1 Number of Unique Major Counters 73
Figure 5.2 Distribution of delta value that less than 4. 75
Figure 5.3 Average ratio of the most common counter value in a 16kB region of which the delta among minor counters is less than 4. 75

Figure 5.4 Data and MAC layout. 76
Figure 5.5 Overall architecture. 78
Figure 5.6 Major Counter Entry and Layout of Per-Block Counter. 79
Figure 5.7 An example of major counter register management 79
Figure 5.8 Handling of Minor Counter Overflow 80
Figure 5.9 Delta counter representation and associated access patterns. 81
Figure 5.10 DRAM flow for memory read and write. 82
Figure 5.11 Delta Counter Management 83
Figure 5.12 Counter Cache Management 84
Figure 5.13 Compression Cache Management 85
Figure 5.14 Overall Performance of Different Designs 90
Figure 5.15 Ratio of Counter Traffic in Memory. 91
Figure 5.16 Ratio of Compressible Memory Access. 91
Figure 5.17 Distribution of Number of Encryptions that Each Memory Access Needs to Try. 92

CHAPTER

1

INTRODUCTION AND BACKGROUNDS

1.1 Introductions

Cloud computing has become the predominant computing paradigm. With the cloud being a shared resource, it is critical to provide users with sufficient privacy and security guarantees. Toward this end, hardware-based trusted execution environments (TEEs), such as Intel SGX [Gue16b; Cor19] and ARM TrustZone [Pin19], have been integrated onto the CPUs to provide secure isolated computing environment for cloud users. TEEs can protect against both compromised system software, e.g., hypervisors and operating system (OS), and physical attacks such as memory tampering [Gue16b]. A critical building block for TEE is *secure memory engine*, which keeps data in memory encrypted [Gue16a; Kap16] and protects its integrity [Gue16a]. However, such secure memory architecture support is missing on GPUs, a key accelerator in clouds for a wide range of workloads including machine learning, scientific computing, 3D rendering, etc. As the system security is determined by its weakest link, we argue that accelerators such as GPUs also need to provide TEE.

Recognizing the needs, some recent works, including Graviton [Vol18] and HIX [Jan18],

tried to provide TEE for the workloads offloaded to GPUs. Graviton assumes that the system software stack, including GPU drivers, the OS, and hypervisor, cannot be trusted and the attackers have physical access to the hardware. To protect against software attacks, the GPU management operations are offloaded to the GPU command processors instead of being relegated to the GPU drivers, which runs within the untrusted host kernel space. Furthermore, Graviton provides new primitives, such as secure memory copy, to prevent attackers from snooping upon the PCIe bus. HIX, alternatively, mainly focuses on protecting the I/O path between the hardware and software. Although it also keeps the GPU drivers out of the trust boundary by isolating it from the kernel space, it relies on the secure CPU enclaves to protect the refactored GPU drivers. However, in these previous works, the threat model is weaker than conventional CPU TEE, as they do not provide secure memory support and require system software (e.g., GPU driver) to be trusted.

In this dissertation, we explore the design space and implication of supporting hardware based TEEs for GPUs. We start our investigation with a detailed performance analysis for GPU secure memory designs, and identified that the memory bandwidth competition between the regular data and security metadata is the major performance bottleneck. In our second work, we propose a simple yet effective scheme, called PSSM (Partitioned and Sectored Security Metadata), to alleviate the memory bandwidth pressure of security metadata access. In our third work, we analyze the different security mechanisms in conventional CPU TEEs, and argue that heterogeneous GPU memory may not always need all the security guarantees as conventional CPU TEEs. And in our last work, we analyze the encryption counter behaviors, and propose an bandwidth efficient encryption counter representation, namely, delta counter, for secure GPU memory.

1.2 Background

1.2.1 Threat Model and Scope of Work

The threat model for CPU TEE, such as Intel SGX, assumes two types of threats: compromised system/privileged software (such as the OS and hypervisor) and attackers that have physical access to the remote server and the abilities to snoop/scan and modify data stored in off-chip memory. CPU TEE assumes that the processor chip provides a security boundary, where all on-chip components are assumed to be out of the reach of attackers. In general, TEE requires three major architecture supports: *hardware key management*, *attestation*, and *secure memory engine* (which encrypts and protects the integrity of data stored off the

processor chip) [Lie00; Suh07; Yan03; Yan06]. Of these, secure memory incurs the largest performance overheads as it must be active at all time and affects the critical path of load instructions. Due to the enormity of the overheads, recent attempts to extend CPU TEE to GPUs [Jan18; Vol18] ignore the physical attack threats and enlarge the trust base (e.g., adding GPU memory module to the trust base). In contrast, this research assumes the same threat model in CPU TEE also affects GPUs, and assumes that GPU chip provides the security boundary, where all the data stored in the on-chip resources such as registers and caches are safe. The attackers may have physical access to the GPUs hardware, and have the capability to snoop the GPU memory buses or to scan/tamper the GPU memory content.

The scope of this paper covers the design space of secure memory for GPUs. Hardware key management and attestation have been well studied in previous work (i.e., Graviton [Vol18] and HIX [Jan18]) and are beyond the scope of this paper. Securing the communication channel between CPU and GPU is also outside the scope of this work. Existing solutions for that may include PCI modification [Jan18], or for tighter integration, secure cache coherence protocol [Rog08a]. Furthermore, side channel attacks such as timing-based side channel attack [Jia16] are also out of the scope of our work.

1.2.2 Security Mechanisms

Memory encryption. The goal of memory encryption is to protect data confidentiality [Lee05; Rog08b; Rog06; Zha05]. The cryptographic hardware engines residing in the memory controller are responsible for performing encryption/decryption before the data is sent to/returned from the off-chip memory. Generally, there are two approaches for memory encryption as shown in Fig. 1.1: counter-mode encryption and direct encryption.

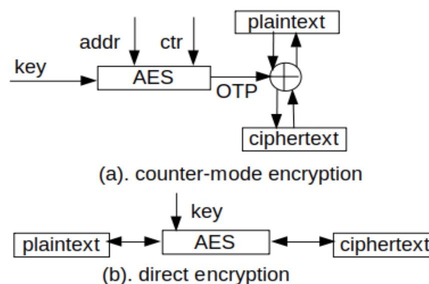

Figure 1.1: Counter-mode encryption and direct encryption

Counter-mode encryption can hide the decryption latency by overlapping it with memory reads, and thus offload the decryption latency from the critical path of load instructions. In counter-mode encryption, there is a counter associated with each cache line. When the memory controller is fetching the data from off-chip memory, the corresponding counter and the block address are used to generate a one-time pad (OTP) using the encryption engine such as AES. The memory controller can recover the plaintext by XORing the data and OTP in one cycle and then supply the data to processors. To guarantee the security, the counters cannot be reused. Hence, at each dirty eviction from the last level cache (LLC), the counter corresponding to the cache line will be incremented. Since the encryption strength of counter-mode memory encryption is conditional upon not reusing the counters, if the attacker can trigger reuse of counters, the encryption will be broken [Yan06; Rog07]. Therefore, counter-mode encryption *fundamentally relies* on counter integrity protection to provide data confidentiality.

Direct encryption can also be used for the security purpose. In contrast to counter-mode encryption, with direct encryption, confidentiality does not necessarily require integrity protection. The main disadvantage of direct encryption is that it exposes the latency of decryption to the critical path of memory reads, because decryption can only start after the data is fetched from memory [Suh03].

Memory Integrity Verification. As discussed earlier, counter-mode encryption requires integrity protection of counters [Rog07]. Furthermore, MACs are needed to protect against memory tampering attack [Yan06; Zou19]. As pointed out by Rogers et. al [Rog07], stateful MACs calculated from the ciphertext, block address and the corresponding counter can provide data integrity protection, and the Merkle tree (MT) only needs to cover the counters to protect against counter-replay attacks. This scheme is named BMT and can help to reduce the storage overheads of the MT. Moreover, the BMT is much shallower than the original MT. Any modification to the data or MACs stored in the off-chip memory can be detected by comparing the MACs when the data is fetched from memory. Similarly, any modification or replay of the counters can be detected by traversing the BMT and comparing the hash values in the BMT nodes. In contrast, with direct encryption, encryption protection is not dependent on integrity verification; and integrity verification provides additional protection against memory tampering and replay attacks. Integrity protection relies on (1) MAC of the ciphertext and (2) a Merkle/Hash Tree based on the entire memory with a special on-chip register storing the root of the tree [Gas03].

1.3 GPU Security

Recent studies highlight the needs for security support within accelerators. Deep learning (DL) is such an application that attracts special attention [Liu17]. Several recent works [Zuo20; Hua20b; Hua20a] aim to extend the existing memory encryption and integrity verification schemes to DL accelerators.

The work [Zuo20] by Zuo et al. pointed out that the gap between the throughput of AES engine and the GPU memory bandwidth is the key bottleneck for secure GPU memory. To reduce the overhead of encryption, they proposed: (1) selective memory encryption, (2) co-location of data and the corresponding counter. In comparison, our work differs from this prior work in the following ways. First, our work consider data integrity protection, which is missing in this prior work. Second, we perform a detailed study on metadata cache designs, including the impact from the sectored L2 caches as well as unified metadata caches and separate metadata caches. Third, we exploit pipelined AES engines to overcome the AES throughput limitation and explore the design trade-offs for different AES throughput. Fourth, the proposed solution by Zuo et al. [Zuo20] is specific to DL while our work covers a wide range of workloads.

Similar to Zuo's work [Zuo20], Hua et al. [Hua20b; Hua20a] aimed to provide memory protection for the deep learning workloads running on the accelerators. As mentioned in [Hua20b], the overheads of memory encryption and integrity verification come from the extra memory accesses generated by the security metadata. In their system design, they proposed to provide memory encryption and integrity verification based on a coarse granularity of memory objects. Their key observation is that accelerators often explicitly move data between on-chip memory and off-chip DRAM at the object granularity. In comparison, our work provides detailed performance analysis of both counter-mode encryption and direct encryption and explore different metadata cache design choices.

Other works, including HIX [Jan18] and Graviton [Vol18] that aimed to provide the TEE for GPUs, mainly focused on protecting the user program and data from untrusted OS or privileged malicious software (hypervisor), and did not consider the physical attacks, i.e., memory scanning and tampering attacks. In other words, these two works did not provide hardware-based memory encryption and integrity verification protections. The data stored on GPU off-chip memory is plaintext and there is no support for integrity checks. In comparison, our work defines a stronger threat model, and provides detailed performance study on memory encryption and integrity verification for GPUs.

CHAPTER

2 ANALYZING SECURE MEMORY ARCHITECTURE FOR GPUS

2.1 Introduction

Similar to the CPU TEE (e.g., Intel SGX), the secure memory hardware is placed in the memory controller in our design, as shown in Fig. 2.1. As GPUs incorporate multiple memory controllers to provide high memory bandwidth, the same secure memory hardware is replicated in each memory controller.

We start our investigation with counter-mode encryption and Bonsai Merkle Tree (BMT) [Rog07], which are the state-of-art secure memory architecture on CPUs. Counter-mode encryption is introduced to hide the decryption latency and BMT is used to verify the integrity of data through its counters, which are used to encrypt/decrypt the data along with the secret key. In counter-mode encryption, the security metadata include counters, BMT, and MACs of the ciphertext. To reduce the extra memory accesses for fetching the metadata, on-chip metadata caches are commonly employed. We model the counter-mode encryption and BMT support upon a GPU model based on Nvidia Volta [Jia18]

architecture, and identify that hardware-based secure memory architecture can incur significant performance overhead for GPUs. The main reason is that the metadata data accesses generate a lot of memory traffic even in the presence of metadata caches. Given the high bandwidth requirements of GPUs, the additional memory traffic contends for the memory bandwidth and significantly slows down the GPUs performance. Also, we observe that due to the sectored L2 cache structure in GPUs, the streaming data accesses leads to a high ratio of secondary misses (i.e., misses to the same cache block after it being requested) in the metadata caches, which highlight the importance of the MSHRs to filter out redundant memory requests. Furthermore, due to the nature of high-throughput computation of GPUs, high-throughput cryptographic engines are also needed to produce a balanced system.

Besides counter-mode encryption, we also analyze the alternative design of direct encryption. Without counters, BMT is infeasible. Therefore, we resort to a Merkle Tree (MT) to perform integrity checks. In other words, with direct encryption, the security metadata include MACs and the MT, both of them are used for memory integrity verification. Our analysis reveals the following interesting observations. First, direct encryption itself does not lead to high performance overheads because GPUs are designed to be latency tolerant. Actually, our results show that direct encryption can perform better than counter-mode encryption even with a high encryption latency of 160 cycles. Second, memory integrity protection may become the performance bottleneck due to the memory traffic generated by MAC and MT accesses.

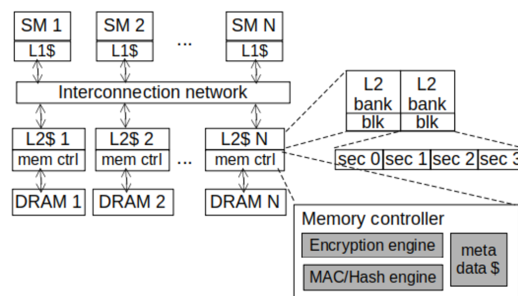

Figure 2.1: GPU architecture with secure memory support

Table 2.1: Baseline GPU configuration

SM config	80 SMs, 1132 MHz
Register File	256KB/SM, 20MB in total
L1 D-Cache	32KB/SM
Shared Memory	96KB/SM
L2 cache	2 banks/partition, 96KB/bank, 6MB in total
DRAM	850MHz, 868GB/s, 32 partitions

2.2 Methodology

We model different secure memory architecture supports using GPGPU-Sim v4.0 [Kha20]. The configuration of our baseline GPU model is shown in Table 2.1, which is based on the Nvidia Volta architecture [Jia18]. In our experiments, we assume that a range of 4GB device memory is protected, and the metadata cache line size is aligned to the data cache line size, which is 128B. The metadata organization and storage overhead are listed in Table 2.2.

For counter-mode encryption, each counter cache line maintains one 128-bit major counter (shared by data blocks within a 16KB memory chunk) and 128 7-bit per block minor counters, thereby covering 128 lines of data. In other words, the ratio between data and counter capacity is 128 and the overall counter blocks take 32MB (=4GB/128) off-chip storage. Using a 64-bit MAC for each 128B data, the overall MAC consumes 256MB storage. As a result of the sectored L2 cache, we use truncated MAC, i.e., 16-bit MAC for each 32B sector. For the BMT, we build a 6-level 16-ary hash tree and its overall capacity is 2.14 MB, excluding the counter blocks that serve as the leaf nodes of the BMT. Therefore, the overall capacity of metadata is 290.14MB (=32+256+2.14 MB) for counter-mode encryption.

With direct encryption, using the same 64-bit MAC for each 128B data, the overall MAC consumes 256MB memory. As the MT is built upon the 4GB data, we use a 7-level 16-ary tree and the overall capacity is 17.1MB, excluding the MACs that serve as the leaf nodes of the MT. Therefore, the overall memory overhead is 273.1MB (=256+17.1 MB) for direct encryption.

The separate metadata caches, which include the counter cache, MAC cache, and BMT/MT cache, are modeled in each memory partition (i.e., each memory controller). The metadata cache specification is listed in Table 2.3. State-of-the-art secure memory architecture in CPUs uses speculative verification and lazy update for BMT/MT [Gas03]. We also adopt these schemes on GPUs. Speculative verification means that the memory

Table 2.2: Metadata organization and storage

Metadata Type	Counter-mode Encryption	Direct Encryption
Counter	128b/16KB, 7b/blk, 32MB	-
MAC	8B/blk, 2B/sector, 256MB	8B/blk, 2B/sector, 256MB
BMT/MT	16 ary, 6 levels, 2.14MB	16 ary, 7 levels, 17.1MB

Table 2.3: Metadata cache organization

Counter cache	{2,4,8,16,32,64}KB/Memory Partition, 2KB default, 128B blk, 64 MSHRs, allocate-on-fill policy
Mac cache	{2,4,8,16,32,64}KB/Memory Partition, 2KB default, 128B blk, 64 MSHRs, allocate-on-fill policy
(Bonsai)Merkle Tree cache	{2,4,8,16,32,64}KB/Memory Partition, 2KB default, 128B blk, 64 MSHRs, allocate-on-fill policy
Unified metadata cache	6KB/Memory Partition, 128B blk, 192 MSHRs, allocate-on-fill policy
Hash/Mac latency	40 cycles default
AES engines	{1,2}/Memory Partition, 2 default.

controller can supply the data to the core before the corresponding integrity check is finished. Later on, if there is a failure in integrity verification, an exception would be raised. Lazy update means that only when a counter block or a tree node is evicted from the counter cache or BMT/MT cache, its parent will be updated in the BMT/MT cache.

In our study, we assume pipelined AES engines and pipelined MAC units. This way, the encryption latency or the MAC computation latency would not affect the throughput. With AES-128, 16B of data can come out of a pipelined AES engine each cycle. With the memory clock frequency of 850MHz, the throughput of the pipelined AES engine is 13.6 GB/s (=16B*850MHz). With 2 AES engines in each memory partition, the throughput would match the memory bandwidth, resulting in a balanced design. Therefore, we use 2 pipelined AES engines in each partition by default. We model different AES latencies and a 40-cycle latency for the MAC unit. With counter-mode encryption, the AES latency does not matter as it is hidden by design.

Our benchmarks are selected from the Rodinia-3.1 [Che09], Parboil [Str09] and polybench [GG09] benchmark suites to cover a wide range of workloads. Table 2.4 lists the details of these benchmarks. For each benchmark, we simulate until it has run for 4 million cycles.

Table 2.4: Benchmarks

Categorization	Benchmark name	Bandwidth utilization	IPC
non memory intensive	heartwall	<1%	1,195.37
	lavaMD	<1%	4,615.23
	nw	<2%	23.90
	b+tree	12%-14%	2,768.61
medium memory intensive	backprop	25%	3,067.61
	cfd	15%-50%	1,076.98
	dwt2d	20%-50%	784.70
	kmeans	40%-45%	97.04
memory intensive	bfs	5%-60%	699.51
	srad_v2	79%- 80%	3,306.82
	streamcluster	78%-80%	1,178.18
	2Dconvolution	53%	2,487.22
	fdtd2d	82%-83%	1,773.95
	lbm	58%	552.12

Table 2.4 also reports the bandwidth utilization and the IPC (instruction-per-cycle) for each benchmark when running on the baseline GPU without secure memory support. In this paper, we categorize the benchmarks which consume more than 50% peak DRAM bandwidth as memory intensive, the benchmarks which consume less than 20% peak DRAM bandwidth as non memory intensive, and the remaining as medium memory intensive.

2.3 Counter-Mode Encryption

In this section, we perform an in-depth study on extending counter-mode encryption and BMT to GPUs. Different design options, as listed in Table 2.5, are explored to reveal the performance bottlenecks.

2.3.1 Performance Overhead

Fig. 2.2 shows the performance of various GPU secure memory models normalized to the baseline GPU without security support. From the figure, we can see that adding secure memory support may incur significant performance overhead. The normalized IPC is reduced by 65.9% on average using the geometric mean (Gmean), and can be up to 91.06% for memory-intensive workloads such as lbm.

To identify the performance bottleneck, we model different idealistic designs. From

Table 2.5: Evaluated designs for counter-mode encryption

Scheme	What It Represents
baseline	Baseline GPU without secure memory support.
secureMem	Baseline GPU with secure memory support using counter-mode encryption and BMT. In Section 2.3.1, no MSHR is modeled, in Section 2.3.2, 2.3.3, 2.3.4, 2.3.5, there are 64 MSHRs for each metadata cache.
secureMem_xMB	secure GPU memory with different L2 capacities upon our baseline GPU with counter-mode encryption and BMT.
0_crypto	secureMem with 0 MAC latency and encryption latency.
perf_mdc	secureMem with perfect metadata caches, i.e., there are no cache misses and write backs.
large_mdc	secureMem with unlimited capacity for metadata caches, meaning that there are only cold misses.
mshr_x	secureMem with different mshr capacity for metadata caches.
separate	secureMem with separate metadata caches in each memory partition/unit.
unified	secureMem with a unified metadata cache, which caches all types of metadata, in each memory partition/unit.

Fig. 2.2, we can see that zero cycle cryptographic operation latency (zero-cycle MAC and AES) does not alleviate the performance overhead. This is expected as GPUs are massively parallel machines and leverage high-degrees of TLP to hide the latency. However, when we model ideal metadata caches or unlimited capacity metadata caches, the GPU performance with secure memory support becomes very close to the baseline. This indicates that the accesses to the security metadata is the performance bottleneck.

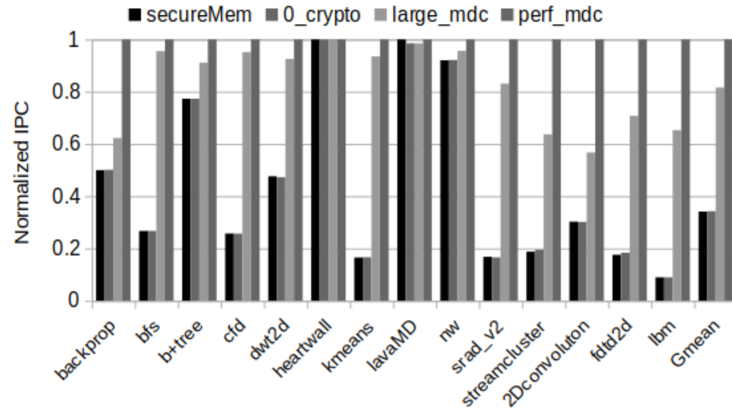

Figure 2.2: Normalized IPC of counter mode encryption with Bonsai Merkle tree.

To further analyze the impact of the metadata accesses, in Fig. 2.3 we compare the amounts of different types of memory requests when secure memory is integrated, i.e., the secureMem model in Table 2.5. The memory-request types include regular data read and write requests (labeled 'data'), counter reads (labeled 'ctr'), MAC reads (labeled 'mac'), BMT reads (labeled 'bmt'), and all the writebacks from the metadata caches (labeled 'wb').

From Fig. 2.3, we can make the following observations. First, for all benchmarks, the memory requests to fetch MACs from off-chip memory account for a major portion of the memory traffic (25.58% on average). Second, the counter requests also account for a large portion of memory traffic (21.77% on average). Third, some benchmarks, including bfs, b+tree, kmeans, nw and lbm, have a relatively high ratio of memory requests for fetching the BMT blocks from memory. The reason is that these benchmarks have relatively high counter-cache miss rates and the fetched counters need to be validated. Fourth, extra memory traffic due to metadata accesses may not necessarily lead to performance degradation. For example, for the non-memory intensive benchmarks (heartwall, lavaMD and nw), the metadata accesses account for a large portion of the memory traffic (66.07%,

62.71%, 75.41%), but the performance impact is near zero. The reason is that for these benchmarks, the memory bandwidth is under-utilized as shown in Table 2.4. Therefore, additional traffic does not result in contention for the memory bandwidth. Fifth, some benchmarks, including bfs, dwt2d and lbm, have relatively high amounts of metadata-cache writebacks, which also impact the performance.

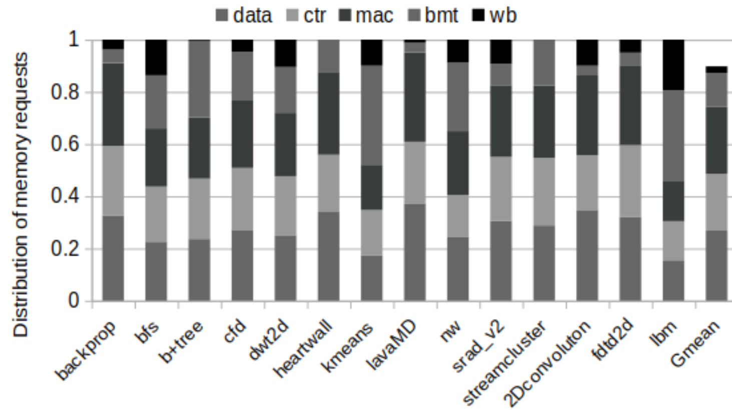

Figure 2.3: Distribution of different types of memory requests.

2.3.2 MSHRs for Metadata Caches

As shown in Fig. 2.3, metadata accesses are the main contributor for additional memory traffic. A deeper analysis reveals that many metadata cache misses are secondary misses, meaning that the missed metadata blocks have already been requested but have not returned from the memory. We report the ratio of secondary misses in different benchmarks in Fig. 2.4. We can see that the secondary misses account for 64.96%, 59.67% and 85.63% for counter/MAC/BMT cache misses on average. And it can be even more than 90% for some memory-intensive benchmarks like streamcluster. The reason is that contemporary GPUs usually employ the sectored cache structures to reduce memory bandwidth requirement. However, the sectored L2 cache combined with streaming data access pattern will lead to multiple sequential access to the same cache line in metadata cache. Suppose we have a streaming memory access pattern with 4 memory requests {0x0, 0x20, 0x40, x60}. In a non-sectored L2 cache with 128B cache line, one cache miss will be generated, which in turn leads to one counter/MAC cache miss. With a sectored L2 cache (4 sectors and each sector

size of 32B), in contrast, four L2 misses are generated, which leads to four counter/MAC cache misses (1 primary and 3 secondary) to the same counter/MAC cache line.

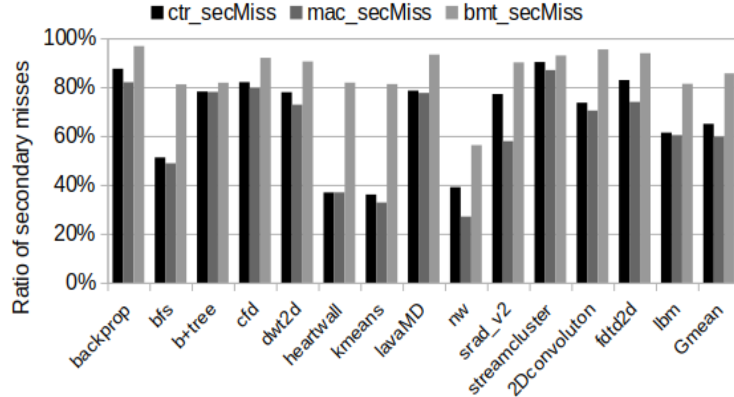

Figure 2.4: Amount of secondary misses in metadata caches.

The solution to avoiding the memory traffic generated by the secondary metadata cache misses is to add MSHRs to metadata caches. Here, we show the performance impact with different MSHR sizes for metadata caches and the results are shown in Fig. 2.5. Due to the metadata cache organization (i.e., one metadata cache line may cover multiple data cache lines/sectors), we assume each MSHR entry can merge at most 512/64/64 requests in the counter/MAC/BMT cache. As we can see from Fig. 2.5, 64 MSHRs in a metadata cache can be a good choice considering the performance and hardware cost. Hence, we assume 64 MSHRs as the default size in our experiments.

2.3.3 Metadata Cache Size

In the next experiment, we vary the metadata cache size from 2KB to 64KB. The performance results are shown in Fig. 2.6. Since our baseline GPU has 32 memory partitions, the overall metadata cache capacity would vary from 192KB (=32x3x2KB) to 6MB (=32x3x64KB).

As expected, enlarging the metadata cache reduces the memory traffic and improves the performance. However, there is still a performance degradation of 46.17% on average even when the overall metadata cache capacity is enlarged to 6MB in total, which is equal to the capacity of our baseline GPU. For memory-intensive benchmarks, the performance overhead can be even more significant. For example, with the 6MB metadata cache, secure

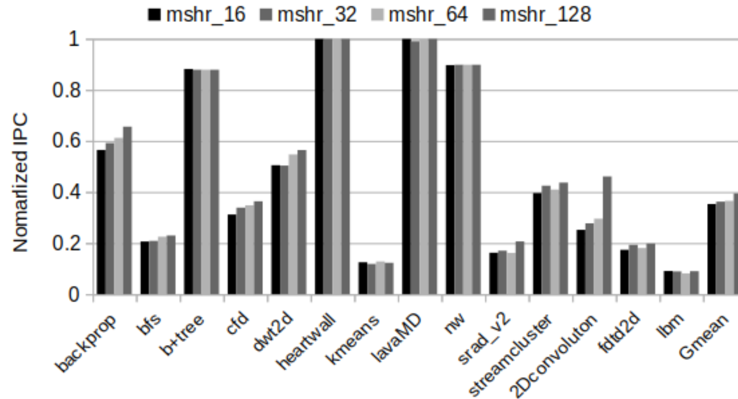

Figure 2.5: Normalized IPC of secure memory with different numbers of MSHRs in metadata caches.

memory support still slows down the GPUs by 78.87% for kmeans, 67.82% for srad_v2 and 72.64% for lbm. On one hand, these benchmarks have very high memory bandwidth utilization, therefore any additional memory traffic will incur more contention to the memory system. On the other hand, these benchmarks have many cold misses even in the presence of large metadata caches.

In the baseline GPU, each memory partition has 192KB (=92KB*2) L2 cache capacity, and each counter cache line has 128 minor counters. To maintain the counters for all the L2 cache blocks in this memory partition, the capacity of counter cache should be at least 192KB/128 = 1.5KB. Therefore we use 2KB as the default size for metadata caches in each memory partition.

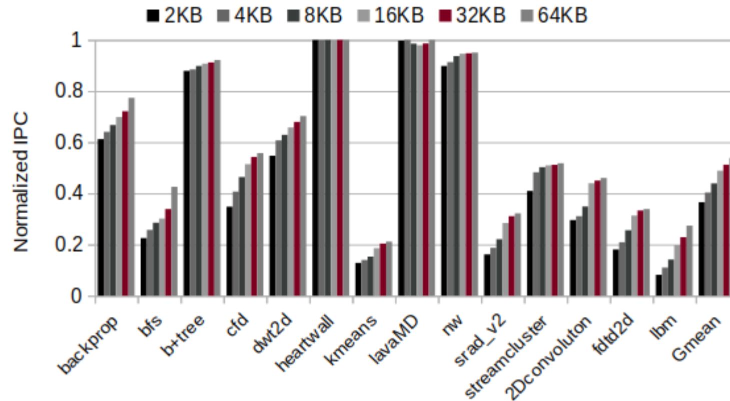

Figure 2.6: Normalized IPC for different metadata cache sizes.

2.3.4 Unified vs. Separate Metadata Caches

On CPU secure memory, Lehman et al.[Leh18] have studied the metadata cache access patterns and concluded that caching all metadata together in an unified metadata cache is better than caching them in separate metadata caches. In our experiment, we model an unified metadata cache in each memory partition that has the same capacity as the combined capacity of the three separate metadata caches. The performance comparison between the unified and separate metadata caches is shown in Fig. 2.7.

From Fig. 2.7 we can see that in contrast to CPUs, separate metadata caches outperform the unified metadata cache on GPUs. To figure out the reason, we report the miss rates of different types of metadata in Fig. 2.8. From Fig. 2.8, we can see that with unified metadata cache, the miss rate of different metadata all raises. From 22.77% to 24.03% for encryption counters, from 31.75% to 31.82% for MACs and from 4.02% to 5.93% for BMT. The reason is, the newly fetched metadata blocks keep on evicting the other metadata blocks from the unified metadata cache due to the streaming access pattern of the GPU workloads, which results in higher metadata misses. Moreover, every evicted counter and BMT node may need to update their parents, which potentially leads to more write updates to the BMT cache and more dirty evictions from BMT nodes. Our evaluation results show that the memory traffic due to metadata writebacks in unified metadata cache can be 1.47X higher than those with separate metadata caches on average.

We also study the reuse distance of metadata in GPU secure memory. We choose the benchmark fdtd2d as our case study and present the reuse distance distribution of its counter and MAC accesses in Fig. 2.9 and Fig. 2.10, respectively. With the unified cache, we first collect the metadata access trace from partition 0 and extract sub-traces for each type of metadata. Then we compute the reuse distance based on the sub-traces. The references are grouped into different reuse distance buckets, where [x,y] means the reuse distance between x and y. From the figure, we can see that since GPUs feature streaming data access pattern, the metadata also shows a streaming pattern as most metadata accesses have a reuse distance of zero (i.e., accessing the same metadata cache line). In addition, with unified metadata caches, the counter and MAC accesses show higher numbers of accesses with reuse distances in the range between 65 and 512 while having smaller numbers of accesses with reuse distances between 1 and 8. This indicates that higher capacities are required to capture these reuses using unified caches compared to separate caches. Note that accesses with small reuse distances (e.g., 0) do not always result in cache hits. The reason is that if the first access is a miss and the data has not been fetched, subsequent

accesses to the same cache line become secondary misses.

Overall, our conclusion is that for streaming data access patterns, we may either use separate metadata caches or adopt smart replacement policies to avoid the thrashing behavior. Note that the thrashing-avoiding replacement policies proposed for CPU last-level caches may not be readily adopted. The reason is that each newly fetched metadata block may be accessed multiple times as each metadata block protects multiple data blocks. For example, one MAC block contains MACs for 16 data blocks, with perfectly streaming accesses, the MAC block will be reused for 16 times.

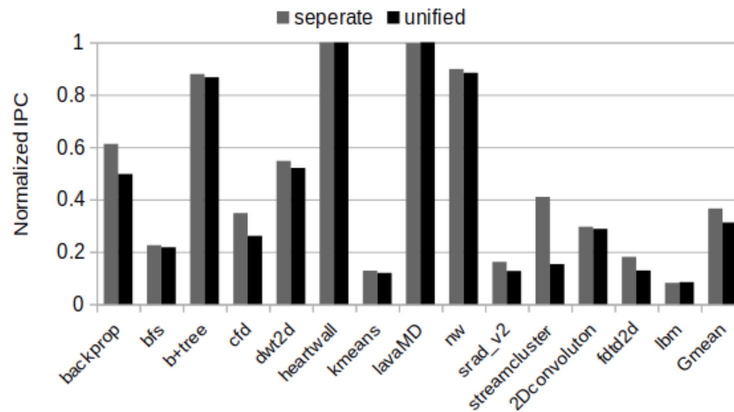

Figure 2.7: Normalized IPC of unified metadata caches vs separate metadata caches.

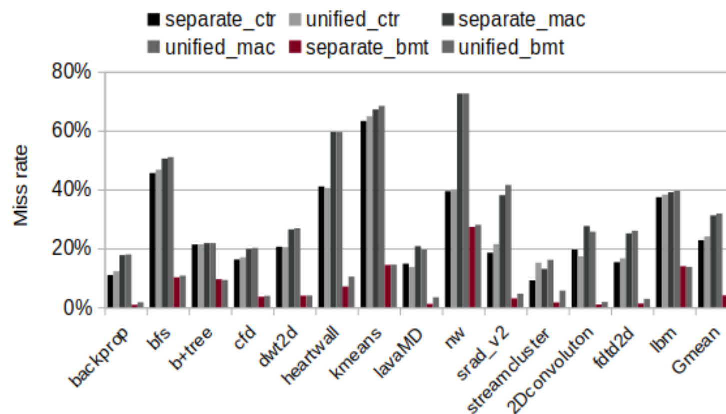

Figure 2.8: Miss rates for different types of metadata in unified vs. separate metadata caches.

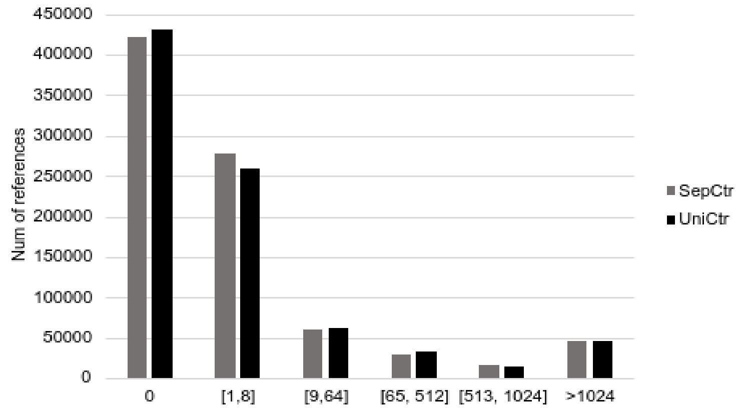

Figure 2.9: Reuse distance of counters of the benchmark fdtd2d

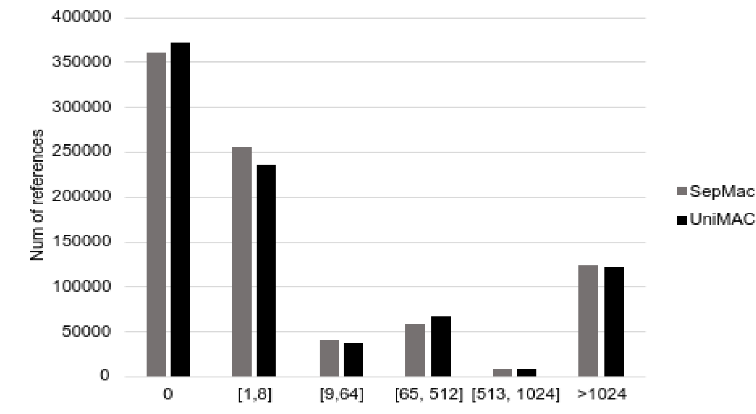

Figure 2.10: Reuse distance of MACs of the benchmark fdtd2d

2.3.5 AES Engine Throughput

To achieve high-throughput computation, GPUs usually have high requirements for the memory bandwidth. For example, our baseline GPU has a peak memory bandwidth of 868 GB/s. So far, we have assumed that each memory partition can be equipped with 2 pipelined AES engines such that the encryption throughput can catch up with the memory bandwidth. In this case, there would be a total of 64 (=2*32) AES engines residing on the GPU chip.

To analyze the performance impact with different numbers of AES engines for each memory partition/unit, we reduce the number of AES engines in each memory partition/unit from 2 to 1 and the performance results are shown in Fig. 2.11. From the figure, we can see that although some benchmarks such as b+tree and kmeans observe a little performance degradation, most of the benchmarks are not affected by having just one pipeline AES engine in each memory partition. The reason is that for benchmarks with relatively low memory utilization, neither the memory bandwidth nor the AES throughput is the performance bottleneck. For the workloads with high memory bandwidth utilization, the performance bottleneck is the extra memory traffic generated by metadata accesses. Only after the memory bottleneck is addressed, the limitation due to AES throughput may be exposed. Some memory-intensive benchmarks, such as srad_v2, streamcluster, 2Dconvolution, fdtd2d and lbm also exhibit slight performance improvement when the number of AES engines is reduced from 2 to 1. The reason is that these benchmarks have relatively high memory bandwidth utilization and large numbers of metadata write backs. Delaying some memory accesses due to limited AES throughput leads to a change in warp scheduling decisions as well as the cache access patterns, which results in small performance increase.

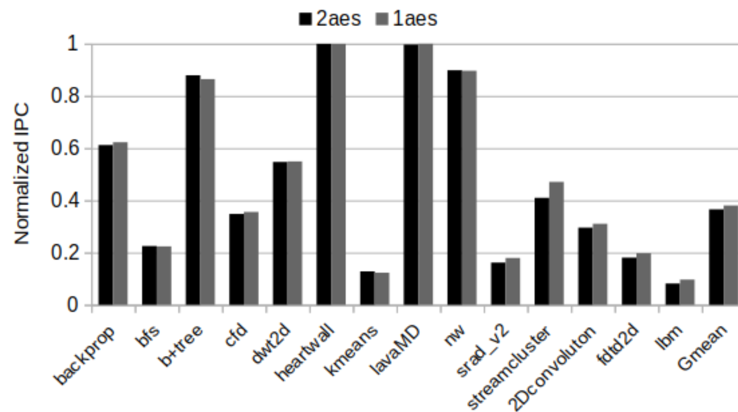

Figure 2.11: Normalized IPC with different numbers of AES engines in each memory partition

2.3.6 Die Area

GPUs are highly-parallel processors, and most of their die area is dedicated for computational resources such as SIMT cores. For example, our baseline GPU models Nvidia Quadro

Table 2.6: Die area of the AES engine

-	Tech	Die Area
JSSC'11[Mat11]	45nm	0.15mm^2
JSSC'19[Sin19]	130nm	13241μm^2
JSSC'20[Kum20]	14nm	4900 μm^2

Table 2.7: Scaled down die area of the AES engine and caches

-	Area(mm^2)/tech	Area(mm^2)/12nm
AES engine	0.0049/14nm	0.0036
64KB cache	0.125821/32nm	0.01769
96KB cache	0.128101/32nm	0.01801

GV100, which is fabricated using 12nm FinFET Nvidia (FNN) technology with 21.1 billion transistors integrated on a die with the size of 815mm^2 [Jia18]. It integrates 80 SMs and each SM has 64 FP32 cores, 32 FP64 and 8 Tensor cores, which means a total of 5120 FP32 cores, 2560 FP64 cores and 640 Tensor cores.

To evaluate the die area required for counter-mode encryption, we estimate the area of both AES engine and the metadata caches. We list the results from prior works on AES design in Table 2.6 and scale the most recent design [Kum20] to the 12nm technology as shown in Table 2.7. From the table, we can see that the area of one AES engine is estimated as 0.0036 mm^2. As there are 32 memory partitions, the total area for 32/64 AES engines is 0.1152/0.2304 mm^2.

We use CACTI v6.5 to estimate the die area for the caches. As CACTI reports the area estimation using the 32nm technology, we also scale the results down to the 12nm technology, as shown in Table 2.7. As CACTI does not support modeling of small caches like 2KB, we report the area estimate of 64KB as 64KB is the aggregated capacity of one type of metadata caches in 32 partitions.

To make room for the die area required by the AES engines and metadata caches, we choose to reduce the L2 cache size. Since each L2 bank is 96KB, we also use CACTI to estimate the area. To accommodate 32 AES engines, the L2 cache capacity would need to be reduced by 0.1152 mm^2/0.01801mm^2*96KB = 614KB. Similarly, the metadata caches will occupy 0.01769*3 = 0.05307 mm^2 on chip area, and hence reducing the L2 capacity by 0.05307/0.01801*96KB = 283KB. If we assume MAC units have similar die area compared to AES engines, the security related hardware resources will reduce the L2 capacity by 614+614+298 = 1526KB in total (24.84% L2 cache capacity).

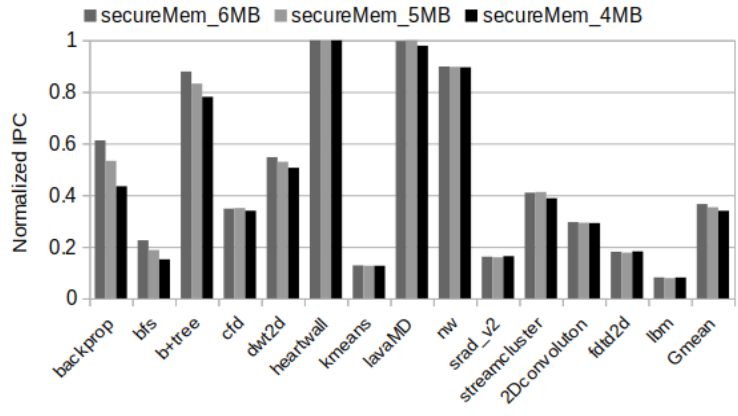

Figure 2.12: Normalized IPC with different L2 cache capacities

To evaluate the performance impact of the reduced L2 capacity, Fig. 2.12 shows the normalized performance for different L2 cache sizes, ranging from 4MB to 6MB. From the figure, we can see that although many benchmarks are not sensitive to the L2 cache capacity, a few of them show relatively high performance degradation. To better understand the reasons, we also report the L2 cache miss rate in Fig. 2.13. From the figure, we can see that some medium-memory-intensive benchmarks show high sensitivity to L2 capacity. It is expected since computation-intensive benchmarks (e.g., heartwall and lavaMD) have small numbers of L2 accesses whereas highly memory intensive benchmarks such as lbm or fdtd2d have very high L2 miss rates. Changing L2 capacity has little impact on these workloads.

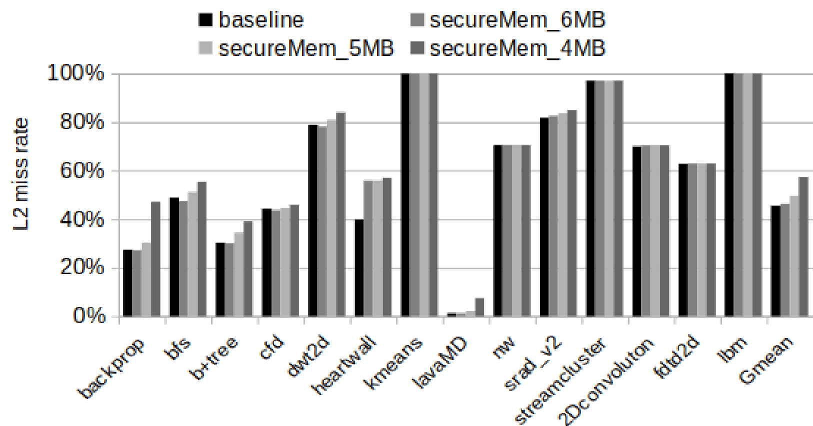

Figure 2.13: L2 cache miss rate

Table 2.8: Evaluated designs for direct encryption

Scheme	What It Represents
direct_x	direct encryption with different encryption latency of x cycles
ctr	counter-mode memory encryption without any integrity protection
ctr_bmt	counter-mode encryption with BMT to protect counter integrity
ctr_mac_bmt	counter-mode encryption with BMT and MACs
direct_mac	direct encryption with MACs.
direct_mac_mt	direct encryption with MACs and MT.

2.4 Direct Encryption

In this section, we evaluate direct encryption and explore the option of different levels of integrity protection. The designs that we evaluate in this section are listed in Table 2.8.

2.4.1 Performance Overheads of Direction Encryption

As direct encryption exposes the decryption latency to the critical path of memory read accesses, it can slowdown the performance heavily on CPUs [Yan03]. However, GPUs are designed for high-throughput computation by exploiting the high degrees of TLP. Hence, GPUs are able to tolerate long operation latency. It is expected that adding some encryption latency to the memory access would not hurt the GPU performance significantly.

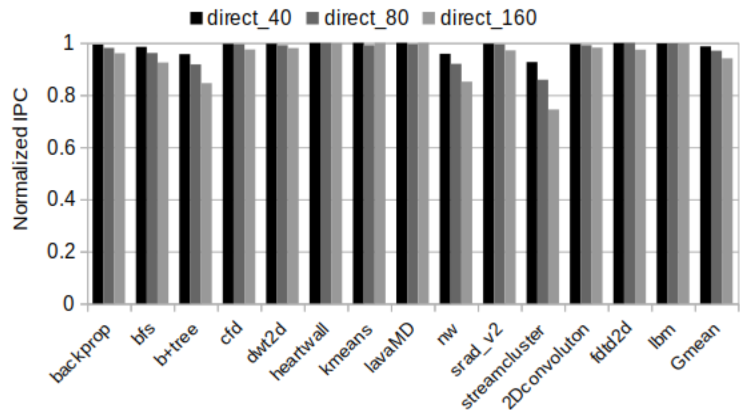

Figure 2.14: Normalized IPC of direct encryption with different encryption latencies.

We model the direct encryption with different encryption/decryption latencies upon our baseline GPU and the performance results are shown in Fig. 2.14. As expected, direct encryption does not affect the GPU performance much. When the encryption latency varies from 40 cycles to 160 cycles, the IPC slowdown is 1.33%, 3.02%, 5.93%, respectively on average. Some benchmark, e.g., b+tree, nw and streamcluster, show more than 10% slowdowns at the high encryption latency of 160 cycles. The reasons are different among these three benchmarks: (1) benchmark streamcluster suffers from high L2 miss rate (97.02%) as shown in Fig. 2.13, which leads to many memory read accesses, (2) benchmark nw is limited by the small kernel (shown in Table 2.4) such that they do not have enough threads to hide the encryption latency. (3) benchmark b+tree shows interesting results because it's neither bounded by kernel size nor L2 miss rate. Digging deeper, we find that with a high encryption latency of 160 cycles, b+tree suffers from 2.42X dram stall time compared the baseline GPU. Our results show that given longer dram stall time, all the available warps in b+tree suffer 12.2% more pipeline cycles to wait data from memory on average, and thus slowdown the performance. Although high encryption/decryption latency does not impact the GPU performance significantly, we choose 40 cycles as the default encryption/decryption latency, which is consistent with prior work on AES encryption for CPUs [Leh18].

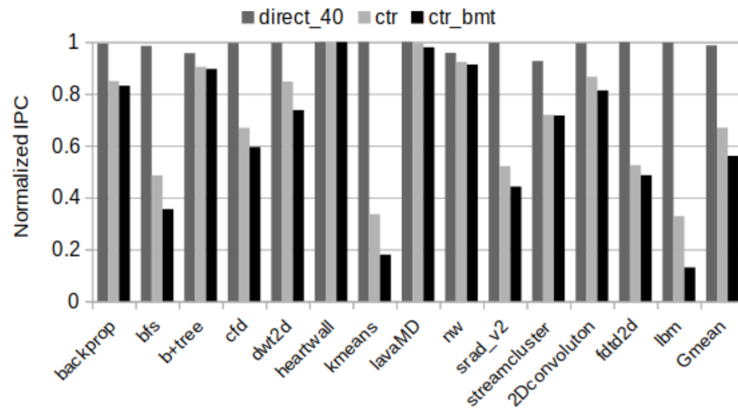

Figure 2.15: Normalized IPC of direct encryption and counter-mode encryption

2.4.2 Direct Encryption vs. Counter-mode Encryption

To better understand the trade-offs between direct encryption and counter-mode encryption, we examine their performance in Fig. 2.15. From Fig. 2.15, we can make two observations. First, as discussed above, the performance impact of direct encryption is almost negligible. Second, compared with direct encryption, counter-mode encryption without integrity checks can lead to a relatively high performance overhead (33.06% on average, and up to 66.44% for the memory-intensive workload lbm). As we studied in section 2.3, the main reason is that counter-mode encryption will generate additional memory traffic to fetch/store counters from/to the off-chip main memory.

Moreover, as we discussed in Chapter 1 section 1.2.2, counter-mode encryption *fundamentally relies* on counter integrity protection to provide data confidentiality. The reason is that the counters are used for encryption and decryption. Without integrity checks of the counters, the GPU cannot tell whether a counter has been altered by the attacker or not. In this case, an attacker may be able to manipulate the counters to recover the plaintext. It can be illustrated as follows. Let us use P as the plaintext, C as the ciphertext, and K as the secret key of AES. Then the ciphertext is generated by $C = E_K(A||Ctr) \oplus P$, where A is the address of the memory block and Ctr is the counter. P can be recovered with $C \oplus E_K(A||Ctr)$ if Ctr can be controlled by the attacker. Hence, with counter-mode encryption, BMT is needed anyway to provide integrity protection to the counters stored in the off-chip memory.

As we can see from Fig. 2.15, adding the integrity protection to counters further increases the performance overhead, 43.94% on average for all the benchmarks in our study.

In summary, our evaluation suggests that if the memory only needs to be encrypted, direct encryption would be a better choice for GPUs.

2.4.3 Integrity Protection

Memory encryption can provide data confidentiality. However, it cannot provide memory integrity protection when the attackers have the ability to tamper the memory contents. With counter-mode encryption, data integrity protection is provided by stateful MACs, and counter integrity protection is provided by BMT. With direct encryption, one can choose to build MAC upon cyphertext with/without a Merkle Tree (MT). The MT is needed to prevent replay attacks and can be built with the MACs as its leaf nodes.

We model these schemes and compare the performance results in Fig. 2.16. For fairness, we assume the same on chip resource used by the metadata caches. In counter-mode

encryption, as mentioned in Section 2.3, a 2KB on-chip metadata cache is used for each type of metadata and a total of 6KB on-chip resource is used in each memory partition. In direct encryption with MAC, we model a MAC cache with the size of 6KB. In direct encryption with both MAC and MT, we model a 3KB MAC cache and 3KB MT cache (the MACs will not access the MT cache as they are cached in the MAC cache).

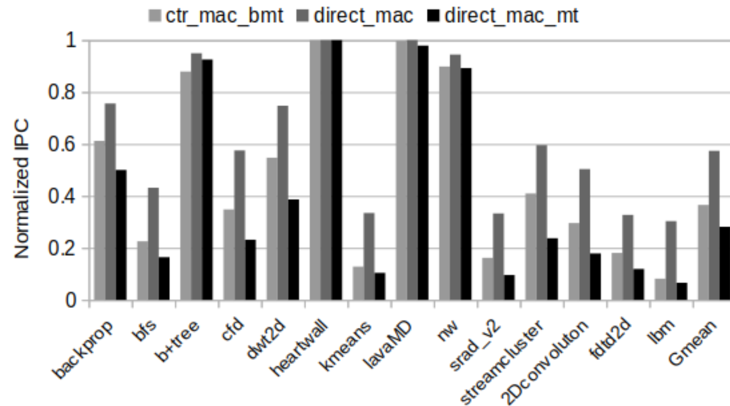

Figure 2.16: Normalized IPC of direct encryption and counter-mode encryption with integrity protection

From Fig. 2.16, we can make two observations. First, given the same on-chip resource for metadata caches, direct encryption with MAC can still perform better than counter-mode encryption scheme with BMT (42.65% **vs.** 63.45% IPC slowdown on average). Second, MT can have high performance impact upon direct encryption. Combined with MACs, it can slowdown the GPUs IPC by 71.87% on average. The reason is that with the same memory protection range, the height of MT (7-level 16-ary tree) is higher than the BMT (6-level 16-ary tree). On one hand, a higher hash tree means that the integrity verification process (every newly fetched MAC block must be authenticated) needs to traverse a longer path and potentially incurs more memory traffic to fetch/store the tree nodes from/to off-chip memory. On the other hand, a higher hash tree, although just 1 level higher, has much larger capacity (due to the arity), which leads to high contention for the MT cache.

2.5 Conclusions

In this paper, we perform detailed performance analysis of different secure memory architecture designs for accelerators like GPUs. From our study, we conclude that we need to architect GPU secure memory differently from it in CPUs. Our key observations include: (1) Due to the latency-hiding capability, direct encryption can be a better alternative to counter-mode encryption for GPUs if only data confidentiality is needed. (2) The AES throughput limitation can be reasonably addressed with one pipelined AES engine in each memory partition. (3) To support integrity verification, both counter-mode encryption and direct encryption need to be further optimized to reduce the performance overhead. In either design, the key bottleneck is the memory traffic generated by security metadata accesses. (4) Metadata caches can reduce the memory traffic and separate metadata caches can be a better choice than unified metadata caches for GPUs. (5) The use of sectored L2 cache on GPUs necessities MSHRs for the metadata caches to reduce the memory traffic.

CHAPTER

3

PSSM: ACHIEVING SECURE MEMORY FOR GPUS WITH PARTITIONED AND SECTORED SECURITY METADATA

3.1 Motivations

3.1.1 Performance Impacts of Naive Design

To pinpoint the performance bottlenecks of adopting CPU secure memory to GPUs directly, we first perform a detailed performance analysis. The simulation methodology is presented in Section 3.3.1. We model the secure memory architecture with split-counter mode encryption, combined with MAC and BMT for integrity protection. A 64KB cache is added for each type of security metadata. There are 32 memory partitions in our baseline GPU model, thus each memory partition is equipped with a 2KB counter cache, a 2KB BMT cache, and a 2KB MAC cache. We assume the metadata cache (MDC) line size of 128B, the same as the data cache line size. A split-counter block with 128B consists of 1×128-bit major counter and 128×7-bit minor counters (SC_128) is adopted in this naive design, as shown in Fig.

3.5. This way, one counter block is used for 128 data blocks or a major counter is shared among 128 data blocks.

Fig. 3.1 reports the instruction per cycle or IPC (higher is better) of secure memory designs normalized to the baseline GPU without secure memory support. In the figure, the design directly adopted from CPU secure memory is labeled **secureMem**. Two idealistic (but infeasible) designs are also included to determine performance bottlenecks. **large_mdc** represents an ideal design with unlimited MDC capacity, while **0_crypto** represents an ideal design with zero-latency cryptographic operations (i.e., zero encryption and MAC computation latencies). In all these designs, the metadata are generated using the physical addresses. In other words, the addresses used in counter mode encryption and MACs are physical addresses of the data blocks.

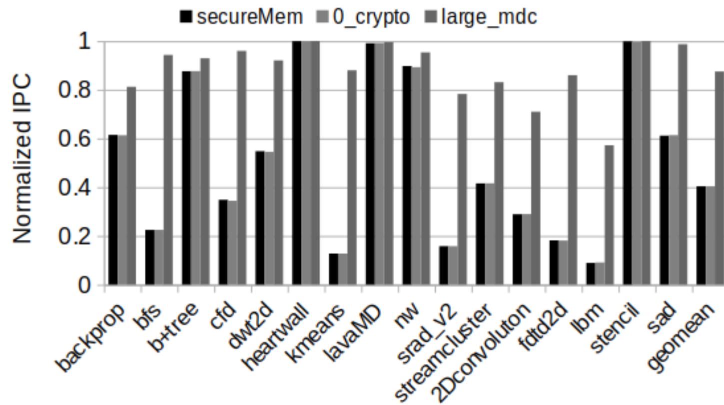

Figure 3.1: Normalized performance of secure memory designs to the baseline GPU without secure memory support.

From Fig. 3.1, we can see that directly adopting the CPU secure memory design results in a performance slowdown for GPU of 59.66% on average (geometric mean). The performance degradation is even higher for memory-intensive benchmarks like lbm (91%) and srad_v2 (84%). Zero cryptographic latency does not improve the performance of secure memory, primarily because GPUs being designed to be latency tolerant. On the other hand, with unlimited MDC sizes, the average performance is more than doubled and approaches the baseline GPU without secure memory, indicating that the memory accesses to fetch/store security metadata from/to off-chip memory is the main performance bottleneck. Moreover, we can observe that even with unlimited MDC capacity, where only cold misses on metadata remain, the performance slowdown can still be significant, 13% on average. The reason

is that the performance of GPUs is highly bounded by memory bandwidth and we would better avoid any additional memory bandwidth contention.

3.1.2 Problem Diagnosis

As presented in Section 3.1.1, even with unlimited MDCs, there is a performance gap between the baseline GPU and the one with secure memory support. By analyzing the MDCs in different partitions, we discover that the partitioned memory structure and memory interleaving used in contemporary GPUs lead to redundant metadata being fetched to and stored in the MDCs in different partitions. This implies that some memory bandwidth is wasted due to unnecessary data transfers. The main reason for the redundant metadata among different memory partitions is that the state-of-the-art split-counter mode encryption organizes the counters based on data blocks' physical addresses. However, with partitioned memory structure, memory blocks within the same physical page are mapped to different memory partitions so as to avoid the partition camping problem [Aji11][Yan10]. In our baseline GPU, pseudo random memory interleaving is employed and the interleaving granularity is 256B (2 memory blocks or cache lines with the block/line size of 128B). With split counters, one counter block contains the encryption counters for many data blocks mapped to different partitions. Since each memory partition has its own memory controllers and cannot access other partitions directly, there is a question of how to maintain the split counters.

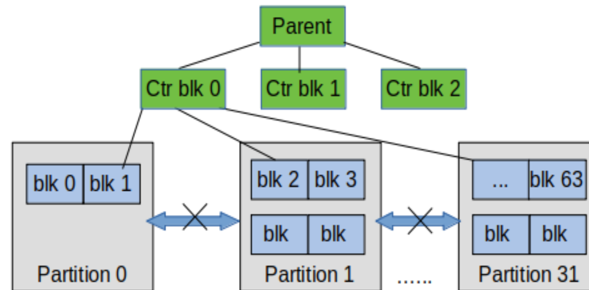

Figure 3.2: A single (128B) counter block corresponds to 128 data blocks in 32 partitions. Similarly, one BMT node requires multiple counter blocks.

The problem is further illustrated with Fig 3.2. Here, we simply assume the sequential interleaving scheme and the interleaving granularity is 2 memory blocks. With split counters,

one 128B counter block protects 128 data blocks and these data blocks are distributed across the 32 memory partitions. As a result, the same counter block needs to be accessed by all the 32 memory partitions and it may be stored in the counter cache in each partition, leading to significant redundancy. Besides the redundancy in counter caches, another key problem is, which memory partition should be used to accommodate this counter block in off-chip memory? There are two options.

Option 1: Redundant counters We can store several copies of the counter block in different partitions such that each partition can access the its own copy. For example, a copy of counter block 0 can be stored in off-chip memory corresponding to partition 0, another stored in partition 1, etc. However, the issue is the coherence among the multiple copies of the same counter block. If there is one minor counter overflow in one partition, the major counter would need to be updated for all the copies of the counter block. But there is no communication channel across different partitions to support such an operation. The remedy would be adding an interconnect network across different partitions, which would incur high hardware cost.In our paper, we adopt this design with redundant counters as our baseline.

Option 2: Single copy of counters We can also store all the metadata in one partition. For example, all counter blocks can be stored in partition 1. When partition 2 received a memory request, which needs to access its counter block, the memory controller in partition 2 would have to access partition 1, thereby requiring the interconnects across different partitions.

Note that the BMT nodes share the same problems as the counters since the BMT is built on top of the counter blocks. In other words, a BMT node needs to be accessed by multiple partitions.

3.1.3 Coarse-Grain Interleaving

To solve the counter redundancy and coherence problem, one possible solution is to enlarge the interleaving granularity to larger memory chunks like page-level memory interleaving [Zha00]. For example, if four consecutive memory pages in the physical memory space (assuming 4KB pages) can be assigned to the same memory partition, all the 128 data blocks (16kB = 128*128B) corresponding to a 128B counter block would reside in the same memory partition. However, the problem of such coarse interleaving granularity is *partition camping* [Aji11], which means that multiple streaming multiprocessors (SMs) may try to access the data from the same memory partition, resulting in contention at the partition

and low memory bandwidth utilization overall.

To evaluate the performance impact of coarse interleaving granularity, we model the memory interleaving at 1-page and 4-page granularity without secure memory and normalize the performance to the IPC of baseline GPU without secure memory. The results are shown in Fig. 3.3. As we can see from Fig. 3.3, 1-page interleaving slows down GPU performance by 13% on average compared to the baseline GPU, which uses 2-block/256B interleaving. Moreover, 1-page interleaving cannot solve the counter storage problem as one counter block still contains the counters from more than one partition. In comparison, 4-page interleaving can eliminate the counter storage problem entirely, as all the data blocks, whose counters are in the same counter block, reside in the same partition. The overheads, however, are increased: 4-page interleaving reduces GPU performance by almost 29% on average due to more severe partition camping.

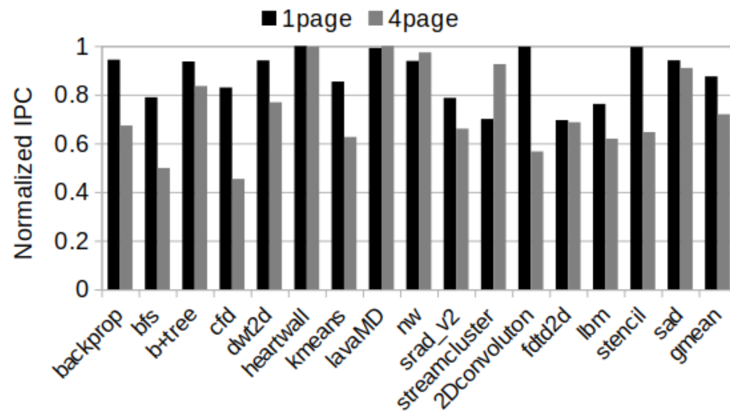

Figure 3.3: The IPC of different page interleaving granularities, normalized to the baseline GPU with 256B interleaving and without secure memory support.

Even with coarse-grain interleaving, the problem with the BMT remains. Some BMT nodes, especially those in high levels of the tree, are still needed by different partitions, as one BMT node may have several children nodes that span over multiple partitions.

3.1.4 Sectored MDC

Sectored caches are commonly used for commercial GPUs to reduce memory bandwidth consumption. As pointed out in Section 3.1, the key performance bottleneck of GPUs with secure memory support is the bandwidth contention due to metadata accesses. Therefore,

we expect that the GPU performance can benefit from sectored MDCs. In our baseline GPU, the data caches (both L1 and L2) use 128B cache lines and each cache line has 4 sectors, i.e., each sector has a size of 32B. Similarly, we model the MDC with 4 sectors in each cache line. Fig. 3.4 shows the performance comparison of secure GPU with sectored and regular (i.e., non-sectored) MDCs of the same capacity and set associativity. As expected, we can clearly see that the performance of sectored MDC is significantly better than the non-sectored MDC. As such, we conclude that sectored MDCs are preferred for GPU secure memory design.

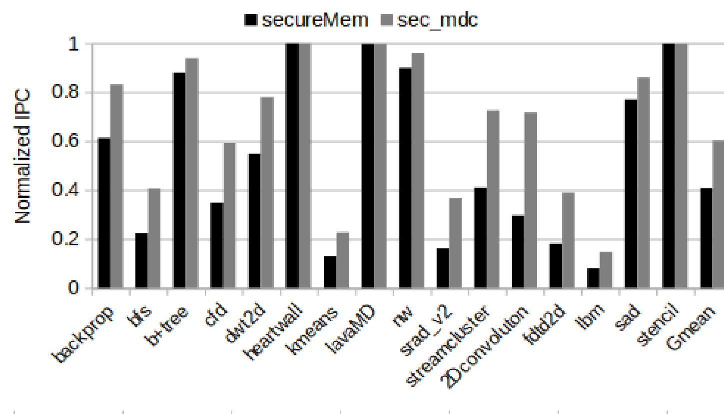

Figure 3.4: Performance comparison of secure GPU with non-sectored MDCs (labeled 'secureMem') and sectored MDCs (labeled 'sec_mdc').

However, there is a problem with separating split-counters into sectors as shown in Fig. 3.5. With a 128B counter block being divided into 4 sectors, the major counter (128b) along with 18 minor counters (7b each) is usually stored in the first sector (32B) while the remaining minor counters are stored in other sectors. If an L2 cache miss leads to a counter access to the sector other than the first, the memory controller still needs to issue 2 memory requests to fetch 2 sectors containing the major counter and the corresponding minor counter to recover the counter value needed for memory encryption/decryption. To avoid such additional memory transactions, the counter block organization needs to be redesigned such that it can be friendly to the sectored cache structure.

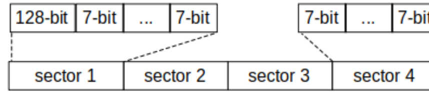

Figure 3.5: A Split-counter block of 128B, containing 1×128-bit major counter and 128×7-bit minor counters. When split into 4 sectors, the first sector contains the major counter and some minor counters.

3.1.5 Sectored Data Cache and MAC Verification

In CPU secure memory, a MAC is calculated for each cache line. On GPUs, sectored data caches are commonly used to save bandwidth consumption. However, secure memory presents a new trade off between regular data bandwidth and metadata bandwidth consumption. On one hand, if a MAC is generated for each line, MAC computation and verification would require all the sectors in the cache line, which would force a sectored cache to operate like a non-sectored one. On the other hand, if one MAC is used for each sector, the amount of MAC data will be highly increased, which would lead to high MAC data storage and bandwidth overheads. One partial solution is to truncate the MAC size for a sector. Although this might compromise the strength of MAC verification, we consider the performance impact of such MAC truncation in our evaluation.

As the design of one MAC per cache line would essentially make a sectored cache to behave like a non-sectored one, we examine the performance difference between the sectored and non-sectored caches for GPUs without secure memory. In our experiment, we compare a non-sectored L2 cache with a sectored L2 while keeping L1 caches as sectored. The reason is that sectored L1 caches help reduce bandwidth pressure on the L1-L2 interconnect network. We report the IPCs of non-sectored L2 cache designs normalized to the sectored L2 cache design in Fig. 3.6. The label 'nL2_N' denotes the model of non-sectored L2 cache where each L2 MSHR (miss status handling register) can merge up to N requests that missed in the L2 cache and each request is a cache line. In our baseline GPU with sectored L2 cache, each L2 MSHR can merge 4 L2 miss requests and each request is one sector (i.e., 32B) rather than a cache line. The structure of such a MSHR is illustrated in Fig. 3.7 (a), where the primary miss (labeled 'priMiss') is the first request to the cache line/sector and the secondary misses (labeled 'secMiss') are the merged requests. If a request finds a matching MSHR but all its entries have been occupied, the request will be stalled and block the subsequent requests.

As we can see from Fig. 3.6, the performance of non-sectored L2 is very close to it of

the sectored L2 for most benchmarks when N is 16 or larger. For N being 4, the sectored L2 has significantly higher performance for several memory intensive benchmark. The reason is that the non-sectored L2 design suffers many more MSHR stalls due to the merging granularity. Considering a streaming-like access sequence, A1, A2, A3, A4, A5, all of which are misses and map to different sectors in the same cache line: A1 and A5 to sector 1, A2 to sector 2, A3 to sector 3, and A4 to sector 4. With a sectored L2, A1, A2, A3, and A4 reside in different MSHRs and A5 merges with A1, as illustrated in Fig. 3.7 (b). With a non-sectored L2, A1, A2, A3, and A4 reside in the same MSHR and A5 is blocked since the matching MSHR is fully occupied, as illustrated in Fig. 3.7 (c). After fixing this MSHR stall with higher N values, the performance of the non-sectored L2 is very close to the sectored one. One exception is benchmark kmeans, for which sectored accesses reduce the overall bandwidth and a sectored L2 has better performance as a result of poor spatial locality of the benchmark. On the contrary, the benchmark lbm exhibits strong spatial locality and the non-sectored L2 designs show higher performance than the sectored one.

In summary, we can see that although a sectored L2 is beneficial for certain workloads, the performance of a non-sectored L2 cache is very close the sectored one on average. Note that for a sectored cache with one MAC per cache line, the MSHRs would not be a bottleneck since the requests are processed at the sector granularity.

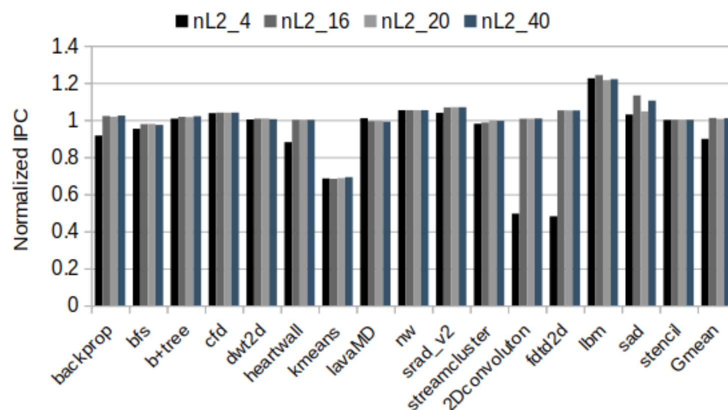

Figure 3.6: The IPC of a non-sectored L2 cache with different numbers of request merges in an L2 MSHR normalized to the baseline sectored L2.

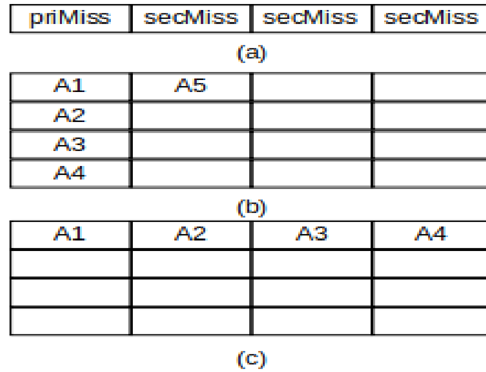

Figure 3.7: The L2 MSHRs. (a) The structure of an L2 MSHR. (b) The MSHR state of a sectored L2 after the access sequence A1-A5. (c) The MSHR state of a non-sectored L2 cache after the same access sequence A1-A5.

3.2 Architecture Design

3.2.1 Overall Architecture

From our performance study in Section 3.1, we observed that CPU secure memory scheme cannot be directly adopted to GPU without losing much performance. To adapt the secure memory architecture design for GPUs, we propose partitioned and sectored security metadata (PSSM). PSSM has two simple yet effective components. First, it uses the partition-local addresses, which are the offsets within a partition, instead of physical addresses to construct the security metadata. Second, it reorganizes the split counter blocks to make them friendly to sectored caches. Our overall architecture design is shown in Fig. 3.8. With partitioned memory, each memory partition has its own memory controller. The memory encryption engine (MEE) [Gue16b] and MDCs are integrated into the on-chip memory controller. PSSM eliminates the metadata redundancy and coherence problem, and thus we can store the security metadata locally in each memory partition. There is also no need for cross-partition communication.

The MEE operates as an extension to the memory controller. It contains the AES encryption engines and hash/MAC engines. All the L2-to-DRAM requests are forwarded to the MEE, and the MEE will encrypt/decrypt data before sending/fetching it to/from the off-chip memory. To verify data integrity, the MEE will also generate additional memory transactions to validate the MAC for each data block, and traverse/update the integrity tree. A special register (labeled as **Root** in Fig. 3.8) is used for the root of the integrity tree. Also, a special off-chip memory region is reserved in each memory partition to store the security

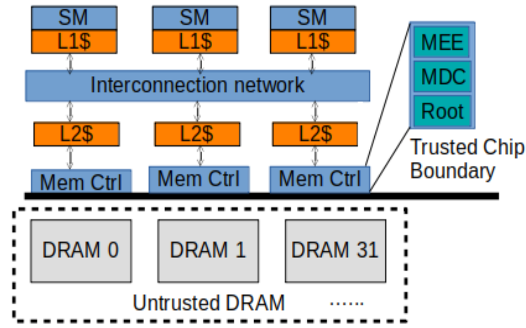

Figure 3.8: Secure GPU architecture with the trust boundary as the GPU chip

metadata including counters, MACs and intermediate integrity tree nodes.

3.2.2 Using Local Addresses for Security Metadata

With a partitioned memory structure, each memory access will be mapped to a partition. If this access misses in the L2 cache banks of the partition, the access goes to the off-chip memory through the memory controller of the partition. There is a mapping function, which converts the physical address into a partition id and a partition offset. In our design, we propose to use the partition offset, which we refer to as local address, to generate/organize the security metadata.

In Fig. 3.9, we illustrate the difference between physical and local addresses with an example of sequential interleaving across 32 memory partitions and the interleaving granularity is 2 memory blocks. With a 4KB page size, 32 memory blocks within the same physical page are mapped to 16 partitions, i.e., blk0, blk1, ..., blk15, where the blkids are physical addresses. With sequential interleaving, the partition id can be computed as (physical address / partition granularity) % number of partitions or blkid/2%32 whereas the partition offset is (physical address / number of partitions / partition granularity) * partition granularity + physical address % partition granularity), or (blkid/64)*2 + blkid % 2, where '/' is integer division and '%' is remainder. PSSM uses local addresses for security metadata. As shown in Fig. 3.9, a local page contains 32 blocks with consecutive local addresses while their physical addresses are not consecutive.

Using local addresses to organize the counter blocks, memory blocks within the same local page will share one counter block. PSSM stores the counter blocks locally within each partition memory. Therefore, the BMT is also constructed solely based on the counter blocks within the same partition. In other words, each partition has its own BMT with the

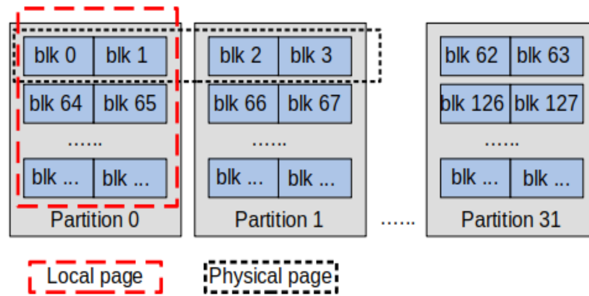

Figure 3.9: Physical and local addresses in partitioned memory organization.

root stored in the corresponding on-chip memory controller. The benefits of this design are: (1) metadata (counter/bmt node) redundancies are eliminated and there is no coherence issue, (2) smaller and shallower integrity trees compared to a single BMT for all partitions.

Our baseline GPU uses pseudo random interleaving [Rau79] memory. There are also other memory interleaving schemes, e.g., sequential interleaving, prime-module interleaving [Law82], skewed-interleaving [HAR87], etc. A common feature of memory interleaving is the nature of bijection [Rau79], which means that with local addresses and partition ids, the corresponding physical addresses can be computed, and vise versa. Also, the cost of address transformation between a physical address and a local address is usually minor.

3.2.3 Making Metadata Friendly to Sectored Caches

As discussed in Section 3.1, sectored MDCs are preferred than non-sectored ones because their bandwidth-saving effects. A MAC cache line and a BMT cache line can be easily broken into smaller sectors since the MAC size and the hash values are a multiple of bytes (e.g., 8). A counter cache line or a counter block, however, is not friendly to sectored caches as discussed in Section 3.1.4. In PSSM, a single major counter is divided into multiple major counters and a major counter is shared by a smaller number of minor counters. Taking a 4-way sectored block as an example in Fig. 3.10, for each sector of 32B, there is a 32-bit major counter and 32 7-bit minor counters and we refer to this counter block design as SC_32. In other words, one sector in a counter block corresponds to 32 data blocks.

Our sectored counter design is inspired from the memory write characteristics of GPU applications. In Fig. 3.11, we report the numbers of stores per kilo instructions. The figure clearly shows that memory writes only account for a very small portion of overall instructions. The write back caches (i.e., L2) combine multiple stores to the same cache lines, further reducing the numbers of writes to memory.

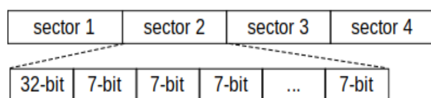

Figure 3.10: Sectored split-counter design: each sector has 1×32-bit major counter and 32×7-bit minor counters.

As a result, although PSSM uses a smaller major counter (32 bits) than the design in Fig. 3.5 (128 bits), there would not be a problem with the potential major counter overflows. Moreover, compared to the design in Fig. 3.5, the PSSM counter block design in Fig. 3.10 has lower overhead of a minor counter overflow: each minor counter overflow leads to 32 blocks to be re-encrypted rather than 128 blocks in Fig. 3.5.

Note that even with our proposed PSSM counter block design, there are subtle issues with sectored MDCs, counter and BMT caches in particular, due to BMT verification. With the secure hash function, a 128B counter block (or a 128B BMT node) is hashed into an 8B value as a part of the counter block's (or the node's) parent. To verify a sector in a counter block (or a BMT node), all the sectors in the same counter block (or the node) are needed to generate the hash. Therefore, the sectored counter cache or the sectored BMT cache needs to operate like a non-sectored one for the verification purpose. The benefit of a sectored counter cache or BMT cache is the reduced write traffic: when a dirty counter block or a BMT node is evicted, not all its sectors are dirty.

In the case where BMT verification is not needed, the sectored counter cache can operate normally for both read misses and dirty evictions, i.e., one sector at a time rather than one line at a time.

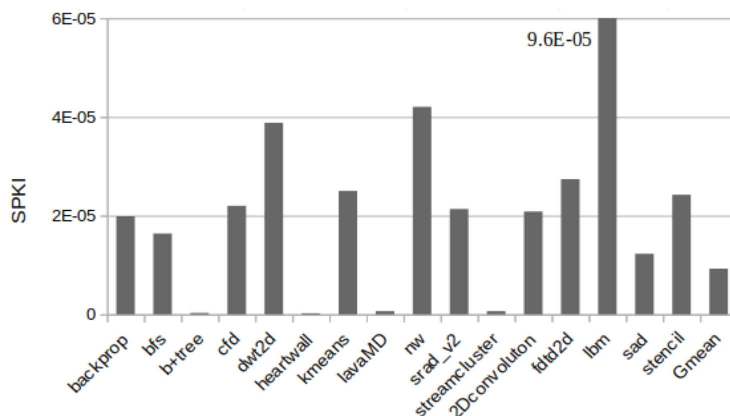

Figure 3.11: Numbers of stores per kilo instructions (SPKI).

3.2.4 Encryption and MAC Engine

In counter-mode encryption, the choice of pad value is critical for security because pad reuse can lead to information leakage. In other words, the pad value must be unique. In conventional CPU secure memory, to ensure the temporal uniqueness, which means that pads are unique over time for each memory block, a counter value is maintained for each memory block as a pad component and is incremented on each write back. To maintain the spatial uniqueness, the physical block address is also included to form the encryption pads.

In PSSM, the encryption/decryption is performed at the granularity of a cache sector. It can be illustrated with Fig. 3.12. However, given that local addresses can be the same across different memory partitions and lead to pad-reuse, PSSM includes the partition id and sector id into the encryption pad. Let us use P as the plaintext, C as the ciphertext, and K as the secret key of AES. The memory encryption can be denoted as

$$C = E_K(local_addr||Ctr||pid||sec_id) \oplus P$$

where $local_addr$ is the local address of the memory block, Ctr is the combination of major counter and minor counter, pid is the memory partition id, sec_id is the sector id within the cache line. The size of each input field is shown in Fig. 3.12. Depending on the number of memory partitions and partition granularity, the sizes of each field for the encryption input are adjusted accordingly.

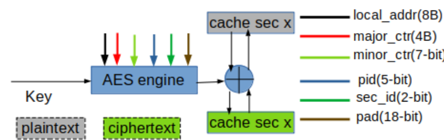

Figure 3.12: The encryption/decryption process in PSSM. The input to the AES encryption engine ensures encryption seed uniqueness, both temporally and spatially.

In PSSM, the MAC may be calculated based on a cache sector or a cache line. The input fields and MAC calculation process are illustrated with Fig. 3.13. PSSM includes the sector id, partition id, and the local address into the MAC calculation to make the MAC location dependent. Note that depending on what MAC algorithm being used, the padding bits of

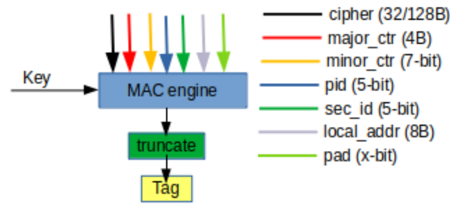

Figure 3.13: The MAC generation process in PSSM. The MAC computation output is truncated to 64/32 bits. Sector id is used when a MAC is generated for each sector.

the MAC engine can be different.

3.2.5 Bandwidth for Accessing MACs

Among different types of metadata, accessing MACs incur high memory bandwidth (See Section 3.3.2). To address this performance bottleneck, PSSM may opt to truncate the MAC value to a smaller size. In our default setup, a MAC of 8B is used for each 32B sector/128B cache line. In our experiments, we evaluate the performance when we truncate the MAC value to 4B. We think that the truncated MAC is sufficient for GPU security for a few reasons. First, any random modification only has a very small chance of producing a hash collision due to the nature of computation resistance of the underlying hash function. With 4B MAC, an attacker only has $\frac{1}{2^{32}}$ or less than 1 in a billion chance to successfully bypass the MAC verification by randomly changing any bits of the data in off-chip memory.

Second, unlike CPUs which may run long-running server applications, most GPU applications execute short-running kernels. Every instance of kernel execution uses a different session key hence the attacker cannot succeed across different kernel executions. Hence, the attacker must succeed in producing a hash collision within a single kernel lifetime. Some attacks, such as rowhammer, takes a while to succeed, e.g., 0.64 second on average to flip a single bit in the RAMbleed attack [GOO19]. Therefore, as long as a single kernel executes for less than 1 hour, the chance of thousands of bit flips producing a hash collision is still much less than one in a million. Furthermore, once MAC mismatch is detected, GPU will be rebooted hence the attacker for practical purposes can only modify memory once before detection.

Table 3.1: Baseline GPU Configuration

SM config	80 SMs, 1132 MHz
Register File	256KB/SM, 20MB in total
L1 D-Cache	32KB/SM
Shared Memory	96KB/SM
L2 cache	2 banks per memory partition, each L2 cache bank is 96KB, 6MB in total
DRAM	850MHz, 32 partitions, 868GB/s, pseudo random memory interleaving.

Table 3.2: MDC and MEE Organization

Counter cache	2KB / memory partition, 128B blk, 4-way sectored, 256 MSHRs, allocate-on-fill policy, sectored as default unless otherwise noted.
Mac cache	2KB / memory partition, 128B blk, 4-way sectored, 256 MSHRs, allocate-on-fill policy, sectored as default unless otherwise noted.
Bonsai Merkle Tree cache	2KB / memory partition, 128B blk, 4-way sectored, 256 MSHRs, allocate-on-fill policy, sectored as default unless otherwise noted.
Hash/Mac latency	40 cycles
AES engines	1 pipelined AES/memory partition

3.3 Evaluation

3.3.1 Methodology

We evaluate our designs with GPGPU-Sim v4.0 [Kha20]. Our baseline GPU configuration is shown in Table 3.1, which is modeled based on the Nvidia Volta architecture [Jia18].

In our experiments, we assume that a range of 4GB device memory is protected. The detailed MDC and MEE organization is listed in Table 3.2. The MDCs are sectored by default unless otherwise specified. The different secure memory designs that we evaluate in our experiments are listed in Table 3.4. In one of our experiments, we also relax our threat model and only focus on GPU memory encryption without integrity protection. Table 3.5 lists the schemes we evaluate for GPU memory encryption, including different designs using split-counters and one using monolithic counters.

We use 16 benchmarks from a wide range of benchmark suites including Rodinia

Table 3.3: Benchmarks

Categorization	Benchmark name	Bandwidth utilization
non memory intensive	heartwall	<1%
	lavaMD	<1%
	stencil	<1%
	sad	5%-7%
	nw	<2%
	b+tree	12%-14%
medium memory intensive	backprop	25%
	cfd	15%-50%
	dwt2d	20%-50%
	kmeans	40%-45%
memory intensive	bfs	5%-60%
	srad_v2	79%- 80%
	streamcluster	78%-80%
	2Dconvolution	53%
	fdtd2d	82%-83%
	lbm	58%

3.1 [Che09], Parboil [Str09] and Polybench [GG09]. Table 3.3 elaborates the details of these benchmarks. For each benchmark, we simulate 4 million cycles. The reason is that previous works [Lin18] have pointed out that the variation of statistic counters becomes very small after the kernel is simulated for 2 million cycles for these benchmarks. Table 3.3 also classify the benchmarks based on their bandwidth utilization when running on the baseline GPU without secure memory support. We report normalized IPCs in our evaluation with the baseline as the GPU with sectored data caches and without secure memory support.

3.3.2 Performance

Overall Performance We evaluate our PSSM designs for both one MAC per sector and one MAC per cache line MAC with different MAC sizes, and report the performance results (normalized to the baseline GPU) in Fig. 3.14. From the figure, we can make the following observations. First, compared with the baseline secure memory design, labeled 'secure-Mem', our PSSM scheme improve the performance significantly. The average performance overhead is reduced from 59.22% to 42.03% for PSSM_sL2_8B_sMdc, 31.09% for PSSM_-sL2_4B_sMdc, 19.06% for PSSM_nL2_8B_sMdc and 16.84% for PSSM_nL2_4B_sMdc. The main reason is that the redundant metadata are eliminated. One exception is benchmark nw, for which our PSSM_sL2_8B design performs worse than the secure memory baseline.

Table 3.4: Evaluated designs for GPU secure memory with both memory encryption and integrity verification.

Scheme	What It Represents
secureMem	Baseline GPU with secure memory, and the security metadata is organized with physical address. Sectored L2 cache and 2B MAC per sector.
PSSM_sL2_xB_sMdc	secure GPU memory with our PSSM design, the L2 cache is sectored, and the MAC is **x** bytes per sector.
PSSM_sL2_8B_nMac	secure GPU memory with our PSSM design, the L2 cache is sectored, and the MAC is 8B bytes per sector. The MAC cache is non-sectored to show the impact of sectored MAC cache.
PSSM_nL2_xB_sMdc	secure GPU memory with our PSSM design, the L2 cache is sectored (but behaving like non-sectored) since the MAC is **x** bytes per cache line.

Table 3.5: Evaluated designs for GPU memory encryption.

Scheme	What It Represents
SC_128_nMdc	Encrypted GPU memory with split counters. The major counter is 128-bit and the encryption counters are organized with physical address and the counter cache is non-sectored.
PSSM_Mono_Ctr_sMdc	Encrypted GPU memory with 32-bit monolithic counters. The counters are organized with local addresses and the counter cache is sectored.
PSM_SC_128_nMdc	Encrypted GPU memory with split counters. The counters are organized with local addresses. The major counter is 128-bit and the counter cache is non-sectored.
PSSM_SC_32_sMdc	Encrypted GPU memory in split counters. The counters are organized with local addresses. The major counter is 32-bit and the counter cache is sectored.

The reason is that this benchmark has an irregular small kernel, whose baseline IPC is only 23.4. With our PSSM and the 8B MAC per L2 sector design, the warp scheduling decision was altered, leading to performance variation. Second, MAC caches benefit from sectored cache designs, as we can see from Fig. 3.14, PSSM_sL2_8B_sMdc performs better than PSSM_sL2_8B_nMac. It is expected because MAC accounts for the most storage overhead for security metadata, and every DRAM access must be authenticated with MAC. Third, a sectored L2 with one MAC per cache line outperforms sectored L2 caches with one MAC per sector MAC. The reason is that the MAC storage overhead of the one-MAC-per-sector design is much higher (N times) that of one-MAC-per-cache-line design, where N is the number of sectors in a cache line. Consequently, the bandwidth requirement of MAC accesses is much higher in one-MAC-per-sector designs, leading to more severe memory bandwidth contention. Fourth, truncating the MAC size from 8B to 4B further reduces the memory bandwidth contention, resulting in a performance overhead reduction for both PSSM_sL2_4B_sMdc and PSSM_nL2_4B_sMdc.

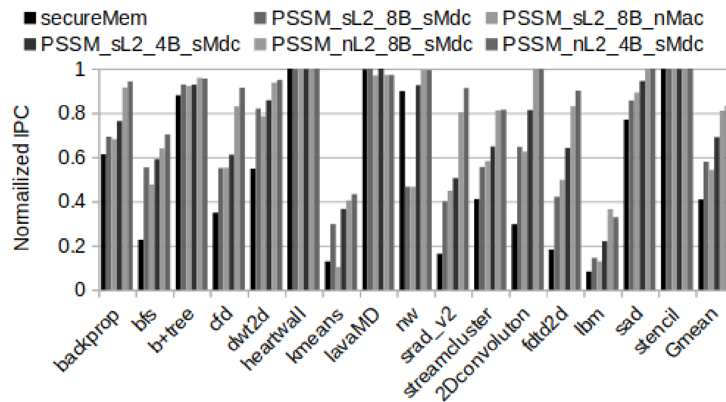

Figure 3.14: Normalized IPC of different secure GPU memory designs.

To better understand the performance impacts, we also present the numbers of metadata cache misses per kilo instructions of our PSSM_nL2_sMdc design in Fig. 3.15. From the figure, we can make two observations: First, the MAC cache has high miss rates for most workloads and potentially contributes to high memory bandwidth consumption, especially for the memory intensive benchmarks. Second, the benchmarks kmeans and lbm have very high cache miss rates for all the three types of metadata. The reason is that these two benchmarks have L2 miss rates higher than 95%, and each DRAM access needs

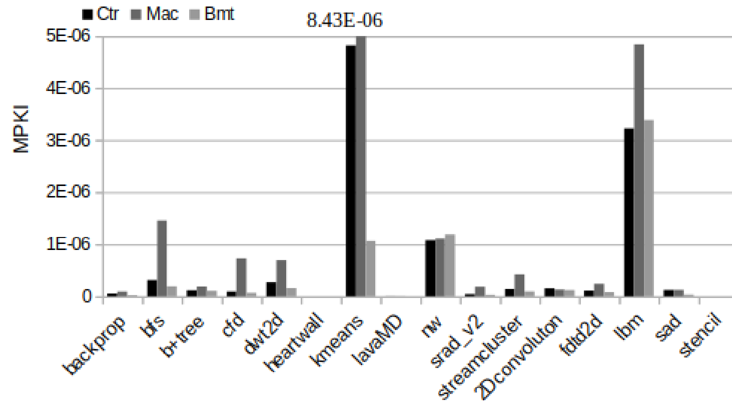

Figure 3.15: MDC miss rates (in MPKI) of the PSSM_nL2_4B_sMdc design.

its corresponding metadata, which contends for memory bandwidth. High L2 miss rates combined with high miss rates of metadata caches lead to high performance overheads, as we can see from Fig. 3.14. As a result, kmeans and lbm still show significant performance slowdowns even with our PSSM design.

Remaining Performance Bottleneck As we can see from Fig. 3.14, even with our best design, there is still some performance overhead, 16.84% on average for PSSM_nL2_SC_-32_sMdc. To better understand where the remaining overhead comes from, we model ideal MDCs under different scenarios. These ideal designs will limit one specific MDC resource and make the other MDC resource unlimited (meaning that there are only cold misses for these types of metadata). For example, the label **smallCtr** models a 2KB counter cache per partition while the MAC cache and the BMT cache have unlimited capacity. Similarly, the label **smallMac** and **smallBmt** means the MAC cache or BMT cache is 2KB while the others are unlimited. The label **large_mdc** models unlimited cache capacity for all the three types of metadata. We present the results in Fig. 3.16. From Fig. 3.16, we can see that the MAC accesses remain to be the main bottleneck. The reason is that the MACs incur the most storage overhead among all three types of metadata. At every memory read or write, the corresponding MACs must be accessed to verify the data or updated and any MAC cache miss would lead to additional bandwidth pressure.

Memory Encryption For the systems where data confidentiality is the main concern, we may forego data integrity protection to reduce the performance and the hardware overhead. With this application scenarios in mind, we evaluate different GPU memory encryption designs as listed in Table 3.5 and the results are shown in Fig. 3.17.

From Fig. 3.17, we can make the following observations. First, compare with the direct

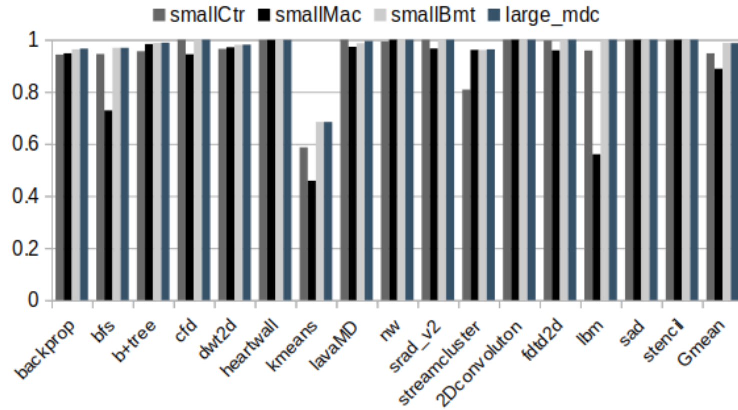

Figure 3.16: Normalized IPC of the PSSM_nL2_4B_sMdc design with different ideal MDCs.

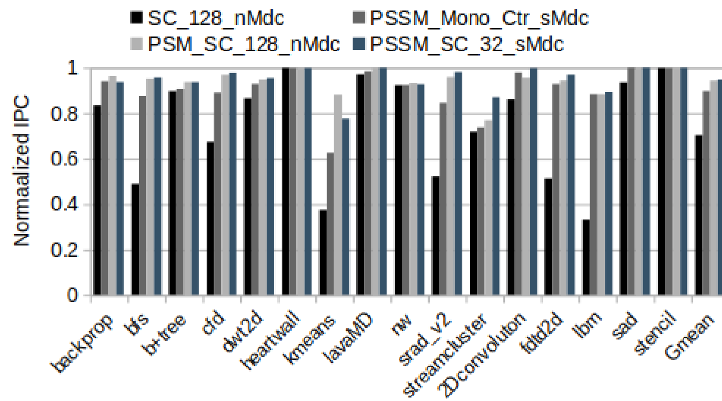

Figure 3.17: Normalized IPC of different GPU memory encryption schemes.

adoption of the split-counter mode encryption from the CPU (labeled 'SC_128_nMdc'), our PSSM designs can improve the performance significantly. The main reason is that the counter redundancy is totally eliminated and the memory bandwidth consumption for the counters is reduced significantly. Second, with our PSSM design, split-counter schemes perform better than the one using monolithic counters. The reason is that with monolithic counters, each 128B block needs a 32-bit counter. In comparison, with split counters, one 128B counter block is shared by 128 128B data blocks. Therefore, one 128B data block requires 8 bits as its counter storage overhead. The 4X higher counter storage in the monolithic counter scheme leads to higher pressure on the counter cache and subsequently higher bandwidth consumption. Third, with split counters in our PSSM design, our sectored counter organization, i.e., SC_32, performs better than the SC_128 organization on average. Hence, we conclude that if the focus is on data confidentiality, the

PSSM_SC_32 design would be the choice.

3.4 Conclusions

In this paper, we propose architectural designs for secure memory support on GPUs. Our performance analysis identifies that the partitioned memory architecture of GPUs is not compatible with conventional secure metadata and the existing counter block organization is not friendly to sectored cache structures. We propose Partitioned and Sectored Security Metadata, which is a simple-yet-effective approach to (a) use partition-local addresses for metadata and (b) reorganize the split-counter block to make it fit with sectored caches. Our results show that our proposed scheme effectively reduces the performance overheads of secure memory support on GPUs. Our study also reveals that even with our proposed scheme, memory bandwidth contention due to metadata accesses remains a performance bottleneck for memory intensive workloads. For systems only requiring data confidentiality, the metadata is reduced to only counters. In such a case, our proposed scheme incurs only 5.18% performance overhead on average.

CHAPTER

4

SHM: ADAPTIVE SECURITY SUPPORT FOR HETEROGENEOUS MEMORY ON GPUS

4.1 Motivation and Design Principles

4.1.1 Heterogeneous Memory on GPUs

To achieve high-throughput computation, GPUs have a complex heterogeneous memory system. It includes registers, local memory, global memory, constant memory, texture memory and several levels of caches. Among these different memory spaces, some are on-chip and do not need any protection as the GPU chip forms the trusted boundary; some are off-chip but have special access constraints during kernel execution, while the remaining ones are vulnerable to conventional physical attacks, and need strong protections.

We analyze the security mechanisms on CPU TEEs, and make the observation that GPUs may not always need freshness guarantee for some memory spaces due to their read-only nature during kernel execution. We show the summary of our analysis on Table 4.1. One observation we make is that the integrity tree does not need to cover read-only spaces like constant and texture memory.

Table 4.1: Security Mechanisms for GPU Heterogeneous Memory

Space	Location	Mechanisms
Register	on-chip	–
Local Memory	off-chip	C + I + F
Shared Memory	on-chip	–
Global Memory	off-chip	C + I + F
Constant Memory	off-chip	C + I
Texture Memory	off-chip	C + I (+ F)
Caches	on-chip	–

Table 4.2: Security Mechanisms for Application Data

Data	Property	Guarantees
Application code	Read-only	C + I
Input	Read-only	C + I
Output	Read/Write	C + I + F
In-flight Data	Read/Write	C + I + F

Moreover, as pointed out in previous work [Na21], some global memory data are most likely to be read-only. The reason is that GPU adopts a copy-then-execute model, and the data copied from host memory will not be updated anymore on device memory after the initial copy. For example, In OpenCL programs, the input buffer can be explicitly defined as read only. CUDA programs, however, allow the kernel code to modify the input, which necessitates the freshness checks. Toward this end, we also analyze the security protections from the application perspective and show it in Table 4.2. Similar to the constant and texture memory spaces, these read-only data also do not need freshness checks.

4.1.2 Seed Generation in Counter-Mode Encryption

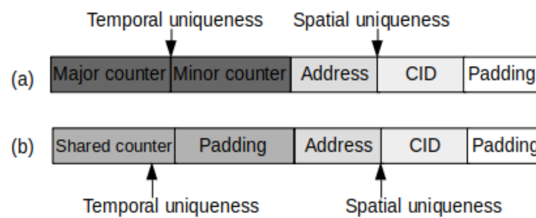

Figure 4.1: Seed generation for (a) not-read-only data and (b) read-only data.

Counter-mode encryption fundamentally requires that the counter used in each message encryption must be unique because counter reuse makes the encryption vulnerable. Hence, in counter-mode encryption, a per block counter is maintained and incremented at every LLC write back. Among the different components of the encryption seed that is fed into the AES engine, the counters are used to ensure the *temporal uniqueness*, while the address and CID are used to ensure the *spatial uniqueness*. A key observation for read-only memory regions, including constant memory, texture memory and some input data, is that these memory spaces are not modified during single kernel execution. In other words, there is no need to maintain the *temporal uniqueness* for read-only regions in single kernel execution. However, we identify a potential physical attack scheme, called *cross-kernel replay attack*, if the GPU context contains multiple kernels. In a multi-kernel workload, the read-only memory space (e.g., constant memory) may be reused (i.e., overwritten by the host) across different kernel invocations. An attacker can replay the read-only values from previous kernels if she/he has physical access. Hence, we need a mechanism to keep temporal uniqueness for read-only space in such scenarios. A shared counter is introduced for this purpose. For non-read-only memory, the full seed is still generated with split counters as shown in Fig. 4.1(a). For read-only regions, the major counter is replaced with a shared counter, which is stored on chip as a special register, and the minor counter are zero-padded (more details in Section 4.2). Since the shared counter is stored on-chip and is out of the reach of attackers, there is no need to check its integrity and freshness. As a result, the integrity tree does not need to cover the read-only data as shown in Fig. 4.2. Consequently, the memory bandwidth overheads due to integrity tree traversing are also eliminated.

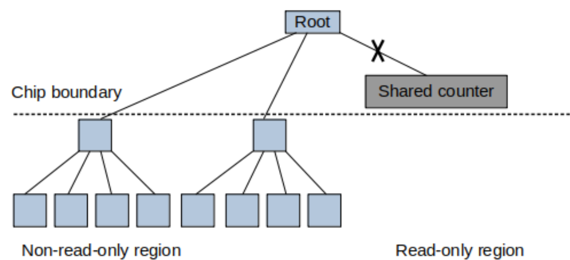

Figure 4.2: Integrity tree with read-only regions excluded.

4.1.3 Overhead of MAC Accesses

As pointed out by previous work [Yua21a], accessing the MACs can be a major overhead for secure GPU memory because at each off-chip memory read/write, the corresponding MAC block must be fetched/updated if it misses in the MAC cache.

To save the memory bandwidth for accessing MACs, PSSM [Yua21a] truncates the MAC from 8B to 4B. However, truncating the MAC reduces the collision space as proved by the birthday attack paradox [Gup15] – "With a birthday attack, it is possible to find a collision of a hash function in $\sqrt{2^n} = 2^{n/2}$, with 2^n being the classical preimage resistance security". As a result, with n = 50, it is possible to find a collision by every $\sqrt{2^{50}} = 2^{50/2} = 2^{25}$ memory updates. For a 4 GB device memory, there are $2^{32}/2^7 = 2^{25}$ memory blocks with the block size of 128B. If $n \leq 50$, there would likely be a collision if an attacker writes to all the blocks. In other words, the minimum size of MAC needs to be at least 50 bits to provide collision resistance if the MAC is generated for each cache line. CPU secure memory uses a 8B MAC per cache block. Directly adopting this MAC granularity to GPUs, however, incurs significant bandwidth pressure.

Our work exploits the unique memory access pattern in GPUs. As pointed out in previous work [Na21; Yua21b], GPU applications feature streaming data accesses. With streaming accesses, all the blocks within a memory region are accessed. The implication is that one 8B MAC can protect a larger memory chunk (e.g., one memory page) than a cache line/sector. The challenge, however, is that such a coarse-grain MAC would incur more bandwidth pressure for randomly accessed regions because at each MAC calculation, all the memory blocks within this memory chunk are needed. As shown in Fig. 4.3, although GPU features the streaming access pattern, there is still a significant portion of the memory accesses, which access memory in a non-streaming (or random) manner. To solve this problem, we propose dual-granularity MAC, in which an 8B MAC is maintained for each streaming accessed chunk, and an 8B MAC is maintained for each cache line within a random-accessed chunk.

4.2 Architecture Design

4.2.1 Overall Architecture

Similar to previous works [Yua21a; Yua21b; Na21], we assume that the GPU chip forms the TCB. The overall GPU secure memory architecture is shown in Fig. 4.4. We adopt a scheme

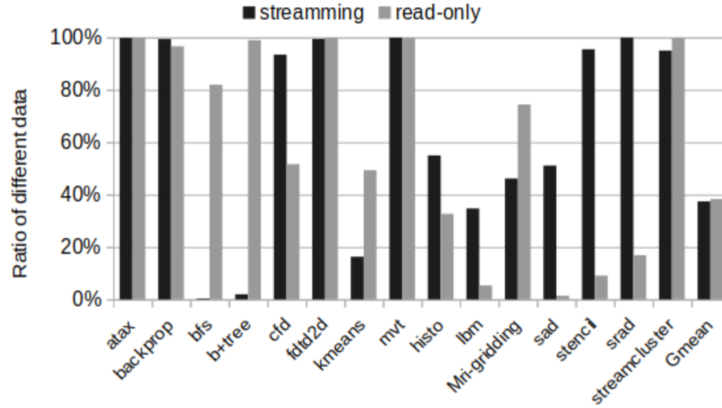

Figure 4.3: The ratio of memory accesses (i.e., L2 misses and L2 write backs) accessing streaming data as well as read-only data in various GPU workloads.

similar to PSSM [Yua21a], which integrates the memory encryption engine (MEE) into each memory controller and each MEE solely protects a single GDDR memory partition. The metadata caches (MDCs) including the counter cache, the MAC cache and the BMT cache, are embedded into each memory controller to save the bandwidth for accessing security metadata, which are generated using the partition local addresses to remove redundancy across partitions [Yua21a]. A secure root is stored in each partition for its corresponding integrity tree. A new on-chip shared counter is introduced as a special on-chip register, which is shared by all the read-only regions for encryption/decryption.

With the MEE on GPUs, each memory access is forwarded to the MEE, to encrypt/decrypt and authenticate the data. A key generator is also integrated onto the GPU command processor. When a GPU context is initialized, the key generator produces a key tuple (K_1, K_2, K_3) for memory encryption, memory integrity and integrity tree, respectively.

The security metadata is stored in off-chip GDDR memory. Compared with conventional CPU TEEs, we allocate space for dual-granularity MACs, per block MAC, which is calculated from each data cache line and its corresponding counters; and per-chunk MAC, which is produced by hashing the per block MAC within this chunk. During GPU context initialization, both per chunk and per cache line MACs are calculated and written into the device memory since we assume streaming accesses by default. At runtime, the hardware predicts the memory access patterns, and makes the decision of fetching either the per block MAC or per chunk MAC to verify the data read from off-chip memory. Note that for the read-only regions, neither the per block MAC nor per chunk MAC will be updated during kernel execution.

To adaptively select the data protection mechanisms, the GPU hardware needs to be aware of the data type (i.e., read only or not) and the pattern (i.e., streaming or not). Hence, we propose hardware-based detection schemes to detect the read-only regions and streaming accessed chunks, as shown in Fig. 4.5, which illustrates the design in one memory partition. In our baseline GPU, there are two L2 banks in each partition. In each memory partition, we maintain two prediction bit vectors, one as read-only predictor and the other as streaming predictor, and several memory access trackers (MAT) to detect the streaming access pattern. For the read-only predictor, the bit vector is maintained with the granularity of a memory region with the region size of M kB (e.g., M = 16) using local addresses. Here, we use the terminology from [Yua21a], where a local address means the offset within a partition after the physical address is mapped to partition ID and partition offset. The streaming data prediction vector is maintained with the granularity of a memory chunk (e.g., 4 KB) using local addresses.

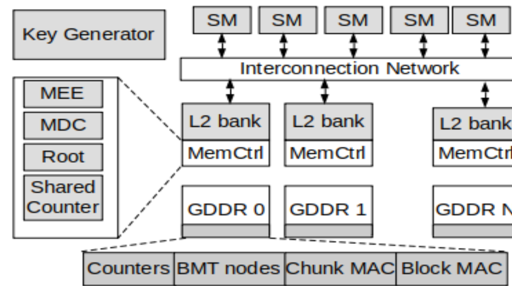

Figure 4.4: Overall architecture.

4.2.2 Detecting Read-only Regions

To detect read-only regions at runtime, we use an N-entry bit vector, which is indexed with the region ID. For example, with the region size of 16 KB, the least significant 14 bits of a local address will be ignored and the next $log_2 N$ bits are the index to the bit vector. All the entries in the bit vector are initialized to 0, representing not-read-only by default. During GPU context initialization, when the command processor allocates the memory space for the input region, all the regions updated by CUDA memory copy APIs will be set to be as read only by setting the bit vector entries to 1. If the GPU programming model is able to provide additional information on different regions (for example, the input buffer of openCL

programs), the corresponding bit vector entries can also be initialized by the command processor. In our evaluation, we do not assume such support from the programming model or compiler.

During kernel execution, once a memory region is updated by a store instruction or another CUDA memory copy API, the corresponding bit in the bit vector will be reset to 0, indicating that this region is not read only. Since all read-only regions share a single on-chip counter, once a region is detected as not read only, we need to resort to the per block counters, whose values will be propagated from the shared counter. To do so, we reserve the counter storage space in the off-chip memory as if all the protected space would use per block counters. Note that although allocated, the per block counters corresponding to read-only regions are not accessed. If a region transits from read-only to not-read-only, the shared counter will be copied as the major counter for this memory region, and the minor counter corresponding to the block to be updated will be incremented by one from the padding value (0 by default). Simultaneously, the minor counter of other blocks within this region will be set as the padding value. Fig. 4.6 shows such an example. During step (a), a memory region A is in read-only state (the corresponding bit in read-only vector is 1), and the shared counter value is 3. In step (b), when a write request (i.e., a write to A[2] or the third cache line in region A) is sent to the memory partition where region A is located, the bit in the read-only bit vector will be reset to 0 immediately, indicating per-block counters will be used for region A afterwards. In the meanwhile, counter update requests are generated to update all the major counter corresponding to region A as the value of shared counter and the minor counter corresponding to block A[2] will be incremented by 1 from the padding value. These updates occur directly in the counter cache. In our example, the shared counter is 3 and the padding value has been initialized as 0. Therefore, the counter update increments the block counter for A[2] by 1, and sets its corresponding major counter as 3. In step (c), there is another update to A[1], i.e., the second block in region A. Since per block counters have been used, the corresponding minor counter is incremented as shown in the figure. During step (b), after propagating the per block counters, the BMT also needs to be updated to cover the newly added region by traversing from BMT leaves to the root. This is achieved naturally as a result of counter updates.

Since our bit vector is indexed with region id and we do not keep tag information, it is possible that different regions map to the same bit in the bit vector. This would lead to lost opportunities for bandwidth saving but will not affect the security or correctness. The reason is that we only allow a chunk to transit from read-only to not-read-only. As a result, conflicts in the bit vector can only miss-classify a read-only region as not-read-only. In this

case, per block counters are used although all the counter values would be 0.

As mentioned above, in our read-only detection scheme, once a region is detected as not-read-only, it will always stay in this way. This may be over pessimistic in recognizing read-only regions. When analyzing the GPU workloads, we found that some multi-kernel applications may reuse the input region such that right before each kernel invocation, new input data from the host are copied to the same device location and such inputs are read only during kernel execution. Following our scheme, however, once a region is overwritten, it will be recognized as not-read-only. To recover such opportunities, we propose to a new API, $InputReadOnlyReset(addressrange)$, which informs the command processor to (a) reset the regions within the specified address range as read only, and (b) reset the shared counter value to the maximum major counter value within this specific range to avoid counter reuse. The reason for resetting the shared counter is to avoid the abuse of this API for cross-kernel replay attacks discussed in Section 4.1.2. To reset the shared counter value, the command processor need to issue a request to the memory controller and scan the counter values for the regions specified by this new API. This process can be illustrated in Fig. 4.7. When a memory region, i.e, $addr_range$, is reset to be read-only by this API, the corresponding counter region is scanned and the maximum per block major counter value is returned (90, in this case), and this maximum per block counter is then compared with the shared counter value to update the on-chip shared counter. As showed in previous work [Na21], the memory scanning overhead is typically negligible due to the high bandwidth accesses of consecutive memory locations.

The consequence of altering the shared counter is that the previously detected read-only regions cannot be reused as they are encrypted with the old shared counter value. In our study, we found that the multi-kernel workloads completely overwrite the input region and do not reuse read-only regions. For a workload with such read-only region reuses, we can choose to (a) not take advantage of this new API and treat the otherwise read-only regions as not-read-only, and (b) re-encrypt the affected region with the new shared counter value. Note that resetting a not-read-only region to read only has no impact on the BMT, as the affected path is simply not traversed if the region is indeed read only. If not, any update to the region will make it not-read-only and update the per-block counters, which induces BMT traversing to the root.

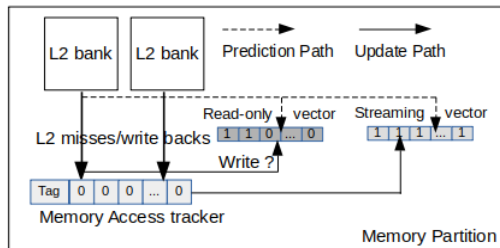

Figure 4.5: The read-only detector and streaming detector in a memory partition. Their inputs are the LLC misses and write backs.

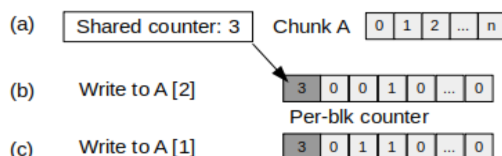

Figure 4.6: An example showing the propagation from the shared counter to the per block counters.

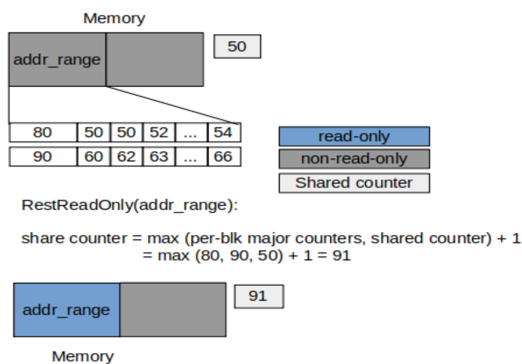

Figure 4.7: The process of shared counter update when using the read only reset API.

4.2.3 Detecting Streaming Accessed Chunks

The purpose of streaming access detection is to use dual-granularity MACs, i.e., coarse-grain MAC (i.e., per chunk MAC) for streaming-accessed chunks and fine-grain MAC (i.e., per block MAC) for random-accessed ones, to reduce the MAC access bandwidth. To support dual-granularity MACs, we reserve space for both MACs and access only one of them at runtime based on the access pattern.

Our hardware scheme to detect streaming accessed chunks is shown in Fig. 4.5. It contains two components. The first one is a bit vector indexed by local chunk IDs to predict whether a chunk is streaming accessed or not. The second one is chunk-level memory access trackers, each of which contains a chunk tag, a 1-bit write flag and a set of counters to

monitor the block access patterns within a chunk. Since GPU applications feature streaming accesses, we eagerly initialize the bit vector predictor to all 1s, indicating all chunks are streaming accessed. Whenever there are memory accesses, i.e., L2 misses or L2 write backs, a memory access tracker will be used to start monitoring the memory access pattern in the corresponding chunk. In our design, a chunk-level access tracker has an array (32 entries) of 1-bit counters. The 1-bit write flag is set whenever there is a write back in the chunk. We maintain N memory access trackers in each memory partition (we use N as 8 in our experiments). In other words, our design can concurrently monitoring N chunks in each memory partition.

When a memory access (i.e., an L2 miss or write back) happens to a memory partition, we check the bit vector to see whether the corresponding chunk is streaming accessed or not. If not, the chunk is predicted as random accessed, and we will fetch the block-level MACs for integrity verification (i.e., compared it with the MAC computed from the fetched data block). If the chunk is predicted as streaming accessed, the chunk-level MAC will be fetched and used. More specifically, the fetched regular data block (or the dirty eviction block) will be used to compute the block-level MAC, which is stored in the MAC cache. When the pattern detection (explained next) result is available, if the chunk is streaming accessed, the block-level MACs in the MAC cache are used to produced the chunk-level MAC, which is then compared with the chunk-level MAC fetched from memory. For a write stream, all the blocks are verified first with the old chuck-level MAC and then each block produces its block-level MACs, which are used to produce the new chunk-level MAC. The updated MACs, either chunk- or block-level, are stored in the MAC caches. The updated block-level MACs of a streaming accessed chunk are marked 'not dirty' in the MAC cache so as to eliminate the traffic overhead due to block-level MACs for streaming accessed chunks.

In the meanwhile of using either block- or chunk-level MAC for integrity verification, we start monitoring the subsequent memory accesses to determine whether the chunk is streaming accessed or not. To do so, the tag is set and only accesses to the same chunk will update the access counters based on their chunk offsets at the cache block/line granularity. At the end of the monitoring phase of K memory accesses, the counters in a tracker are examined. For the chunk size of 4kB, we choose $K = 32$. We also introduce a time-out scheme to prevent a randomly accessed chunk from occupying a memory access tracker for a long time (6K cycles) without reaching the K accesses. After time out, the counters are examined the same way as if we reach the end of a monitoring phase. The following criterion is used to determine whether a chunk is streaming accessed or not. For an access tracker, if all the blocks in the chunk have been accessed (i.e., all access counters are non

zero), the chunk is considered streaming accessed since all of its blocked are touched. If some blocks have non-zero accesses while others in the same chunk are not accessed at all (i.e., some access counters being 0), we consider this chunk as random-accessed. The bit vector is then updated accordingly. Also, if the write flag is set for the chunk, we know that there is at least one write back to the chunk. If the detected pattern is streaming, we need to re-produce and update the chuck-level MAC.

It is possible that one random-accessed chunk is miss-classified/mispredicted as streaming accessed or vice versa. The handling of mispredictions is dependent upon whether the access is a read access or write access and whether the accessed chunk is read-only or not. We list the different scenarios in Table III and Table IV. The read-only information of the chuck is retrieved from the read-only bit vector (Section IV-B). For a correct prediction, i.e., the predicted stream/random pattern matching the detected outcome, either the per chunk MAC or per block MAC is fetched/updated and there will be no additional bandwidth overheads.

For a read access in a read-only region, when a random pattern (i.e., detected as random) is mispredicted as streaming (i.e., predicted as stream), besides fetching the chunk-level MAC, the secure memory engine needs to re-fetch the per-block MAC to verify the data. When a streaming pattern is mispredicted as random, there is no additional bandwidth overheads since the per-block MACs are always up to date for read-only regions. For a read access in a non-read-only region, when a random pattern is mispredicted as streaming, the secure memory engine needs to fetch the chunk-MAC. Upon the detection of the misprediction, however, the per-block MACs in the chunk are to be updated because the predictor entry is updated as 'random' and the per-block MACs will be used from now on. To do so, all the data blocks in the chunk need to be re-fetched (and validated with the chunk-level MAC) to produce the updated block-level MACs. On the other hand, when a streaming pattern is mispredicted as random, the secure memory engine just re-fetches and re-produces the corresponding chunk-level MAC as all the blocks in the chunk are accessed and validated with block-level MACs (due to the streaming access).

Predictions from write accesses are treated similar to read accesses to a non-read-only region. For a write access, when a random pattern is mispredicted as streaming, the secure memory engine fetches all the blocks in the chunk from off-chip memory, and updates all the per-block MACs. When a streaming pattern is mispredicted as random, the secure memory engine just updates the chunk-level MAC. When updating the chunk-level MAC, the updated block-level MACs in the MAC cache are marked 'not dirty'.

A more subtle issue, however, occurs when chunks with different access patterns conflict

Table 4.3: Handling Streaming Predictions for Read Accesses

Prediction	Action	Detection	Read-Only	Bandwidth Overheads
Stream	Fetch chunk MAC	Stream	Y/N	Zero
Stream	Fetch chunk MAC	Random	Yes	Re-fetch blk-MAC
Stream	Fetch chunk MAC	Random	No	Re-fetch all the data blocks in the chunk
Random	Fetch blk MAC	Random	Y/N	Zero
Random	Fetch blk MAC	Stream	Yes	Zero
Random	Fetch blk MAC	Stream	No	Re-fetch chunk-level MAC

at the bit vector. For example, chunk A is streaming accessed while chunk B is random accessed. Both A and B share the same index to the bit vector due to the limited length of the bit vector. After chunk A updates its chunk-level MAC and the bit vector entry is set to 1 (i.e., streaming), when chunk B is accessed, its chunk-level MAC will be accessed as a result. However, as chunk B was previously treated as random accessed, its chunk-level MAC can be out of date although its per block MACs are up-to-date. Due to the out-of-date MAC, the integrity verification would fail. There are two remedies for this issue. One is to always update both chunk-level and block-level MACs. This solution essentially trades write traffic for read traffic and may lead to performance degradation for write-intensive workloads. The second solution is that if one integrity check fails, the other MAC needs to be checked. This way, as long as one of the duel-granularity MACs is up-to-date, the integrity check would be successful. If the number of such conflicts, i.e., chunks with different MAC granularity mapping to the same entry in the bit vector, is small, the performance impact would be limited. In our work, we choose the second solution.

4.2.4 Using L2 as Victim Cache for Security Metadata

In our study, we observe that some GPU applications do not utilize the L2 cache well. Either it is underutilized or it suffers from very high miss rates due to poor temporal locality. Actually, streaming accesses have little data reuse and would lead to high L2 miss rates. In

Table 4.4: Handling Streaming Predictions for Write Accesses

Prediction	**Action**	**Detection**	**Action**	**Bandwidth Overheads**
Stream	Produce blk MAC	Stream	Produce and update chunk MAC	Zero
Stream	Produce blk MAC	Random	Update blk MAC	Re-fetch data and produce the blk-MAC
Random	Produce blk MAC	Random	Update blk MAC	Zero
Random	Produce blk MAC	Stream	Produce and update chunk MAC	Zero

such cases, we propose to use the L2 cache as a victim cache for security metadata caches, especially the MAC cache. The rationale is that a MAC block (128B) would contain sixteen block-/chunk-level MACs (128B = 16x8B) and would provide more reuse opportunities than a 128B data block.

To ensure that the victim cache traffic would not interfere with regular data traffic, we dynamically enable L2 as the victim cache only if the regular data miss rate is very high (e.g., 90%). To collect accurate data miss rates, we reserve a small portion of the L2 cache lines such that they are only accessed with regular data accesses, similar to the set sampling approach used in [Qur06].

4.3 Methodology

We model our proposed schemes with GPGPU-Sim v4.0 [Kha20]. Our baseline GPU configuration is shown in Table 4.5, which is based on the Nvidia Turing architecture [Nvi21]. We assume a range of 4GB device memory to be protected by the secure memory engine.

Our baseline secure memory support is modeled based on PSSM [Yua21a], in which the partition-local offset is used to construct the security metadata. The detailed MDC and MEE organizations are listed in Table 4.6.

Our benchmarks are from the Rodinia-3.1 [Che09], Parboil [Str09] and Polybench [GG09] benchmark suites, and cover a wide range of workloads with different memory utilization as well as heterogeneous memory usage. Since computation intensive workloads are not

Table 4.5: Baseline GPU Configuration

SM config	30 SMs, 1506MHz
Register File	256KB/SM, 7.5MB in total
L1 D-Cache / Shared Memory	96KB/SM
L2 cache	2 banks per memory partition, each L2 cache bank is 128KB, 3MB in total. For each L2 bank, 192 MSHR entries, and each entry can merge 16 requests.
DRAM	3500MHz, 12 partitions, 336GB/s.

Table 4.6: MDC and MEE Organization

Counter cache	2KB / memory partition, 128B blk, 4-way sectored, 256 MSHRs, write-allocate policy
Mac cache	2KB / memory partition, 128B blk, 4-way sectored, 256 MSHRs, write-allocate policy.
Bonsai Merkle Tree cache	2KB / memory partition, 128B blk, 4-way sectored, 256 MSHRs, write-allocate policy.
Hash/Mac latency	40 cycles
AES engines	1 pipelined AES/memory partition

Table 4.7: Benchmarks

Benchmark	**Bandwidth Utilization**	**Memory Space**
atax	23%	constant
backprop	27%-50%	constant
bfs	15% -50%	constant
b+tree	12%-15%	constant
cfd	27%-75%	constant
fdtd2d	90%-93%	constant
kmeans	67%-81%	constant/texture
mvt	22%	constant
histo	55%	constant
lbm	95%	constant
mri-gridding	30%-47%	constant
sad	17%	constant/texture
stencil	11%-42%	constant
srad	20%-22%	constant
srad_v2	72%-78%	constant
streamcluster	78%	constant

Table 4.8: Evaluated designs for GPU secure memory with both memory encryption and integrity verification.

Scheme	What It Represents
Naive	Baseline GPU with secure memory, and the security metadata is organized with physical address.
Common_ctr	Secure GPU memory with common counters [Na21] scheme, and the security metadata is constructed with physical address.
PSSM	Secure GPU memory with PSSM scheme [Yua21a]
PSSM_cctr	Secure GPU memory with common counters scheme, and the security metadata is constructed from local address as PSSM [Yua21a] design.
SHM	Our secure heterogeneous memory design, with the PSSM scheme to construct security metadata.
SHM_cctr	Our secure heterogeneous memory design, combined with common counters scheme.
SHM_vL2	Our secure heterogeneous memory design, and using L2 cache as the victim cache for all the security metadata.
SHM_readOnly	Our secure heterogeneous memory design, which use per-blk MAC, but use shared counter to optimize the overheads of encryption counters and BMT.
SHM_upper_-bound	Our secure heterogeneous memory design, with unlimited MATs and unlimited predictor sizes, and the predictors are initialized with L2 miss/write back profiling.

sensitive to secure memory, we choose 15 memory intensive workloads and report the benchmark details in Table 4.7, including the bandwidth utilization, and different memory spaces usage. Global and local memory are used by all workloads, and hence we only report the constant and texture memory usage. For benchmarks with low simulation time, we simulate the entire benchmarks; for benchmarks with long simulation time, we simulate the first 6 million cycles.

The different secure memory designs that we evaluate in our experiments are listed in Table 4.8. We report normalized Instructions per cycle (IPC) in our evaluation with the baseline being the GPU with sectored data caches and without secure memory support. By default, we assume 8B MAC per cache line. Similar to the state-of-the-art CPU secure memory, the data fetched from memory is sent to the GPU cores without waiting for integrity verification results. An exception will be thrown if a verification failure occurs.

Table 4.9: Hardware Overhead

Hardware	Tag	Write Flag	Entries	Entry size
read-only predictor	-	-	1024	1 bit
streaming predictor	-	-	2048	1 bit
access tracker	20 bits	1 bit	32	1 bit

4.3.1 Hardware Overheads

The storage overhead of our proposed hardware components are listed in Table 4.9. In our design, the read-only predictor has 1024 entries, thereby 128B in total. The read-only predictor is maintained in the 16 KB granularity. The streaming access predictor has 2048 entries, thereby 256B in total. The streaming predictor is maintained in the granularity of 4 KB. For the access tracker, each access tracker has a 20-bit tag (32-bit local addresses and 4kB chunk size) and 32 1-bit counters to record the number of accesses, and 1-bit write flag, which is set for a write access. To track the end of a monitoring phase, each memory access tracker also needs a 5-bit access counter and 13 bit time-out counter. Therefore, each access tracker needs 20 + 1 + 32 + 18 = 71 bits. We use 8 memory access trackers in our design. To summarize, each memory partition maintains one read-only predictor (128B), one streaming access predictor (256B), and 8 memory access trackers (8×71-bit = 71B). With 12 partitions, the total overhead is 5,460B (5.33 KB).

4.4 Evaluation

4.4.1 Read Only Prediction

We first evaluate our read-only prediction scheme. We use our read-only predictor to predict each memory access (including all L2 misses and L2 writebacks), and compare the predictions with the results from offline profiling. We show the accuracy of our read-only prediction scheme in Fig. 4.8. As we can see that our scheme can capture the read-only region reasonably well, 89.31% on average. We further break down the prediction results into three parts: correct predictions (labeled as 'Correct-Prediction'), mispredictions due to initialization (labeled as 'MP_Init'), mispredictions due to aliasing in the predictor (labeled as 'MP_Aliasing'). As we can see from the figure, mispredictions due to initialization

contribute to most mispredictions in read-only regions, while the mispredictions due to aliasing in the predictor are negligible.

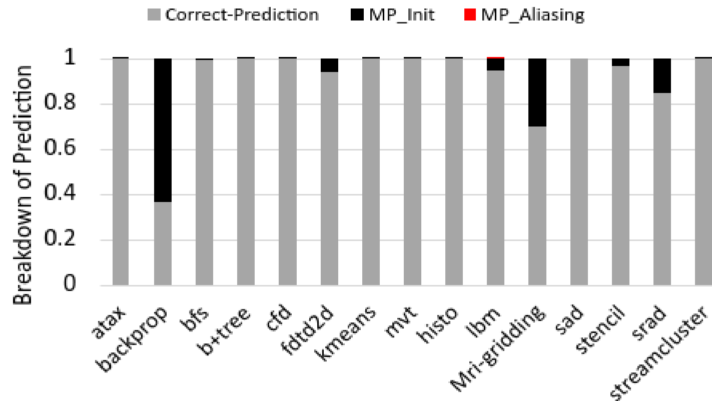

Figure 4.8: Breakdown of read-only predictions.

4.4.2 Streaming Access Pattern Detection

We use 8 memory access trackers in each memory partition and report the results in Fig. 4.9. We measure the accuracy of streaming pattern prediction with an oracle memory access tracker, which has unlimited capacity to detect the pattern of every memory chunk. For each memory access (either L2 miss or L2 write back), if the detection result agrees with the prediction result, the prediction is considered a correct one, otherwise it is a misprediction. We count all correct predictions and mispredictions to calculate the prediction accuracy. The prediction accuracy results are shown in Fig. 4.9. As shown in the figure, our design can achieve good prediction accuracy, 83.36% on average. We break down the predictions for streaming patterns into five parts: correct predictions (labeled as 'Correct-Prediction'), mispredictions due to initialization (labeled as 'MP_Init'), mispredictions due to runtime pattern change in read-only regions (labeled as 'MP_Runtime_Read_Only'), mispredictions due to runtime pattern change in non-read-only regions (labeled as 'MP_Runtime_Non_-Read_Only'), and mispredictions due to aliasing in the predictors (labeled as 'MP_Aliasing'). As we can see from the figure, for streaming pattern prediction, some benchmarks suffer from high misprediction rates due to initialization of the predictor, while some other benchmarks show high mispredictions due to runtime pattern changes. As discussed in Section

4.2, not all mispredictions incur the same bandwidth overheads.

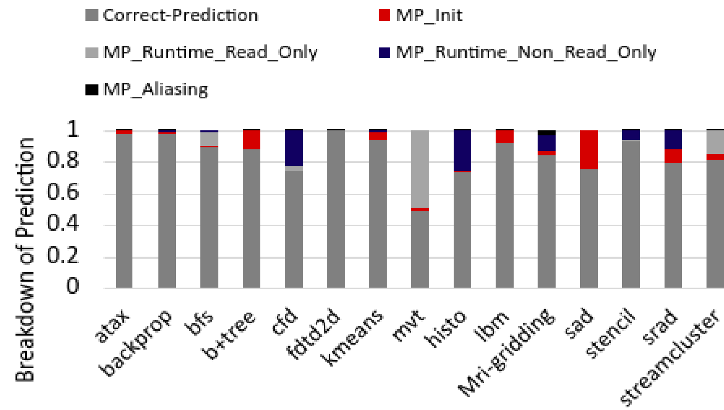

Figure 4.9: Breakdown of streaming pattern predictions

4.4.3 Overall Performance

We evaluate our secure heterogeneous memory design and compare it with different previous works, as shown in Fig. 4.10. From the figure, we can make the following observations. First, the naive design, labeled as 'Naive', in which the security metadata is constructed with physical addresses as conventional CPU secure memory, degrades the GPU performance by 53.9% on average. Second, compared with the naive secure memory design, the common counters scheme (labeled as 'Common_ctr') can improve the performance and reduce the overheads of secure GPU memory to 49.4%. It is more effective for the workloads with a large portion of streaming access pattern such as atax (23.1%) and mvt (19.4%). The reason is that the common counters scheme significantly reduces the memory bandwidth for accessing the encryption counters. However, there still exists a high overhead after common counters optimization because the security metadata is constructed from physical addresses and the same security metadata is accessed by different memory partitions, leading to redundant memory traffic. Third, compared with common counters, PSSM improves the performance significantly, reducing the performance overheads to 18.6% on average, and the reason is that it eliminates the redundancy and adopts the sectored design to save the memory bandwidth [Yua21a]. However, as the MAC is produced in the granularity of a cache line, it remains to be a major overhead. Fourth, our proposed design, labeled as 'SHM', improves the performance significantly, and further reduces the overheads to 8.09% on average, and

for most workloads, SHM can reduce the overheads to less than 5%. The reason is that, our design leverages read-only data and uses a dual-granularity MAC, and can effectively optimize the overheads. However, for the workloads that feature random access patterns and write-intensive memory footprints such as bfs, lbm and mri-gridding, our SHM scheme still shows relatively high overheads. The reasons include (1) the MACs are maintained at the block granularity, and each memory access needs to verify/update the MAC; and (2) these benchmarks are write intensive, and the per-block counters are maintained for these workloads. Fifth, our upper bound analysis, labeled as 'SHM_upper_bound', shows that the performance overheads of our SHM design are very close to the idealized design: the SHM design with unlimited capacity of predictor sizes has 6.76% overheads on average, which is quite close to our SHM design.

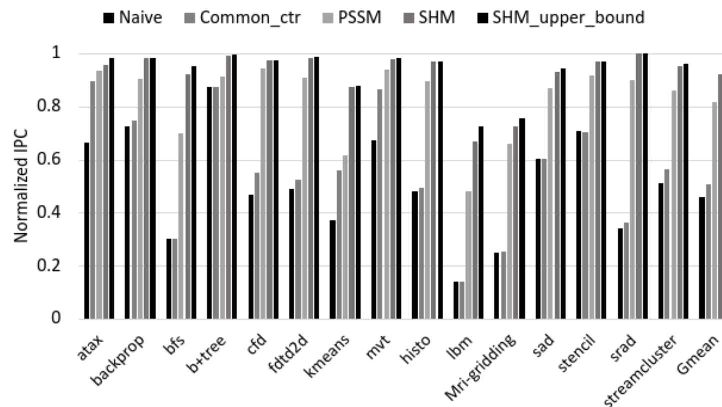

Figure 4.10: Normalized IPC of different secure GPU memory designs.

4.4.4 Performance Breakdown

To examine the effectiveness of different optimizations, we include them one at a time and show the results in Fig. 4.11. From Fig. 4.11, we can make the following observations. First, the combination of common counters and PSSM is beneficial. It reduces the performance overhead by 1.2% on average compared to PSSM. Second, compared with PSSM, our optimization for read-only region (labeled as 'SHM_readOnly'), including the constant memory, texture memory and instruction memory can be very effective and further reduce the performance overhead by 2.5% on average, the reason is that our SHM scheme does

not need to maintain per-block counters and does not need to traverse the integrity tree for read-only regions. Consequently, the memory bandwidth for both encryption counters and integrity tree can be saved. This scheme can be very effective for some benchmarks that highly utilize the read-only memory spaces like constant memory and texture memory. For example, the benchmark kmeans shows more than 14% performance improvement compared with PSSM, when the optimization to read-only memory space is applied. A detailed L2 miss breakdown shows that among all the L2 cache misses, texture memory accesses contribute to more than 27.75% L2 misses for kmeans. Third, when our dual-granularity MAC is applied, the MAC bandwidth is reduced for the reasons that have been discussed in Section 4.4.3. Fourth, our SHM design is compatible with the common counters scheme. As we can see from Fig. 4.11, adding the common counters scheme onto our SHM scheme (labeled as SHM_Cctr) can further reduce the performance overhead for secure GPU memory by 0.4% on average.

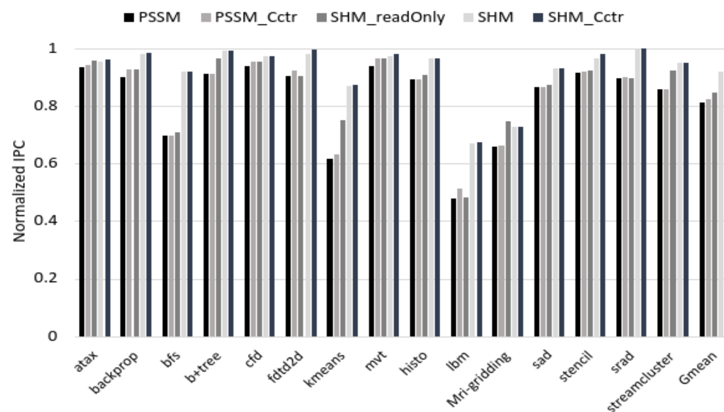

Figure 4.11: Performance impacts of different optimizations.

4.4.5 Bandwidth Saving

We compare the bandwidth overhead (including all the security metadata access and additional data accesses due to mispredictions in our SHM design) for different designs, as shown in Fig. 4.12. The bandwidth is obtained by counting the number of bytes for different security metadata fetched/updated from/to DRAM, and dividing them by the execution time. Then, we normalize them to the regular data bandwidth. We can see that our SHM

design significantly reduces the bandwidth overhead compared to the naive design, from 189.07% (naive secure GPU memory design) to only 5.95% on average. For benchmarks with high bandwidth utilization, the reduced bandwidth overheads directly translate to performance gains as reported in Fig. 4.10. On the other hand, for the benchmarks with relatively low bandwidth utilization, such as atax or mvt (Table 4.7), the high bandwidth reduction from SHM leads to relatively small performance gains.

We also isolate the bandwidth savings from the two optimizations, i.e., read-only and streaming data optimization, in our scheme. First, compared with PSSM design (17.1% bandwidth overhead on average), SHM_readOnly reduce the bandwidth overheads to 13.2% because our read-only optimization can reduce the bandwidth for both encryption counters and integrity trees. Second, compared with SHM_readOnly, our SHM design further reduces the bandwidth overheads to only 5.95% on average because our SHM design significantly reduces the bandwidth requirements for MACs. Taking the benchmark fdtd2d as an example, our SHM scheme can achieve near zero (0.78% in total) bandwidth overheads. The reasons are (1) fdtd2d has 99.87% read-only accesses in GDDR memory as we can see from Fig. 4.3, and our read-only optimization reduces the overheads of counters and BMTs to near zero (0.44%); (2) fdtd2d also features perfect streaming (99.35% of off-chip memory accesses are streaming) data access patterns as we can see from Fig. 4.3. With SHM, only the chunk-level MACs (8B MAC per 4KB chunk) need to be transferred over the GDDR memory, which reduces the MACs bandwidth overheads to 0.34%; and (3) the streaming prediction accuracy for fdtd2d is 99.69%, meaning almost no additional bandwidth overheads due to mispredictions.

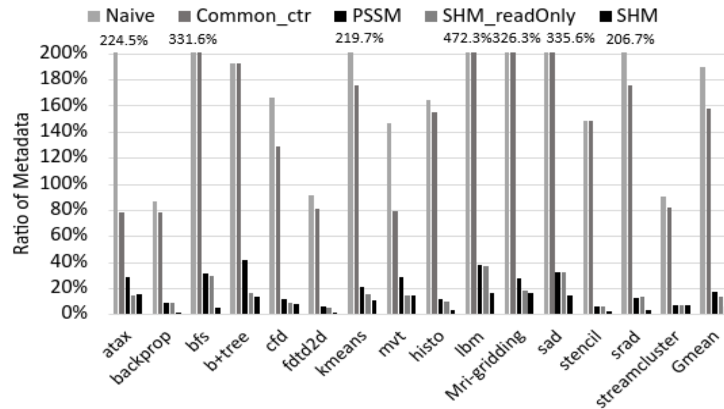

Figure 4.12: Bandwidth overheads due to security metadata, normalized to regular data bandwidth, of different designs.

4.4.6 Power Saving

We report the power efficiency of our SHM design, and compare it with the prior works, as shown in Fig. 4.13. We extend the GPUWattch [Len13] to model the power and energy consumption of different designs. We use CACTI_v6.5 [NM09] to evaluate the power/energy consumption of metadata caches (32 nm technology). our energy model includes all the GPU components and the metadata caches while the energy consumption of the AES and MAC engines are not included. We accumulate the total energy of the kernels and calculate the energy per instruction for different secure GPU memory designs, and normalize it to the baseline GPU without secure memory support. As we can see from the figure, compared to the naive secure GPU memory design, our SHM design reduces the normalized energy consumption per instruction from 215.06% to 106.09% on average. In other words, the energy overhead of our SHM scheme is 6.09% compared to the baseline GPU without secure memory support.

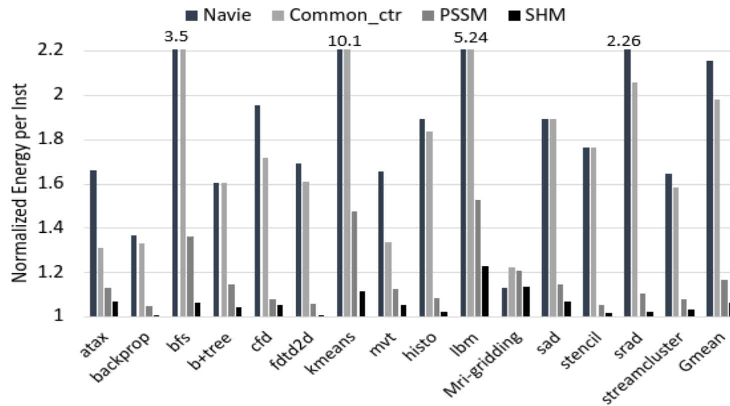

Figure 4.13: Normalized Energy Consumption per Instruction for different designs.

4.4.7 Using L2 as a Victim Cache

We present our results of using L2 as the victim cache for security metadata caches in Fig. 4.14. We dynamically sample the miss rate from the L2 cache in each memory partition, and enable this feature only when the sampled L2 miss rate is higher than 90%. For benchmarks with multiple kernels, the sampling counters are reset after each kernel execution. From Fig. 4.14, we make the observation that using L2 as the victim cache for security

metadata can further reduce the performance overhead by 0.65% on average. This scheme is more effective for memory-intensive benchmarks that suffer very high L2 miss rates, e.g., 4% performance improvement for the benchmark lbm and 3.4% for the benchmark sad.

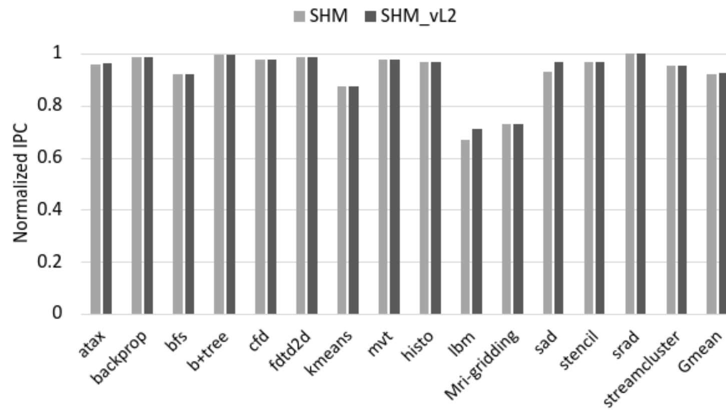

Figure 4.14: Normalized IPC when enabling L2 as a victim cache for security metadata.

4.5 Conclusions

Security metadata traffic is the key performance bottleneck for GPU secure memory. In this paper, we propose adaptive security support for GPU heterogeneous memory to reduce the performance overhead. First, we point out that read only regions do not need freshness protections as they are immune to replay attacks. By letting all read-only regions share an on-chip counter, we can reduce the traffic of counters and BMT. To optimize bandwidth for MAC access, we propose dual-granularity MACs with coarse-grain MACs for streaming-accessed regions and fine-grain MACs for random-accessed regions. Our hardware design consists of two lightweight predictors to detect read-only regions and streaming-accessed regions so as to adapt the security mechanisms accordingly. Our evaluation results show that it outperforms the state-of-the-art schemes: by up to 41.63% and 9.5% on average compared to PSSM and 84.04% on average compared to common counters for memory-intensive workloads.

CHAPTER

5

DELTA COUNTER: BANDWIDTH-EFFICIENT ENCRYPTION COUNTER REPRESENTATION FOR SECURE GPU MEMORY

5.1 Summary

The security of accelerators such as GPUs has recently gained significant attention with their wide adoption in the cloud since they are vulnerable to physical attacks. To support secure memory for GPUs, the critical performance bottleneck is the memory bandwidth contention between the regular data and the security metadata [Yua21b]. With counter-mode encryption, the security meta includes counters, the Bonsai Merkle Tree (BMT), and message authentication codes (MACs).

In this work, we focus on encryption counters given their impact on the counter and BMT traffic while leveraging prior schemes [Sai18; Taa18a] to address the MAC traffic. We

first analyze the characteristics of the encryption counters from a wide range of GPGPU benchmarks and make two key observations. (1) With the split counter scheme, the cache blocks in a large portion of the memory space, sometimes the entire GPU memory space, share the same major counter value. (2) The difference among minor counters is fairly limited. We then propose a novel scheme to reduce the encryption counter traffic. Our design includes (a) a highly compact way of counter representation and (b) a verification scheme to determine the correct minor counter values. In our design, we use a few on-chip registers to hold the major counters and use a (7-bit) base value along with two small (2-bit) deltas to represent the minor counters in a large memory chunk, one delta for the most frequent delta between minor counters and the base, the other delta for the maximal difference between a minor counter and the base. This way, for a large memory chunk (e.g., 16kB), the counter overhead becomes nearly negligible (less than 2B). We then leverage the existing MAC verification logic to verify the minor counters computed from the base and deltas. Our approach essentially trades off decryption and integrity-check latency for reduced counter-data traffic to take advantage of the latency-hiding nature of GPUs. Compared to prior works on reducing counter traffic [Na21], our scheme handles more counter value patterns (as we don't restrict the counters to be the same in a memory chunk) and is more effective in reducing counter traffic.

Our study also reveals that the GPU memory data are typically compressible. As a result, we can co-locate the MACs with the compressed cache blocks, similar to [Taa18a]. Our experimental results show that our proposed delta counter scheme significantly reduces the storage and bandwidth overheads of encryption counters and achieves secure GPU memory with an average performance overhead of 2.01% compared to GPU without security support. Our delta scheme is also compatible with SYNERGY [Sai18], which leverages ECC chips to store MACs, and our achieved performance overhead is 2.83%.

5.2 Motivations

To gain a better understanding of the characteristics of encryption counters, we study all the memory-intensive workloads from multiple benchmark sets, including Parboil [Str09], Rodinia [Che09], Polybench [GG09], and Pannotia [Che13].Similar to previous studies [Yua21b; Yua21a; Yua22], we defined memory-intensive workloads as those with at least 20% peak memory bandwidth utilization.

In the split-counter scheme, a major counter is shared by all the blocks (at the cache line

granularity) in a large memory chunk (usually at the page granularity), and a minor counter is maintained for each memory block. We ran the workloads for long periods (over 15 days), allowing them to complete the full simulation in the GPGPU-Sim performance model, and then obtained the traces of encryption counters. Next, we analyze the characteristics of both major and minor counters from the traces.

5.2.1 Major Counter Analysis

We conducted a kernel-level analysis of the number of unique major counter values. In the case where a workload has multiple kernels, we report the maximum number of unique major counter values among the kernels in the entire workload. Figure 5.1 shows the result. From the figure, we can see that most workloads have only a single unique major counter for the entire run. Although some workloads, such as cfd, fdtd2d, sssp, and sssp_ell, have more than one major counter value, the number of unique major counter values is fairly limited. Also, it is worth noting that even when a workload has multiple major counter values, such as cfd, most of the kernels within this workload only have one or two unique major counter values. Therefore, we conclude that only a few major counters would suffice for GPU workloads.

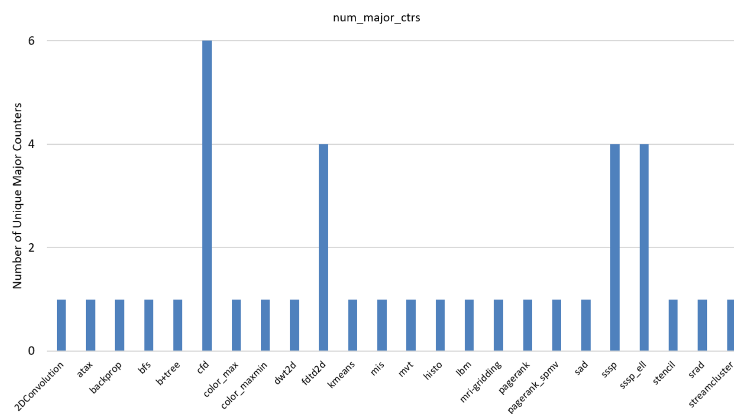

Figure 5.1: Number of Unique Major Counters

5.2.2 Minor Counter Analysis

We observe from the encryption counter traces that minor counters for blocks in a large memory region are similar to each other. To quantify such observation, in Figure 5.2, we show the distribution of delta values for minor counters in the *non-zero counter space*, where the delta is defined as the difference between the maximum and minimum minor counters within a 16KB memory chunk. When the delta value is 0, it means that all minor counters within the 16KB memory chunk are the same. From the figure, we can see that, on average, for 58.1% of minor counters, their deltas range from 0 to 3. For workloads that exhibit a strong streaming data access pattern, such as b+tree, sad and srad, this ratio can reach 100%. Also, we can see that there are many regions with non-zero deltas, for which the previous common counters failed to exploit.

Given our observation that all the minor counters in a large memory region (e.g., 16kB) are similar to each other, we propose the following efficient (but lossy) representation. We propose to use a counter pair, base, delta to represent all the minor counters in the region. The base is the minimum minor counter value and delta is the difference between the maximum and minimum counter values. As the minor counter values are similar, we limit the size of the delta to 2 bits. When the delta is larger than 3, we change the bit flag to indicate that the delta representation is not to be used for this region.

As our delta representation only provide a range of values for minor counters in the region, we need to recover the exact values for decryption. To do so, we resort to the existing stateful MAC verification and leverage the latency-tolerant capabilities of GPUs. In other words, we trade extra MAC verification latency for reduced counter traffic. Recent studies [Yud22; Yua21b] have demonstrated that due to the massive-thread programming model, GPUs can tolerate high latency, making such tradeoffs beneficial.

To reduce the number of trials needed to recover a minor counter from our delta counter representation, we also look into the most common minor counter values in a memory region and the results are shown in Figure 5.3. From the figure, we can see that the most common minor counters have high coverage, 86.63% on average. Note that one may observe inconsistency between Fig.3 and Fig. 4. For example, the benchmark sad shows 37.67% of the delta value of 0, 46.02% of the delta value of 1, and 16.31% of the delta value of 2. In Fig 5.3, 81.02% of the minor counters in a 16kB region share the same value, i.e., the same delta. The reason is that in a region with a certain delta value (e.g., 2), most of the minor counters share the same delta (e.g., 1).

Given the high coverage of most common deltas in 16-kB regions,we extend our delta

counter representation from a pair to a tuple, {base, delta-common, delta-max}. Here, delta-common is the delta for the most common counter value within a memory chunk, and delta-max is the delta for the maximum counter value in that memory chunk. By including the most common counter value within the delta tuple, the secure memory engine always tries the common counter value first, thereby reducing the number of trials to recover the minor counter values.

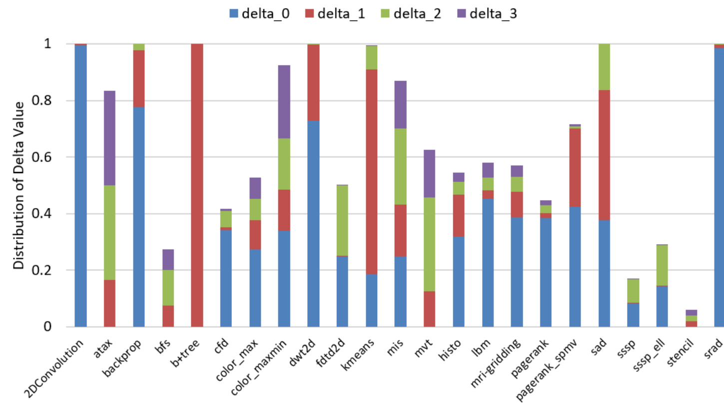

Figure 5.2: Distribution of delta value that less than 4.

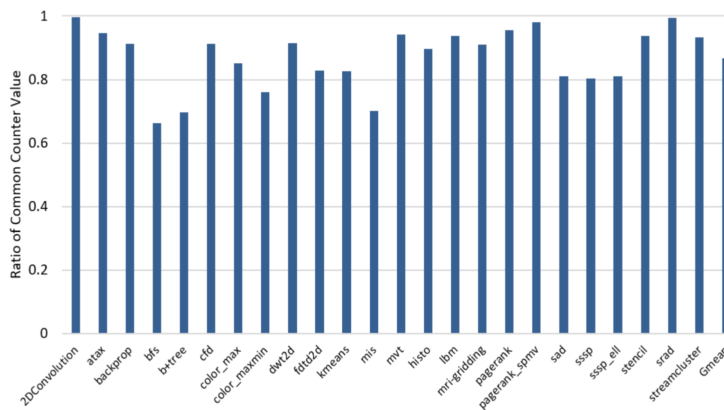

Figure 5.3: Average ratio of the most common counter value in a 16kB region of which the delta among minor counters is less than 4.

5.2.3 MAC Bandwidth Requirement

As discussed above, our proposed delta counter design relies on the stateful MACs to recover the correct minor counter value by trying different values starting from the (base + delta-common). As a result, per-block MACs are required by our design. However, recent works [Yua22; Yua21b; Yua21a; Yud22] identified that MACs contribute the most memory traffic when secure memory is integrated into GPUs. As a result, SHM [Yua22] proposes to use dual-granularity MACs to reduce the MACs bandwidth. In this approach, a large memory chunk can use an 8B MAC when the memory access pattern is streaming, while a per-block 8B MAC is used when the memory access pattern is random. When the coarse-grain MAC is used, the per-block MAC is not stored in off-chip DRAM. Hence, our delta counter design is not directly compatible with it.

As an alternative, we adopt different schemes to reduce the MAC traffic, including VAULT [Taa18b] and Synergy [Sai18]. In VAULT, a regular data block is compressed to make room for the MAC to be embedded into the data block. In Fig. 5.4, we show the layout of data and MAC in the DRAM. If a memory block is compressible, we can embed the 8B MAC into the data cache line(shown in Fig. 5.4 (a)). Otherwise, the per-block MAC is still allocated (shown in Fig. 5.4 (b)). The compressibility of a cache line in our design is defined by the size after compression, i.e., if a 128B cache line can be compressed to less than 120B, we define it as **compressible**. Otherwise, it's **uncompressible**. In our design, we use a commonly used deflate memory compression algorithm – Lempel-Ziv 77 algorithm (LZ77) [Hid19; Pan22], for memory compression. And we assume the compression or decompression latency is two cycles, same as the previous work [Taa18b].

In Synergy, MACs are stored in the ECC chips and can be accessed along with the data block. We evaluate our delta counter design with both VAULT and Synergy in our experiments.

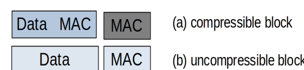

Figure 5.4: Data and MAC layout.

5.3 Architecture Design

5.3.1 Overall Architecture

The overall architecture of our secure memory system for GPUs is shown in Figure 5.5. Similar to previous works, the GPU chip forms the trusted computing base (TCB). Our scheme builds upon the recent work of PSSM [Yua21a], where each memory controller includes a memory encryption engine (MEE) that operates independently to protect a single GDDR memory partition/channel. Each memory controller also contains metadata caches (MDCs) such as the counter cache, MAC cache, and BMT cache to reduce bandwidth consumption for accessing security metadata. The metadata is generated using the partition local addresses to eliminate redundancy across partitions [Yua21a]. Each partition stores a secure root for its corresponding integrity tree and operates independently. In addition, a key generator is integrated into the GPU command processor.

In addition to the conventional security architectural support inherited from CPU Trusted Execution Environments (TEEs), our design introduces several new features. Firstly, we incorporate a few (e.g., four) on-chip registers to hold the major counters and their corresponding memory ranges. Secondly, we implement a small on-chip cache to hold the delta counters, which represent the encryption counters for a large memory chunk.With the embeded MAC design, we include a small on-chip cache to store the compression metadata, where each bit indicates the compressibility of the corresponding memory block. Furthermore, one memory compressor and decompressor are incorporated in each partition to compress and decompress the memory data blocks.

All security metadata is stored in off-chip GDDR memory and can be cached in MDCs. Our design incurs additional memory storage for delta counters and compression metadata if VAULT is enabled. For a 4GB device memory that needs to be protected, 384KB of delta counter storage is required with each 12-bit delta counter protecting a 16KB memory chunk. Additionally, 4MB of compression metadata storage is necessary if each memory block (128B) requires 1 bit for its compressibility.

5.3.2 Management of Major Counter Registers

In our design, the major counter registers are on-chip registers holding the major counter values. As motivated in Section 5.2, GPU workloads feature a small number of unique major counter values. Therefore, a few such on-chip major counters would suffice. In our design, we use four major counter registers, and each has three fields: a 32-bit counter value, the

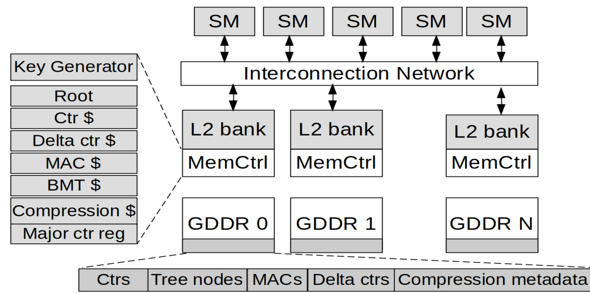

Figure 5.5: Overall architecture.

base address (64-bit), and the size (64-bit) representing the memory chunk that the counter protects. This way, a major counter register can be used for a large memory region whose data blocks share the same major counter value. An example of this major counter entry is shown in Fig 5.6 (a).

Based on the address of a memory access, we check the major counter register to retrieve the major counter value and form a one-time pad with the corresponding minor counter. If the address is not covered by any major counter register, the major counter is read from memory. In our design, similar to previous works, we adopt the split-counter mode with a sectored metadata cache. The layout of one counter sector (32B) is shown in Fig 5.6 (b), with a 32-bit major counter and 32 7-bit minor counters. Each counter sector protects 4KB data memory space. In our design, when a large memory chunk uses the same major counter, the major counter fields will not be transferred over the memory bus to save memory bandwidth because the value of the major counter is already on-chip. As shown in Fig 5.6 (c), each DRAM transaction to fetch/store a counter sector will be in the granularity of 24B instead of 32B. And when a counter sector is allocated in the counter cache, its major counter will be filled by consulting the major counter registers, as shown in Fig 5.6 (d).

Next, let us discuss the management of major counter registers, which is invoked when a minor counter overflows. In a case of a minor counter overflow, the corresponding major counter needs to be incremented. Based on the address of the data block, we check whether a major counter register covers the address. If so, this major counter register is split into two or three, depending the address of data block. If the overflow happens in the middle of the address range, three major counter registers are needed, one covering the range before the block, one covering the 16kB region containing the block, and one for the range afterward. If the block happens to be at the beginning or the end of the range, the major counter is split into two. We show a major counter split example in Fig. 5.7. Assuming the address

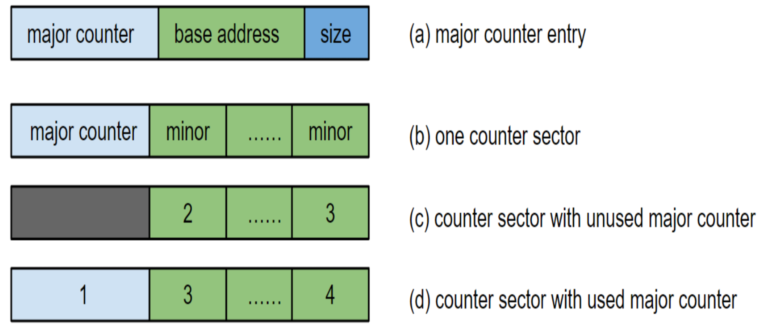

Figure 5.6: Major Counter Entry and Layout of Per-Block Counter.

range of the secure memory managed by a major counter register is 0 64kB, Fig. 5.7 (a), when a minor counter overflows due to a dirty write-back at the block with the address 0x4000, the major counter register is split into three, as Fig. 5.7 (b). During the split process, if there are no sufficient free major counter registers, the major counter needs to be written back to the memory for all the counter blocks within the address range. This happens very rarely for GPU workloads as discussed in Section 5.2.1. During the split process, each newly introduced major counter register is compared with the existing ones to see whether it can be merged with one of them. When two major counter registers cover the address ranges next to each other and share the same major counter value, they can be combined into one. In our example in Fig. 5.7 (b), a subsequent overflow due to a dirty eviction starting at address 0x8000 results in the major counter being updated for the following 32KB region, then the second major counter register is merged with the first one, and the third register is released. The result after the merge is shown in Fig. 5.7 (c).

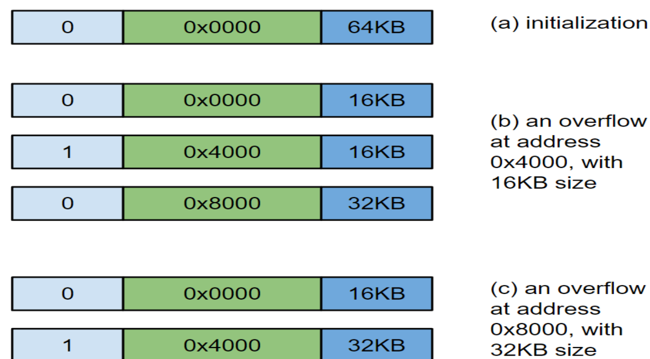

Figure 5.7: An example of major counter register management

Given the counter sector organization, a 32-bit major counter value and 32 7-bit minor counters, each 128B counter block has four sectors, and each sector covers 4kB space (as each minor counter is used for one 128B data block). Here we show how the counter values in the counter cache change in the common scenario of streaming accesses. In Fig 5.8 (a), the initial state of a major counter register has a major counter of value 0, which protects 16KB memory space (corresponding to 4 counter sectors in the sectored counter cache). When the first minor counter overflows due to a memory write, the major counter register splits into two. One has the major counter value increased to 1 and filled into the corresponding counter sector in the counter cache, and the size is changed to 4kB, as shown in Fig. 5.8 (b). The other keeps the counter value as zero and covers the remaining region, which is not shown in the figure. With the streaming data access pattern, the other three counter sectors will be gradually updated, and all the minor counter overflows will be handled with the same procedure, as shown in Fig. 5.8 (c) to (e). Once the entire region is covered by the first major counter register, the second one is deallocated.

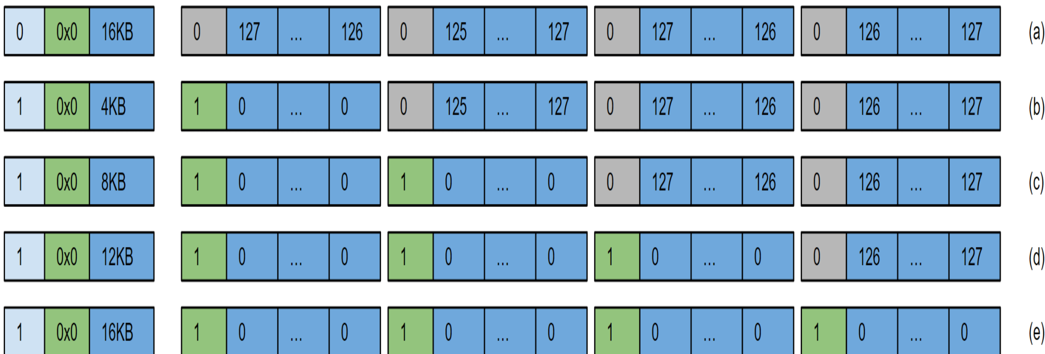

Figure 5.8: Handling of Minor Counter Overflow

Note that our major counter register management is designed for streaming access patterns. When the memory access pattern changes to totally random accesses, the major counter registers will cover only small regions, and the major counters will be updated in memory. When an address is not in the range covered by any major counter registers, the major counters will be accessed from off-chip memory. In such a case, our delta counter design will not have bandwidth benefits but will not affect the system security.

5.3.3 Delta Counter Entry

As described in Section 5.2, we use 12-bits delta counter representation for the encryption counters in a large memory chunk, e.g., 16kB. In our design, each delta entry has four fields as shown in Fig 5.9 (a). Firstly, a 1-bit flag indicates whether the minor counters in this 16KB memory chunk follow the delta counter behavior. If so, this bit is set to 1. Otherwise, this bit is set to 0, and the other three fields can all be set to 0s. Secondly, a 7-bit base counter, which is the same size as the minor counter. Thirdly, a 2-bit delta value can derive the common counter in this memory chunk. Lastly, a 2-bit delta value can derive the maximum counter in this memory chunk.

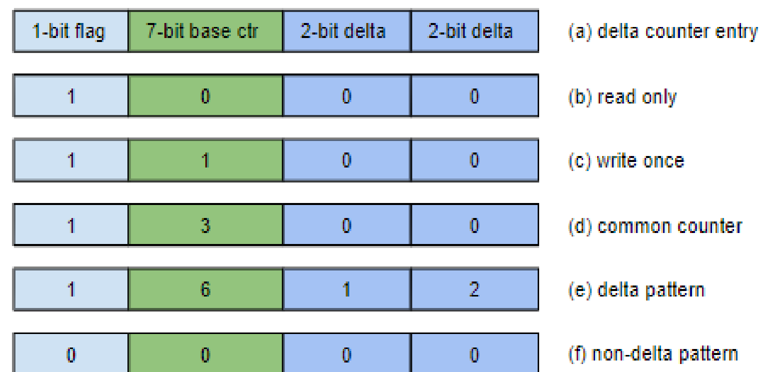

Figure 5.9: Delta counter representation and associated access patterns.

Besides representing all the minor counters in a 16kB chunk, our delta counter can also be used to capture a wide range of memory access patterns as shown in Fig 5.9 (b) to (f). First, it can capture the read-only (Fig 5.9 (b)) as the base and deltas are all zero. Second, it can detect the write-once (Fig 5.9 (c)) data with only the base value being 1. Third, our delta counter can capture the common counter pattern with deltas being 0, as illustrated in Fig 5.9 (d). Fourth, the delta counters can be used to capture the generic pattern where the deltas among the minor counters in a memory region are small (less than 4), as shown in Fig 5.9 (e). When a delta of 4 (or higher) is detected, the flag bit is reset to indicate that the minor counters in the memory chunk do not share similar values and cannot take advantage of our delta counter representation.

In summary, our proposed delta counter representation can detect all the write-back patterns proposed in previous approaches, including read-only data in SHM [Yua22] and Common counters in [Na21], making it a promising solution for memory encryption

counters on GPUs.

5.3.4 Dataflow of DRAM Read and Write

When used with embeded MAC proposed in VALUT [Taa18a], the memory compression and decompression process becomes an integral part of the memory access path. For a memory write operation, the plaintext is first compressed before encryption using the AES engine. The encrypted ciphertext is then passed to the MAC engine to compute per-block MACs. On the other hand, during a memory read operation, the per-block counter needs to be recovered through the stateful MAC by trying different possible counter values from the delta counter representation, one at a time starting from (base + delta-common). Once the correct counter value is recovered, the decrypted data is decompressed and returned to the L2 cache.

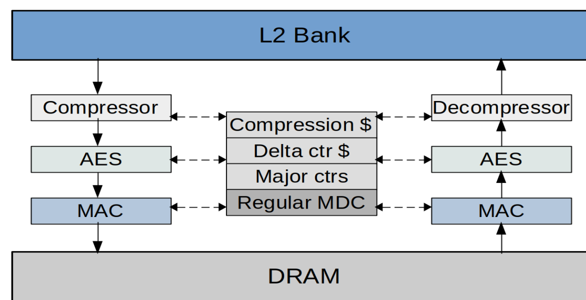

Figure 5.10: DRAM flow for memory read and write.

5.3.5 Delta Counter Cache Management

Separate from the counter cache, our delta counter cache keeps the most recently used delta counter representations. Initially, all the delta counters have their flag bit set to 1 and the other three fields set to 0, indicating the entire memory space is read-only. As the program executes, minor counters are updated, and the corresponding delta counter entry is updated accordingly and inserted into the delta counter cache if not cached yet. The modified delta counter block is marked as dirty and must be written back to memory. To illustrate this process, we present an example of a delta counter in Fig 5.11. Initially, the base counter and deltas in the entry are set to 0 and marked as clean (Fig 5.11 (a)). As program

execution continues, the delta to the maximum minor counter is updated to 1, and the block is marked as dirty (Fig 5.11 (b)). Subsequently, as most memory blocks within the 16KB chunk are written at least once, the delta of the common counter is also updated to 1 (Fig 5.11 (c)). Later on, some blocks may be evicted and written again, causing the delta to the maximum minor counter to increase to 2 (Fig 5.11 (d)). When the memory access pattern changes to a completely random pattern, the encryption counters may no longer follow the delta pattern, and the corresponding delta entry is updated as shown in Fig 5.11 (e).

1 | 0 | 0 | 0 (a) clean
1 | 0 | 0 | 1 (b) dirty
1 | 0 | 1 | 1 (c) dirty
1 | 1 | 1 | 2 (d) dirty
0 | 0 | 0 | 0 (e) dirty

Figure 5.11: Delta Counter Management

5.3.6 Counter Cache and BMT Verification

In our delta counter design, the major counters are provided from the major counter registers, and the minor counters are compactly represented with only 12 bits for all the minor counters in a 16KB memory chunk. After recovering the correct version of a minor counter with stateful MACs, it is placed in the counter cache. As a counter block/sector has multiple minor counters, the un-accessed minor counters are set to -1, as shown in Fig 5.12 (a). In other words, we reserve -1 and a minor counter overflows when it reaches all 1s. As more data is accessed within the same memory chunk, the corresponding minor counters are recovered from the delta counter entry, as shown in Fig 5.12 (b). Once all minor counters are recovered in this counter block, the BMT is traversed to verify the integrity of the counter block. If this fully recovered block is further updated when it resides in the counter cache, it will be marked as dirty and is written back to memory when evicted.

In the case when not all blocks are accessed within a 16KB memory chunk, some of the corresponding minor counter will be -1, indicating a 'partial' recovery of the counter block, as shown in Fig 5.12 (c). To ensure integrity verification, the memory encryption engine must fetch this block from memory and merge it with the partially recovered counter block.

Similar to SHM, we set a time-out period of 6K cycles. If after 6000 cycles any minor counter has not been recovered from the delta entry, we need to fetch the minor counters in the counter block. Additionally, if a partially recovered counter is about to be replaced from the counter cache, the memory encryption engine must also fetch the minor counters before eviction.

In our delta counter design, the minor counters are always written to the DRAM when they are in the dirty state in the counter cache. This ensures the correctness and consistency of encryption counters. We use the delta counter entry to recover the per-block minor counter during memory reads. Since GPU applications are typically not write-intensive, our design reduces memory read bandwidth by avoiding the need to access per-block minor counters, which improves the performance of secured GPU memory. Overall, our delta counter design is efficient and robust in managing encryption counters and maintaining memory access integrity.

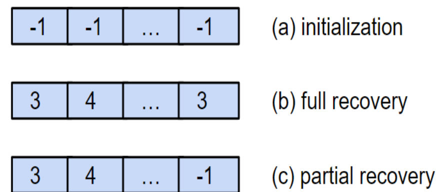

Figure 5.12: Counter Cache Management

5.3.7 Data Compressibility

Our design relies on stateful MACs for recovering minor counters from our super-compact delta counter representation. To reduce the memory bandwidth consumption caused by accessing the MACs, we utilize a technique similar to the one used in VAULT [Taa18a]. Specifically, we compress the regular data block and embed the per-block MAC into the same cache line. Therefore, during each memory access, the memory encryption engine also retrieves the corresponding compressibility information of the data block to determine where to obtain the MAC for verifying the data. If the MAC is embedded within the data block, the data block needs to be decompressed without additional MAC access. Otherwise, a MAC block needs to be accessed beside the data block, although the data block does not need to be decompressed.

In the VAULT design, one bit is reserved in the per-block minor encryption counter to hold the compressibility information. Although this design reduces the minor counter size from 7 bits to 6 bits, it is a good trade-off for reducing the memory bandwidth to access MACs. However, in our delta counter design, the per-block minor counters are not fetched if the minor counters are represented with a delta pattern. Hence, our delta counter design requires the compressibility information to be maintained separately.

In our design, we maintain a 1-bit flag for each 128B cache block to store the compressibility information, and cache it in a small on-chip compression cache. The management of the compression cache is illustrated in Fig 5.13. Given four consecutive memory blocks, assume that they are all compressible in the memory, a write stream to these four memory blocks updates the memory content. If all the memory blocks are still compressible, none of the flag bits is flipped. Therefore, it will remain clean in the compression cache, as shown in Fig 5.13 (a). Otherwise, when any bit is flipped to 0 (or from 0 to 1), the block will be marked as dirty (shown in Fig 5.13 (b)), and needs to be written to the main memory.

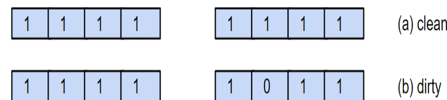

Figure 5.13: Compression Cache Management

In our design, although the compressibility information is maintained separately and stored in memory, it's integrity does not need to be maintained. In other words, we do not need to use MAC to protect these flags. The reason is that, even an attacker could flip one compressibility flag bit. For example, if a compressibility flag bit is flipped from 1 to 0, the memory controller would issue a memory read to fetch the per-block MAC, and get the stale MAC. This will eventually trigger a failure in the verification process. Similarly, if a compressibility flag bit is flipped from 0 to 1, some data bits would be used as MACs, also leading to a verification failure.

Another way to retrieve the MAC information is to leverage ECC bits if the memory is equipped with ECC protection, as proposed in SYNERGY[Sai18]. By storing MACs in the ECC bits, the MAC accesses become free and do not require additional flag bits. We evaluate our proposed delta counter scheme with both the SYNERGY and embedded MAC schemes.

Table 5.1: Baseline GPU Configuration

SM config	30 SMs, 1506MHz
Register File	256KB/SM, 7.5MB in total
L1 D-Cache / Shared Memory	96KB/SM
L2 cache	2 banks per memory partition, each L2 cache bank is 128KB, 3MB in total.
DRAM	3500MHz, 12 partitions, 336GB/s.

5.4 Methodology

We model delta counter designs with GPGPU-Sim v4.0 [Kha20]. Our baseline GPU configuration is shown in Table 5.1, which is based on the Nvidia Turing architecture [Nvi21]. A memory space of 4GB is protected by the memory encryption engine.

We build our scheme based on the state-of-the-art design, PSSM [Yua21a], in which the security metadata is constructed with partition-local offset. The details of the simulated secure memory engine are listed in Table 5.2. The separate metadata caches, which include the counter cache, MAC cache, BMT cache, delta counter cache and compression cache, are modeled in each memory partition (i.e., each memory controller). State-of-the-art secure memory architecture in CPUs uses speculative verification and lazy update for BMT [Gas03] verification. We also adopt these schemes on GPUs. Speculative verification means the memory controller can supply the data to the core before the corresponding integrity check is finished. Later on, if there is a failure in integrity verification, an exception would be raised. Lazy update means that only when a counter block or a tree node is evicted from the counter cache or BMT cache, its parent node will be updated in the BMT cache.

We conducted a comprehensive evaluation of our proposed secure memory designs using a wide range of GPGPU benchmark suites, including Rodinia-3.1 [Che09], Parboil [Str09], Polybench [GG09], and Pannotia [Che13], which cover both regular and irregular workloads. Since computation-intensive workloads are not sensitive to secure memory, we chose to focus on all memory-intensive workloads and report the benchmark details in Table 5.3. For benchmarks with low simulation time, we simulated the entire benchmarks, while for benchmarks with long simulation time, we simulated the first 10 million cycles. Following previous work [Yua21b; Yua21a; Yua22], we defined the workloads that have at least 20% peak memory bandwidth utilization as memory-intensive.

The different secure memory designs that we evaluate in our experiments are listed in

Table 5.2: MDC and MEE Organization

Delta Counter cache	2KB / memory partition, 128B block, fully-associated, write-allocate policy
Counter cache	2KB / memory partition, 128B block, 4-way sectored, write-allocate policy
Mac cache	2KB / memory partition, 128B block, 4-way sectored, write-allocate policy.
Compression cache	2KB / memory partition, 128B block, fully-associated, write-allocate policy.
Bonsai Merkle Tree cache	2KB / memory partition, 128B block, 4-way sectored, write-allocate policy.
Unified Metadata cache	10KB / memory partition, 128B block, 5-way sectored, write-allocate policy.
Hash/Mac latency	40 cycles
AES engines	1 pipelined AES/memory partition
Compression Latency	3 cycles [Taa18b]

Table 5.3: Benchmarks

Benchmark Suite	**Workloads**
Rodinia	backprop, bfs, b+tree, dwtd2d, cfd, kmeans, srad_v2, streamcluster
Parboil	histo, sad, stencil, lbm, mri-gridding
Polybench	2DConvolution, atax, mvt, fdtd2d
Pannotia	color_max, color_maxmin, mis, pagerank, pagerank_spmv, sssp, sssp_ell

Table 5.4. We report normalized Instructions per cycle (IPC) in our evaluation, with the same baseline GPUs architecture.

5.5 Evaluation

5.5.1 Overall Performance

We first evaluate the performance of our delta counter designs and present the results in Figure 5.14. We compare our designs with previous works, including PSSM [Yua21a] and SHM [Yua22]. As shown in the figure, our delta counter design, labeled as ’Delta_ctr’, achieves low overheads for secure GPU memory design, with only 3.75% on average. This outperforms the state-of-the-art SHM and PSSM designs, which have average overheads of 6.74% and 13.89%, respectively. However, for some workloads such as lbm and kmeans, even the state-of-the-art SHM design still exhibits high overheads. This is due to the workloads being extremely memory-intensive, with almost 100% bandwidth utilization. As a result, the performance of these workloads is tightly bounded by additional memory bandwidth due to security metadata. And for workload lbm, our delta counter design still has an overhead of 27.4%, and the reason is that our delta counter design introduces two additional metadata, namely, delta counter and compression metadata. Our simulation shows that these additional metadata suffer high cache miss rates, more than 30%, resulting in further memory bandwidth consumption and slowdowns in the performance. To address this issue, we propose using a unified metadata cache to cache the 5 types of security metadata. Our observation is that GPU applications usually have different data patterns. Specifically, when an application has a large portion of read-only data, the counter and BMT caches are under-utilized. And when an application has a large portion of compressible data, the MAC cache is under-utilized. By unifying all these 5 types of metadata caches into one cache with the same overall capacity, the on-chip cache resource can be better utilized. The results, labeled ’Delta_unified’, show that our delta counter design, combined with a unified metadata cache, can further reduce performance overheads to only 2.01%. Some workloads, e.g., lbm, can get more than 10% performance improvement.

5.5.2 Performance breakdown

To better understand our results, we present a detailed performance breakdown and compare the achieved performance with upper bounds. The label "Compression_only" indicates

Table 5.4: Evaluated designs for GPU secure memory with both memory encryption and integrity verification.

Scheme	What It Represents
PSSM	Secure GPU memory with PSSM scheme [Yua21a]
PSSM_synergy	Secure GPU memory with the PSSM scheme, combined with the SYNERGY [Sai18] design to address MAC bandwidth.
SHM	Secure heterogeneous memory design [Yua22], with the PSSM scheme to construct security metadata.
Delta_ctr	Our delta counter design for secure GPU memory, with the memory blocks being compressed to embed the MACs.
Delta_only	Our delta counter design for secure GPU memory, with all the metadata caches working in the perfect mode (meaning that there is no additional bandwidth caused by security metadata). However, the additional encryption latency due to trying different encryption counters is modeled.
Delta_unified	Our delta counter design for secure GPU memory, with a 10KB unified metadata cache that can hold all the different types of metadata.
Compression_only	Our delta counter design for secure GPU memory, with the memory blocks being compressed to embed the MACs. And the per-block counters and BMTs are enabled. However, when a memory block is uncomressible, the per-block MAC bandwidth is still paid.
Delta_synergy	Our delta counter design for secure GPU memory, combined with SYNERGY [Sai18] design to address MAC bandwidth.
Upper_bound	The upper bound analysis of delta counter design, in which all the metadata caches work in the perfect mode and do not produce any memory traffic. However, the per-block MAC bandwidth is still modeled if the memory block is uncompressible.

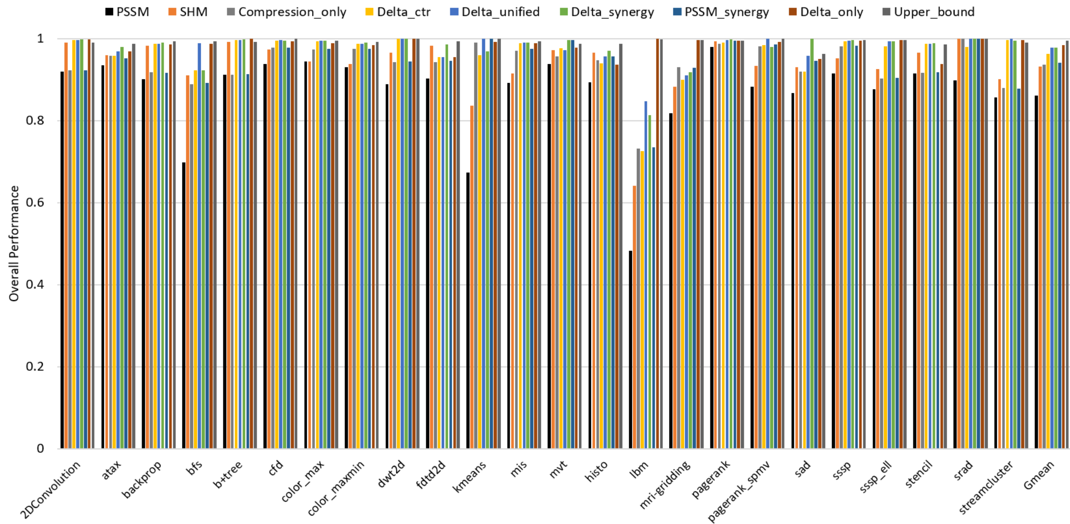

Figure 5.14: Overall Performance of Different Designs

that the per-block counter and BMT are maintained, while memory blocks are compressed to embed the MAC into the data lines, if compressible. The per-block MAC is still maintained if the cache line is not compressible. Compare "Compression_only" design with our delta counter design (labeled as "Delta_ctr"), we can clearly see the performance improvement. This improvement is directly coming from the counter traffic reductions. We present the ratio of counter traffic in Fig 5.15. With the "Compression_only" design, the counter traffic only includes per-block counters. With our delta counter design, the counter traffic includes both delta counters and per-block counters. As we can clearly see from the figure, the counter traffic drops from 2.12% to 0.18% on average. Some workloads, e.g., atax and mvt, can get more than 10% counter traffic reduction. To further confirm our conclusion, we also evaluate the approach used in another previous work, SYNERGY [Sai18], in which the ECC chip is repurposed to address the MAC traffic. We combine this approach with both PSSM design and our delta counter design, and we can clearly see that our delta counter design outperforms the PSSM design.

We report the results of memory compressibility in Figure 5.16. We maintain one counter for compressible memory accesses and one counter for uncompressible memory accesses. If a memory access can be compressed to embed the 8B MAC into the line, we consider it compressible and increment the counter by one; otherwise, we consider it uncompressible and increase the other counter. At the end of the simulation, we calculate the ratio of compressible memory access and plot the results in the figure. Most workloads have a good

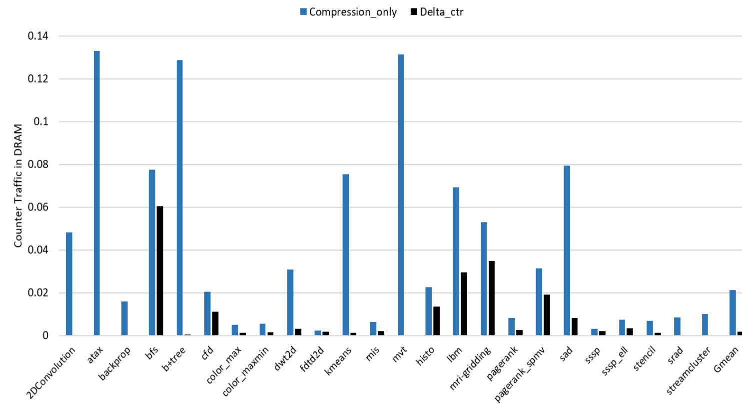

Figure 5.15: Ratio of Counter Traffic in Memory.

ratio of compressible memory accesses. On average, the compressible memory access ratio can reach 77.54%, resulting in a significant reduction in memory bandwidth for accessing MACs, when embedded in data blocks, which ultimately translates to improvements in performance.

Our delta counter tuple includes the delta values for both the common minor counter and the maximum counter, as discussed in Section 5.2. The reason is that the encryption engine needs to try multiple counter values until the correct one is recovered, staring from the most common one. We plot the distribution of the number of recovery trials that each memory access experiences in Fig. 5.17, and we can see that most memory accesses will recover the counter value with just one try, labeled as "try_1."

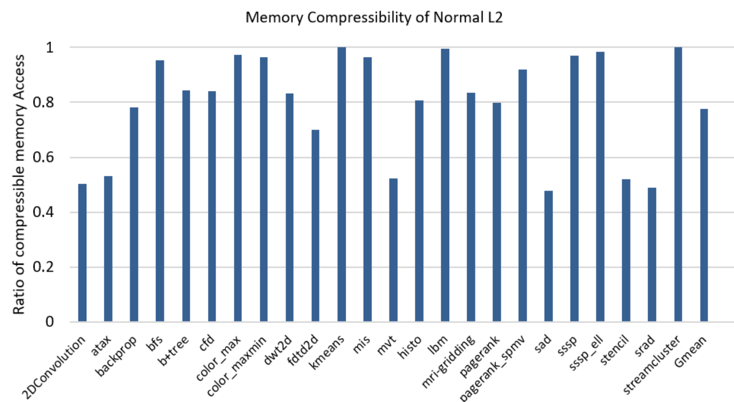

Figure 5.16: Ratio of Compressible Memory Access.

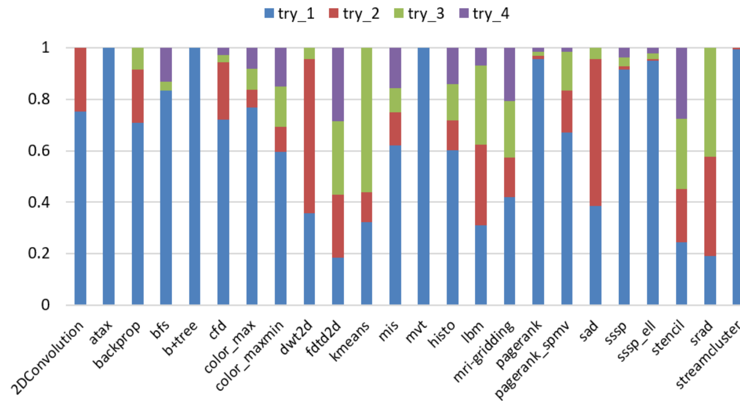

Figure 5.17: Distribution of Number of Encryptions that Each Memory Access Needs to Try.

5.5.3 Upper Bound Analysis

We also investigate the upper bound of the performance. We evaluate two ideal designs, "Delta_only" and "Upper_bound". The label "Delta_only" implies that all the metadata cache works in perfect mode, thereby eliminating any additional memory bandwidth overheads due to accessing security metadata. However, the latency incurred due to trying different delta values is modeled. As we can see that the overhead in this ideal design is 1.5%, due to GPUs' latency tolerance of trying different counter values. The label "Upper_bound", is the design that assumes all the metadata cache except MAC cache work in perfect mode, i.e., not incurring any additional memory traffic. However, the uncompressible memory blocks still need to fetch/store per-block MACs under a cold miss. This design incurs only 0.49% overheads on average, and our delta counter combined with unified metadata cache design can achieve 2.01% overheads on average, which is quite close to the upper bound.

5.6 Conclusions

The security metadata traffic is a major performance bottleneck for GPU secure memory. This paper focuses on encryption counters due to their impact on the counter and BMT traffic while utilizing prior schemes to address the MAC traffic. We introduce a super compact method for encryption counter representation and a novel verification scheme that leverages the latency-tolerance capability of GPUs to determine the correct minor

counter values. Our evaluation demonstrates that our delta counter scheme significantly reduces the bandwidth overheads of encryption counters and achieves secure GPU memory with an average performance overhead of 2.01% compared to the same GPU without security support.

CHAPTER

6

CONCLUSIONS AND FUTURE WORKS

This dissertation presents our study on supporting secure memory architecture for GPUs. We start our investigation from a detailed performance analysis, and identified the major performance bottleneck. In the second work, we proposed a simple yet effective approach for secured GPU memory. In the third work, we explored the heterogeneous memory space on GPUs and streaming data access pattern for providing secure memory on GPUs. In the fourth work, we propose a bandwidth-efficient encryption counter representation for secure GPUs.

In the first work, we explore two secure memory architectures, counter-mode encryption and direct encryption, for GPUs, and show that we need to architect secure memory differently from it for CPUs. Our in-depth study reveals the following insights. First, as GPUs are designed for high-throughput computation, its secure memory needs to deliver high bandwidth. Second, with counter-mode encryption, the memory traffic resulting from the metadata, i.e., the counters, MACs (message-authentication codes), and integrity tree, may cause significant performance degradation, even in the presence of metadata caches. Third, the sectored cache structure adopted by GPUs leads to multiple sequential accesses to the same metadata cache line, which necessitates the use of MSHRs (miss-status handling registers) for meta-data caches. Fourth, unlike CPUs, separate/partitioned metadata caches

perform better than unified metadata caches on GPUs. The reason is that GPU workloads feature streaming accesses, which cause severe contention in the unified metadata cache and the cached counters and integrity tree nodes may be evicted before being reused. Fifth, the massive-threaded nature of GPUs make them latency-tolerant and the performance impact due to the extra encryption/decryption latency is limited. As a result, direct encryption can be a promising alternative for GPU secure memory. The challenge, however, lies in memory integrity verification as the integrity tree may incur high storage overhead and metadata traffic.

In the second work, we point out that conventional CPU secure memory architecture can not be directly adopted to the GPUs. The key reasons include: (1) accessing the security metadata, including encryption counters, message authentication codes (MACs) and integrity trees, requires significant memory bandwidth, which may lead to severe bandwidth competition with normal data accesses and degrade the GPU performance; (2) contemporary GPUs use partitioned memory organization, which results in storage and coherence problems for encryption counters and integrity trees since different partitions may need to update the same counter/integrity tree blocks; and (3) the existing split-counter block organization is not friendly to sectored caches, which are commonly used in GPU for bandwidth savings. Based on these observations, we propose partitioned and sectored security metadata (PSSM), which has two components: (a) using the offset addresses (referred to as local addresses) within each partition, instead of the virtual or physical addresses, to generate the metadata so as to solve the counter or integrity tree storage and coherence problems and (b) reorganizing the security metadata to make them friendly to the sectored cache structure so as to reduce the memory bandwidth consumption of metadata accesses. With these proposed schemes, the performance overhead of secure GPU memory is reduced from 59.22% to 16.84% on average. If only memory encryption is required, the performance overhead is reduced from 29.53% to 5.18%.

In the third work [Yua22], we analyze the security guarantees that used to defend against physical attacks, and make the observation that heterogeneous GPU memory system may not always need all the security mechanisms to achieve the security guarantees. Based on the memory types as well as memory access patterns either explicitly specified in the GPU programming model or implicitly detected at run time, we propose adaptive security memory support for heterogeneous memory on GPUs. Specifically, we first identify the read-only data and propose to only use MAC (Message Authentication Code) to protect their integrity. By eliminating the freshness checks on read-only data, we can use a single counter for such data regions and remove the corresponding part in the Bonsai Merkel Tree (BMT), thereby

reducing the traffic due to counters and BMT. Second, we detect the common streaming data access patterns and propose coarse-grain MACs for such stream data to reduce the MAC access bandwidth. With the hardware-based detection of memory type (read-only or not) and memory access patterns (streaming or not), our proposed approach adapts the security support to significantly reduce the performance overhead. Our evaluation using both memory-intensive and computation-intensive workloads shows that our scheme can achieve secure memory on GPUs with low overheads for memory-intensive workloads while not affecting computation-intensive workloads. Among the fourteen memory-intensive workloads in our evaluation, our design reduces the performance overheads of secure GPU memory from 53.9% to 10.2% on average. Compared to the state-of-the-art secure memory designs for GPUs [Na21; Yua21a], our scheme outperforms PSSM by up to 36.8% and 10.4% on average and outperforms Common counters by 77.5% on average for memory-intensive workloads.

In the fourth work, we focus on encryption counters given their impact on the counter and BMT traffic while leveraging prior schemes [Sai18; Taa18a] to address the MAC traffic. We first analyze the characteristics of the encryption counters from a wide range of GPGPU benchmarks and make two key observations. (1) With the split counter scheme, the cache blocks in a large portion of the memory space, sometimes the entire GPU memory space, share the same major counter value. (2) The difference among minor counters is fairly limited. We then propose a novel scheme to reduce the encryption counter traffic. Our design includes (a) a highly compact way of counter representation and (b) a verification scheme to determine the correct minor counter values. In our design, we use a few on-chip registers to hold the major counters and use a (7-bit) base value along with two small (2-bit) deltas to represent the minor counters in a large memory chunk, one delta for the most frequent delta between minor counters and the base, the other delta for the maximal difference between a minor counter and the base. This way, for a large memory chunk (e.g., 16kB), the counter overhead becomes nearly negligible (less than 2B). We then leverage the existing MAC verification logic to verify the minor counters computed from the base and deltas. Our approach essentially trades off decryption and integrity-check latency for reduced counter-data traffic to take advantage of the latency-hiding nature of GPUs. Compared to prior works on reducing counter traffic [Na21], our scheme handles more counter value patterns (as we don't restrict the counters to be the same in a memory chunk) and is more effective in reducing counter traffic. Our study also reveals that the GPU memory data are typically compressible. As a result, we can co-locate the MACs with the compressed cache blocks, similar to [Taa18a]. Our experimental results show that our

proposed delta counter scheme significantly reduces the storage and bandwidth overheads of encryption counters and achieves secure GPU memory with an average performance overhead of 2.01% compared to GPU without security support. Our delta scheme is also compatible with SYNERGY [Sai18], which leverages ECC chips to store MACs, and our achieved performance overhead is 2.83%.

There are still several challenges that need to be addressed in our future work. Firstly, we need to focus on designing a secure GPU memory system for the unified memory space on GPUs. While existing research primarily concentrates on providing memory encryption and integrity protection for discrete GPU memory, it is essential to develop memory protection mechanisms that are compatible with unified memory, which is a common feature in GPU programming. Secondly, we should tackle the issue of secure memory design for multi-GPU and GPU clusters. Workloads that utilize GPUs, such as scientific computations, machine learning, and neural networks, often require significant computational resources. As a result, multi-GPU and GPU clusters are commonly deployed. Although various solutions exist for securing memory on a single-node GPU, there is currently no research addressing memory security in the context of multi-GPU and GPU clusters. Lastly, while our current work primarily focuses on GPUs, it is worth noting that neural processing units (NPUs) are also widely used for deep learning workloads. In our future research, we may also consider incorporating the security aspects of memory design for NPUs. By addressing these challenges, we aim to enhance the security and efficiency of GPU and NPU memory systems, thus advancing the field of high-performance computing and deep learning.

BIBLIOGRAPHY

[Aji11] A. M. Aji, M. Daga, and W. Feng. “Bounding the effect of partition camping in GPU kernels”. In: *Proceedings of the 8th Conference on Computing Frontiers, 2011, Ischia, Italy, May 3-5, 2011*. Ed. by C. Cascaval, P. Trancoso, and V. K. Prasanna. Italy: ACM, 2011, p. 27.

[Che09] S. Che et al. “Rodinia: A Benchmark Suite for Heterogeneous Computing”. In: *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. 2009.

[Che13] S. Che et al. “Pannotia: Understanding irregular GPGPU graph applications”. In: *Proceedings of the IEEE International Symposium on Workload Characterization, IISWC 2013, Portland, OR, USA, September 22-24, 2013*. IEEE Computer Society, 2013, pp. 185–195.

[Cor19] I. Corporation. “Intel® 64 and IA-32 Architectures Software Developer’s Manual (325462-071US)”. In: (2019).

[Gas03] B. Gassend et al. “Caches and Hash Trees for Efficient Memory Integrity Verification”. In: *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA’03), Anaheim, California, USA, February 8-12, 2003*. IEEE Computer Society, 2003, pp. 295–306.

[GG09] cott Grauer-Gray et al. “Auto-tuning a High-Level Language Targeted to GPU Codes”. In: *To Appear In Proceedings of Innovative Parallel Computing*. 2009.

[GOO19] D. GOODIN. *RAMBleed side-channel attack works even when DRAM is protected by error-correcting code*. 2019. URL: `https://arstechnica.com/information-technology/2019/06/researchers-use-rowhammer-bitflips-to-steal-2048-bit-crypto-key/`.

[Gue16a] S. Gueron. “A Memory Encryption Engine Suitable for General Purpose Processors”. In: *IACR Cryptol. ePrint Arch.* 2016 (2016), p. 204.

[Gue16b] S. Gueron. “Memory Encryption for General-Purpose Processors”. In: *IEEE Secur. Priv.* 14.6 (2016), pp. 54–62.

[Gup15] G. Gupta. “What is Birthday attack??” In: (2015).

[HAR87] D. T. HARPER and J. R. JUMP. “Vector Access Performance in Parallel Memories Using a Skewed Storage Scheme”. In: *IEEE Trans. Computers* C-36.5 (1987), pp. 1440 –1449.

[Hid19] A. Hidayat. *FastLZ*. 2019. URL: `https://github.com/ariya/FastLZ`.

[Hua20a] W. Hua et al. “GuardNN: Secure DNN Accelerator for Privacy-Preserving Deep Learning”. In: *CoRR* abs/2008.11632 (2020). arXiv: `arXiv:2008.11632`.

[Hua20b] W. Hua et al. “MgX: Near-Zero Overhead Memory Protection with an Application to Secure DNN Acceleration”. In: *CoRR* abs/2004.09679 (2020). arXiv: `arXiv:2004.09679`.

[Jan18] I. Jang et al. “Heterogeneous Isolated Execution for Commodity GPUs”. In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI’18)*. 2018.

[Jia16] Z. H. Jiang, Y. Fei, and D. Kaeli. “A complete key recovery timing attack on a GPU”. In: *Symposium on High Performance Computer Architecture (HPCA)*. 2016.

[Jia18] Z. Jia et al. “Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking”. In: *CoRR* abs/1804.06826 (2018). arXiv: `1804.06826`.

[Kap16] D. Kaplan, J. Powell, and T. Woller. “AMD Memory Encryption”. In: (2016).

[Kha20] M. Khairy et al. “Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling”. In: *proceedings of the 47th IEEE/ACM International Symposium on Computer Architecture(ISCA)*. 2020.

[Kum20] R. Kumar et al. “A 4900—μm 2 839-Mb/s Side-Channel Attack-Resistant AES-128 in 14-nm CMOS With Heterogeneous Sboxes, Linear Masked MixColumns, and Dual-Rail Key Addition”. In: *IEEE Journal of Solid-State Circuits (Volume: 55, Issue: 4, April 2020)* (2020).

[Law82] D. H. Lawrie and C. R. Vora. “The Prime Memory System for Array Access”. In: *IEEE Trans. Computers* 31.5 (1982), pp. 435–442.

[Lee05] R. B. Lee et al. “Architecture for protecting critical secrets in microprocessors”. In: *32nd International Symposium on Computer Architecture (ISCA’05)*. 2005.

[Leh18] T. S. Lehman, A. D. Hilton, and B. C. Lee. "MAPS: Understanding Metadata Access Patterns in Secure Memory". In: *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2018, Belfast, United Kingdom, April 2-4, 2018*. IEEE Computer Society, 2018, pp. 33–43.

[Len13] J. Leng et al. "GPUWattch: enabling energy optimizations in GPGPUs". In: *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*. Ed. by A. Mendelson. ACM, 2013, pp. 487–498.

[Lie00] D. Lie et al. "Architectural Support for Copy and Tamper Resistant Software". In: *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000*. Ed. by L. Rudolph and A. Gupta. ACM Press, 2000, pp. 168–177.

[Lin18] Z. Lin, M. Mantor, and H. Zhou. "GPU Performance vs. Thread-Level Parallelism: Scalability Analysis and a Novel Way to Improve TLP". In: *ACM Trans. Archit. Code Optim.* 15.1 (2018), 15:1–15:21.

[Liu17] Y. Liu, Y. Xie, and A. Srivastava. "Neural Torjans". In: *2017 IEEE International Conference on Computer Design (ICCD)*. 2017.

[Mat11] S. K. Mathew et al. "53 Gbps Native GF(2 4) 2 Composite-Field AES-Encrypt/Decrypt Accelerator for Content-Protection in 45 nm High-Performance Microprocessors". In: *IEEE Journal of Solid-State Circuits*. 2011.

[Na21] S. Na et al. "Common Counters: Compressed Encryption Counters for Secure GPU Memory". In: *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 2021, pp. 1–13.

[NM09] N. P. J. Naveen Muralimanohar Rajeev Balasubramonian. "CACTI 6.0: A Tool to Model Large Caches". In: *4HP Laboratories* (2009).

[Nvi21] Nvidia. *NVIDIA TURING GPU ARCHITECTURE*. Tech. rep. USA, 2021.

[Pan22] G. Panwar et al. "Translation-optimized Memory Compression for Capacity". In: *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*. IEEE, 2022, pp. 992–1011.

[Pin19] S. Pinto and N. Santos. "Demystifying Arm TrustZone: A Comprehensive Survey". In: *ACM Comput. Surv.* 51.6 (2019), 130:1–130:36.

[Qur06] M. K. Qureshi and Y. N. Patt. “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches”. In: *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006), 9-13 December 2006, Orlando, Florida, USA*. IEEE Computer Society, 2006, pp. 423–432.

[Rau79] B. R. Rau. “Interleaved Memory Bandwidth in a Model of a Muyltiprocessor Computer System”. In: *IEEE Trans. Computers* 28.9 (1979), pp. 678–681.

[Rog06] B. Rogers et al. “Effective Data Protection for Distributed Shared Memory Multiprocessors”. In: *in Proceedings of the International Conference of Parallel Architecture and Compilation Techniques (PACT)*. 2006.

[Rog07] B. Rogers et al. “Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly”. In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2007).

[Rog08a] B. Rogers et al. “Single-level integrity and confidentiality protection for distributed shared memory multiprocessors”. In: *HPCA*. 2008.

[Rog08b] B. Rogers et al. “Single-Level Integrity and Confidentiality Protection for Distributed Shared Memory Multiprocessors”. In: *in Proceedings of the 14th International Symposium on High Performance Computer Architecture (HPCA-14*. 2008.

[Sai18] G. Saileshwar et al. “SYNERGY: Rethinking Secure-Memory Design for Error-Correcting Memories”. In: *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. Austria: IEEE Computer Society, 2018, pp. 454–465.

[Sin19] A. Singh et al. “Improved Power/EM Side-Channel Attack Resistance of 128-Bit AES Engines With Random Fast Voltage Dithering”. In: *IEEE Journal of Solid-State Circuits* (2019).

[Str09] J. A. Stratton et al. “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing”. In: *IMPACT Technical Report*. 2009.

[Suh03] G. Suh et al. “Efficient Memory Integrity Verification and Encryption for Secure Processors”. In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36*. 2003.

[Suh07] G. E. Suh, C. W. O'Donnell, and S. Devadas. "AEGIS: A Single-Chip Secure Processor". In: *IEEE Design Test of Computers* (2007).

[Taa18a] M. Taassori, A. Shafiee, and R. Balasubramonian. "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*. Ed. by X. Shen et al. ACM, 2018, pp. 665–678.

[Taa18b] M. Taassori, A. Shafiee, and R. Balasubramonian. "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*. USA: ACM, 2018, pp. 665–678.

[Vol18] S. Volos, K. Vaswani, and R. Bruno. "Graviton: Trusted Execution Environments on GPUs". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 2018.

[Yan03] J. Yang, Y. Zhang, and L. Gao. "Fast Secure Processor for Inhibiting Software Piracy and Tampering". In: *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. 2003.

[Yan06] C. Yan et al. "Improving Cost, Performance, and Security of Memory Encryption and Authentication". In: *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*. 2006.

[Yan10] Y. Yang et al. "A GPGPU compiler for memory optimization and parallelism management". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. Ed. by B. G. Zorn and A. Aiken. Canada: ACM, 2010, pp. 86–97.

[Yua21a] S. Yuan, Y. Solihin, and H. Zhou. "PSSM: achieving secure memory for GPUs with partitioned and sectored security metadata". In: *ICS '21: 2021 International Conference on Supercomputing, Virtual Event, USA, June 14-17, 2021*. Ed. by H. Zhou et al. USA: ACM, 2021, pp. 139–151.

[Yua21b] S. Yuan et al. "Analyzing Secure Memory Architecture for GPUs". In: *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2021, Stony Brook, NY, USA, March 28-30, 2021*. IEEE, 2021, pp. 59–69.

[Yua22] S. Yuan et al. "Adaptive Security Support for Heterogeneous Memory on GPUs". In: *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*. IEEE, 2022, pp. 213–228.

[Yud22] A. W. B. Yudha et al. "LITE: a low-cost practical inter-operable GPU TEE". In: *ICS '22: 2022 International Conference on Supercomputing, Virtual Event, June 28 - 30, 2022*. Ed. by L. Rauchwerger et al. ACM, 2022, 7:1–7:13.

[Zha00] Z. Zhang, Z. Zhu, and X. Zhang. "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality". In: *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 33, Monterey, California, USA, December 10-13, 2000*. USA: ACM/IEEE Computer Society, 2000, pp. 32–41.

[Zha05] Y. Zhang et al. "SENSS: Security enhancement to symmetric shared memory multiprocessors". In: *in Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 2005.

[Zou19] Y. Zou and M. Lin. "FAST: A Frequency-Aware Skewed Merkle Tree for FPGA-Secured Embedded Systems". In: *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2019.

[Zuo20] P. Zuo et al. "SEALing Neural Network Models in Secure Deep Learning Accelerators". In: *CoRR* abs/2008.03752 (2020). arXiv: `arXiv:2008.03752`.