Adaptive Security Support for Heterogeneous Memory on GPUs

Shougang Yuan[1], Amro Awad[1], Ardhi Wiratama Baskara Yudha[2], Yan Solihin[2], Huiyang Zhou[1]

[1] Dept. of Electrical & Computer Engineering, North Carolina State University, Raleigh, NC, USA
{syuan3, ajawad, hzhou}@ncsu.edu
[2] Dept. of Computer Science, University of Central Florida, Orlando, FL, USA
{yudha@knights.ucf.edu, Yan.Solihin@.ucf.edu}

***Abstract*— The wide use of accelerators such as GPUs necessities their security support. Recent works [17], [33], [34] pointed out that directly adopting the CPU secure memory design to GPUs could incur significant performance overheads due to the memory bandwidth contention between regular data and security metadata. In this paper, 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 an on-chip shared counter for such data regions and remove the corresponding parts in the Bonsai Merkel Tree (BMT), thereby reducing the traffic due to encryption counters and the BMT. Second, we detect the common streaming data access pattern 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 without sacrificing the security guarantees. Our evaluation shows that our scheme can achieve secure memory on GPUs with low overheads for memory-intensive workloads. Among the fifteen memory-intensive workloads in our evaluation, our design reduces the performance overheads of secure GPU memory from 53.9% to 8.09% on average. Compared to the state-of-the-art secure memory designs for GPU [17], [33], our scheme outperforms PSSM by up to 41.63% and 9.5% on average and outperforms Common counters by 84.04% on average for memory-intensive workloads. We further propose to use the L2 cache as a victim cache for security metadata when the L2 is either underutilized or suffers from very high miss rates, which further reduces the overheads by up to 4% and 0.65% on average.**

***Keywords*-GPUs, secure memory, heterogeneous memory, encryption, integrity check, security metadata cache**

I. Introduction

Graphic Processing Units (GPUs) are a major computing resource in the cloud to accelerate a wide range of workloads like machine learning and scientific computing. However, current GPUs do not support trusted execution environments (TEEs), which makes them vulnerable to a wide range of attacks, including passive eavesdropping between a GPU and its device memory [28], cold boot attacks [9], row-hammer attacks [16], etc. Hence, there are growing needs to provide TEEs for GPUs. CPU TEEs, such as Intel SGX [7], mainly consider two types of threats: compromised system software (including the operating system and hypervisor) and physical attacks. To defend against the physical attacks, CPU TEEs assume that the CPU chip forms the Trusted Computing Base (TCB), and provide several security guarantees including data confidentiality, integrity, and freshness.

Some recent works have tried to provide TEEs for GPUs [10], [11], [17], [29], [33], [34]. Graviton [29] and HIX [11] assume that the system software stack, including the GPU drivers and the operating system (OS), cannot be trusted, and the hardware (i.e., the PCIe bus) is exposed to the attackers. To protect against software attacks, Graviton offloads the GPU management operations from GPU drivers to the GPU command processor. HIX, alternatively, isolates the GPU drivers from the kernel space and relies on CPU enclaves for protection. To protect against physical attacks like the PCIe bus snooping, both Graviton and HIX provide schemes to encrypt the data transferred over the PCIe bus. However, both Graviton and HIX include the GPU device memory in the TCB, which makes them vulnerable to physical attacks against GPU device memory. Common Counters [17], PSSM [33] and the work by Yuan et al. [34] choose to protect the GPU device memory with the similar protection schemes to CPU TEEs, and conclude that secure memory architectures for CPUs cannot be directly adopted for GPUs. The main reason is that the security metadata, including encryption counters, message authentication codes (MACs), and integrity tree nodes lead to severe bandwidth competition with normal data accesses. Common Counters [17] exploit the uniform memory access patterns in GPU applications by using common-value encryption counters to save memory bandwidth due to accessing counters. PSSM [33] recognizes that due to partitioned memory architecture on GPUs, there can be a high amount of redundant metadata if they are constructed using either virtual or physical addresses. It then proposes to use partition-local addresses, which are the addresses after partition mapping, to generate

the metadata so as to eliminate such redundancy. Furthermore, it also proposes to re-organize counter blocks for sectored accesses. However, none of the previous works explored how the unique memory characteristics of GPU heterogeneous memory space can be leveraged to improve the performance of secure memory in GPUs.

In this work, we start with investigating the security guarantees of CPU TEEs, and point out how the heterogeneous nature of GPU memory space opens new opportunities for high-performance secure implementations while retaining the same security guarantees. GPUs have a heterogeneous memory system including global memory, local memory, shared memory, texture memory, and constant memory. Moreover, even for general-purpose memory spaces (e.g., global memory), many GPU workloads feature streaming memory access patterns. Our work leverages these GPU-specific features to reduce the performance overhead due to security metadata accesses.

In CPU TEEs like Intel SGX [3], [7], there are three security mechanisms to protect against physical attacks. First, **Confidentiality (C)**, achieved usually with counter-mode encryption, protects against memory bus snooping and memory scanning attacks. Second, **Integrity (I)**, achieved with MAC (Message Authentication Code), protects against memory tampering attacks. Third, **Freshness (F)**, achieved usually with integrity trees such as Bonsai Merkel Trees (BMTs) [30] or Intel counter trees [7], protects against replay attacks.

Our key observation is that for different types of GPU memory, we can achieve the same security guarantees using different mechanisms. For example, constant memory contains read-only data, which makes replay attacks meaningless. Therefore guaranteeing integrity without freshness checks for constant memory would be sufficient, which eliminates the need for per-block encryption counters. This observation also holds for other types of data as long as the data are not altered during kernel execution. For many GPU applications, all or parts of their input data, although residing in global memory, are also read only either completely or mostly during kernel execution. In Table I, we list different types of GPU memory defined in the GPU programming models like CUDA or OpenCL and the necessary security mechanisms. Among them, on-chip memory does not need protection as we include the GPU die in the TCB. Similarly, in Table II, we list the types of the off-chip GPU memory data based on their purpose and the associated security mechanisms. Although the read-only information can be readily available from the host programs (e.g., the OpenCL specification of read buffers) or from compiler analysis, we propose a lightweight hardware-based read-only detector, which is capable of detecting part of a buffer (e.g., an array) as read only as long as the kernel does not alter it. Once the region is updated, the detector will no longer mark the region as read only and will start employing per block counters as well as fresh checks accordingly.

By leveraging the read-only property, we can reduce the cost associated with encryption counters and the BMT. However, as identified in the previous work [33], the MAC traffic can incur high performance overheads for memory-intensive workloads. To address this challenge, we propose dual-granularity MACs. Rather than the fixed MAC granularity computed for each cache line/block, we use chunk/page-level MAC for streaming data and block-level MAC for random accessed data. To do so, we propose a hardware-based streaming data detector, which classifies each memory chunk as streaming- or random-accessed and employs the MACs accordingly. Note that our proposed detectors, either the read-only or streaming, do not need to be 100% accurate. A miss-classification would simply mean lost opportunities for performance saving rather than security vulnerability or correctness violation. However, there are subtle issues to be addressed as detailed in Section IV.

In summary, this work makes the following contributions:

- We analyze the security mechanisms of CPU TEEs and make the important observation that not all the different types of GPU memory require the same security mechanisms. The reason behind is that, GPUs' memory system is heterogeneous and features unique data accessing properties.
- Based on heterogeneous GPU device memory, specifically, for read-only, we re-architect counter-mode encryption and freshness protection to improve the performance and better utilize the precious memory bandwidth in GPUs.
- As MAC accesses can be a major performance overhead, we propose dual-granularity MACs to reduce the bandwidth consumption.
- We propose two hardware-based detectors. One for detecting read-only regions and the other for streaming-accessed chunks.
- We present performance evaluation and show that our scheme can reduce the GPU secure memory performance overhead from 53.9% to 8.09% on average for memory-intensive workloads.
- We further propose to use the L2 cache as the victim cache for security metadata caches when the L2 is either underutilized or suffers from a very high miss rates, which further reduces the overheads by up to 4% and 0.65% on average.

II. Background and Related Work

A. Threat Model and Scope of Our Work

CPU TEEs such as Intel SGX assume that the system software stack including the OS and hypervisor cannot be trusted, and the servers can be exposed to the attackers who may have physical control over the compute units [7]. To protect against physical attacks, CPU TEEs assume that the

Table I
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 II
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

processor chip forms the TCB [15]. In other words, all the data leaving the processor chip need to be protected and verified when fetched later. To this end, CPU TEEs provide three major security guarantees, namely, *confidentiality, integrity and freshness*.

This paper assumes that GPUs' device memory, specifically, the GDDR memory, is also vulnerable to the physical attacks considered in CPU TEEs. The reason is that GDDR memory is off-chip for GPUs and can be fully exposed to attackers with physical access to the device. Thus, we exclude the GPU GDDR memory modules from the TCB, and assume that the GPU chip forms the security boundary. High bandwidth memory (HBM), however, is not vulnerable to physical attacks if it is soldered within the GPU chip package, and is out of the reach for attackers.

The scope of this paper covers the protection of heterogeneous memory on GPUs, and we propose different schemes to optimize the overheads resulting from the bandwidth consumption of accessing security metadata. GPU context isolation and management as well as PCIe bus protection scheme have been addressed in previous works [11], [17], [29], and are assumed in our design. Furthermore, defending against GPU side channel attacks [5], [12], [18] such as timing-based side channel attacks is out of the scope of this work.

B. Physical Attacks and Security Guarantees on CPU TEEs

CPU TEEs assume two types of physical attacks: *passive attacks and active attacks*. Attackers with ability to perform passive attacks can silently *snoop the memory bus or scan memory chips* to steal the critical information from the processor chip. Meanwhile, attackers with ability to perform active attacks can *tamper with memory content*. Through memory tampering attacks, attacker can randomly change some bits in off-chip memory or communication channels or carefully replay some old values to replace the off-chip data.

To defend against the aforementioned attacks, CPU TEEs define three security guarantees:

Confidentiality: The underlying mechanism to guarantee the memory confidentiality is encryption, which enforces every piece of data transferred over the memory bus or stored in off-chip memory must be in ciphertext. By doing so, attackers cannot get any meaningful information without knowing the encryption key. Counter-mode encryption is a commonly used way for low-latency encryption [24] in CPU TEEs. With counter-mode encryption, per block counters are maintained. A per block counter together with the block address, encryption CID (a 128B cache line need to be broken into multiple 16B chunks as the output size of AES is 16B/128 bits) will be encrypted to generate a one-time pad (OTP) for each last-level cache (LLC) write back, where the same pad is used for decryption in case of a memory read from the same address. The memory controller can get the plaintext (ciphertext) by XORing the OTP with ciphertext (plaintext). In counter-mode encryption, the encryption counters cannot be reused because that would enable known-plaintext attacks. State-of-art secure memory implementations adopt split-counter organization [2], [22]–[24], [30], [30], [31], in which a major counter shared by many memory blocks in a large size memory region (e.g., one physical memory page) and a small minor counter is maintained for each block. When a minor counter overflows, only the blocks within the same memory region need to be re-encrypted. The OTP generation of counter-mode encryption with split counters is shown as step ① and step ② in Fig. 1.

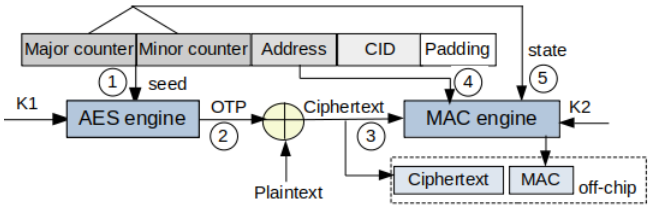

Figure 1. Counter-mode encryption and MAC generation.

Integrity: While memory encryption can protect against passive attacks, attackers with the ability to perform active attacks can modify the values in off-chip memory. Hence, TEEs also need to ensure *memory integrity*. The underlying mechanism to ensure memory integrity is MAC. As shown in step ③, step ④ and step ⑤ in Fig. 1, at each LLC write-back, the MAC is computed and stored in off-chip memory. Note that state-of-art CPU TEEs also include the encryption counters in the MAC computation, in which the counters play a role of state. This scheme is proposed in [2], [22], [30] and named stateful MACs. At each memory read, the pre-computed MAC is fetched together with the ciphertext,

the memory controller reproduces the MAC based on the fetched ciphertext and compares it with the fetched MAC. If there is a mismatch, the memory controller would view it as an attack and raise an exception.

Freshness: MAC can ensure the integrity. However, attackers can carefully monitor the off-chip memory, and replace the memory block content with some stale values that were legally generated by the processor in the past [27]. In this case, the attacker can bypass MAC verification without knowing the key of the MAC engine. This scheme is known as a replay attack. To defend against replay attacks, the CPU TEEs define the *freshness* guarantee. The mechanism to ensure the freshness is integrity trees. An integrity tree is a hash tree that covers the off-chip memory, while the tree root is stored in an on-chip register. At each memory read or write, the memory controller will traverse the integrity tree to either verify the data read from off-chip memory or update the tree nodes from leaf to root for writes. Fig. 2 illustrates different designs of integrity trees. Early CPU TEEs use a regular or standard Merkle Tree to detect replay attacks. However, as a standard Merkle Tree needs to cover all the data and encryption counters, the integrity tree is large. State-of-the-art CPU TEEs adopt different variants of standard Merkle Tree, such as Bonsai Merkle Trees (BMT) which covers only the encyption counters. In this work, we use BMT in our evaluation while our proposed schemes are independent upon the integrity tree implementation.

C. ECC on GPU GDDR Memory

Recent works [32] [25] show that the overhead of accessing MACs for secure memory can be optimized by re-purposing error correction codes (ECC). One may try to adopt the same scheme to address the bandwidth requirements of accessing MACs for GPUs. However, the ECC implementations on GDDR memory is different from conventional ECC designs on CPU DIMMs, in which a dedicated memory chip is used to store the ECC. On GDDR memory, as officially documented on CUDA toolkits [4]: *"On GPUs with GDDR memory with ECC enabled the available DRAM is reduced by 6.25% to allow for the storage of ECC bits. Fetching ECC bits for each memory transaction also reduced the effective bandwidth by approximately 20% compared to the same GPU with ECC disabled, though the exact impact of ECC on bandwidth can be higher and depends on the memory access pattern."* In other words, GDDR memory treats ECC as regular data and does not have dedicate channels to access ECC. This observation motivates our dual-granularity MAC design, which aims to reduce the MAC bandwidth while being compatible with GDDR memory architecture. Although it is possible to add ECC chips for GDDR memory, such a design incurs high bandwidth overhead (e.g., 1/8 bandwidth if 1B ECC for 8B data). Unlike CPUs, which are latency sensitive, GPUs are bandwidth sensitive and it is always desirable to use all the bandwidth for data access.

III. Motivation and Design Principles

A. 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 I. One observation we make is that the integrity tree does not need to cover read-only spaces like constant and texture memory.

Moreover, as pointed out in previous work [17], 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 II. Similar to the constant and texture memory spaces, these read-only data also do not need freshness checks.

B. Seed Generation in Counter-Mode Encryption

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., over-written by the host) across different kernel invocations.

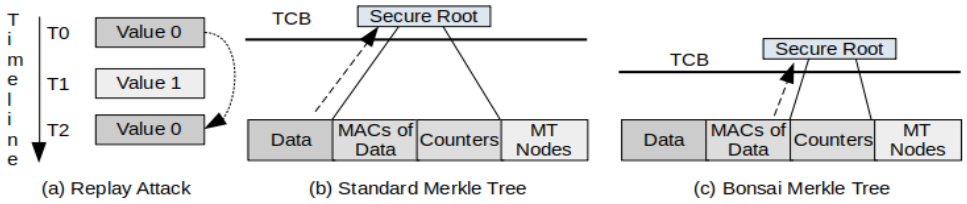

Figure 2. Replay attacks and different integrity tree designs to defend them.

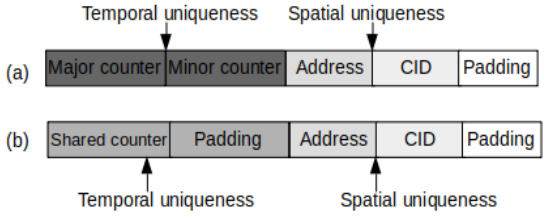

Figure 3. Seed generation for (a) not-read-only data and (b) read-only data.

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. 3(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 IV). 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. Consequently, the memory bandwidth overheads due to integrity tree traversing are also eliminated.

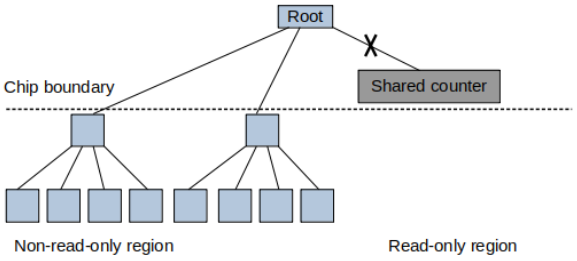

Figure 4. Integrity tree with read-only regions excluded.

C. Overhead of MAC Accesses

As pointed out by previous work [33], 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 [33] truncates the MAC from 8B to 4B. However, truncating the MAC reduces the collision space as proved by the birthday attack paradox [8] – "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 [17], [34], 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. 5, 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.

IV. Architecture Design

A. Overall Architecture

Similar to previous works [17], [33], [34], we assume that the GPU chip forms the TCB. The overall GPU secure memory architecture is shown in Fig. 6. We adopt a scheme similar to PSSM [33], which integrates the memory encryption engine (MEE) into each memory controller and

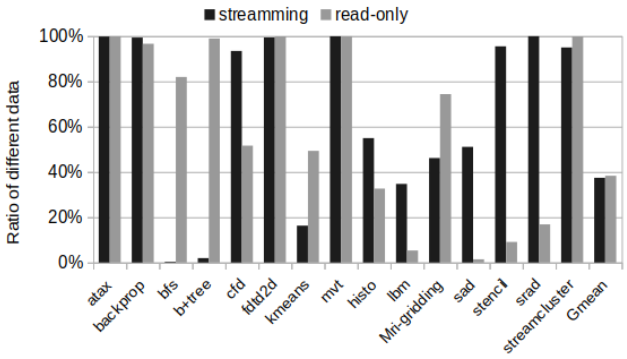

Figure 5. The ratio of memory accesses (i.e., L2 misses and L2 write backs) accessing streaming data as well as read-only data in various GPU workloads.

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 [33]. 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 as discussed in Section II; 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. 7, 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 [33], 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.

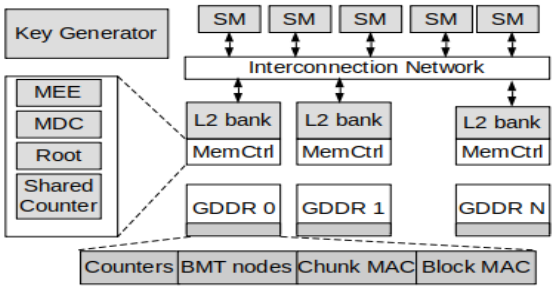

Figure 6. Overall architecture.

B. 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. 8 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 III-B. 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. 9. 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 [17], 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.

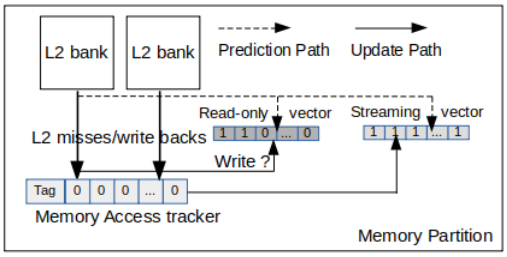

Figure 7. The read-only detector and streaming detector in a memory partition. Their inputs are the LLC misses and write backs.

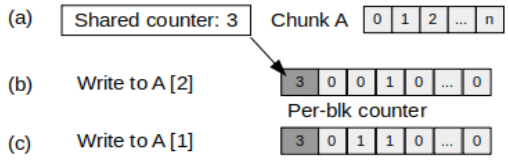

Figure 8. An example showing the propagation from the shared counter to the per block counters.

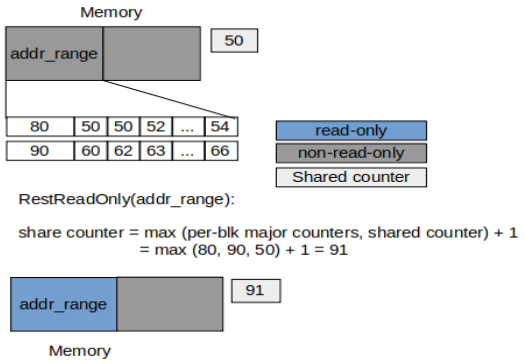

Figure 9. The process of shared counter update when using the $InputReadOnlyReset(addressrange)$ API.

C. 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. 7. 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 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.

D. 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 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 [21].

Table III
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

Table IV
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

V. METHODOLOGY

We model our proposed schemes with GPGPU-Sim v4.0 [13]. Our baseline GPU configuration is shown in Table V, which is based on the Nvidia Turing architecture [20]. We

Table V
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 VI
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

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 [33], in which the partition-local offset is used to construct the security metadata. The detailed MDC and MEE organizations are listed in Table VI.

Our benchmarks are from the Rodinia-3.1 [1], Parboil [26] and Polybench [6] 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 sensitive to secure memory, we choose 15 memory intensive workloads and report the benchmark details in Table VII, 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.

Table VII
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 VIII
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 [17] scheme, and the security metadata is constructed with physical address.
PSSM	Secure GPU memory with PSSM scheme [33]
PSSM_cctr	Secure GPU memory with common counters scheme, and the security metadata is constructed from local address as PSSM [33] 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.

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

A. Hardware Overheads

The storage overhead of our proposed hardware components are listed in Table IX. 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.

Table IX
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

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).

VI. EVALUATION

A. 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. 10. 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.

B. Streaming Access Pattern Detection

We use 8 memory access trackers in each memory partition and report the results in Fig. 11. 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. 11. As shown in the figure, our design

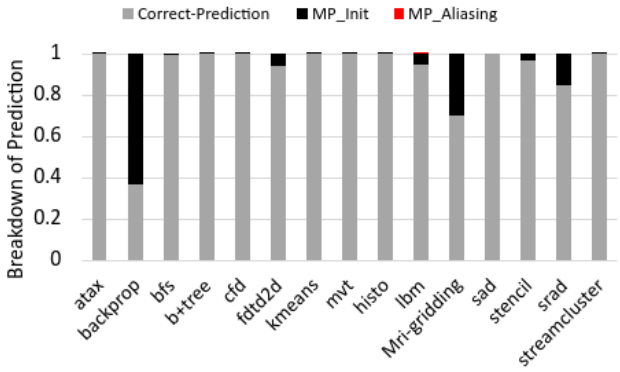

Figure 10. Breakdown of read-only predictions.

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 IV, not all mispredictions incur the same bandwidth overheads.

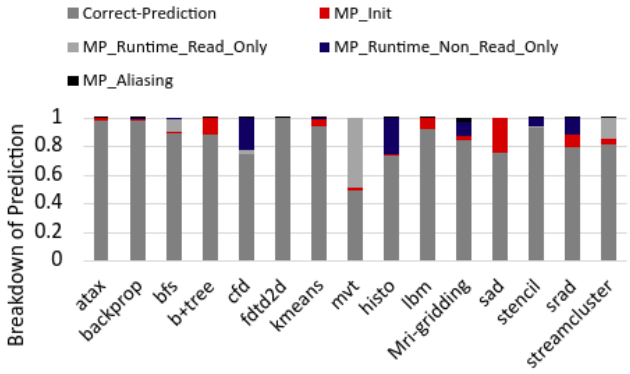

Figure 11. Breakdown of streaming pattern predictions

C. Overall Performance

We evaluate our secure heterogeneous memory design and compare it with different previous works, as shown in Fig. 12. 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 [33]. 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.

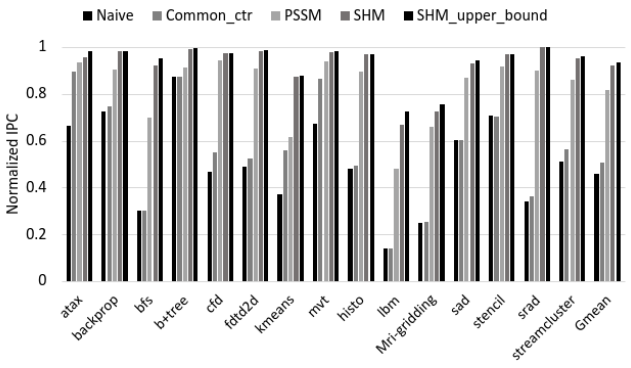

Figure 12. Normalized IPC of different secure GPU memory designs.

D. Performance Breakdown

To examine the effectiveness of different optimizations, we include them one at a time and show the results in Fig. 13. From Fig. 13, 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 VI-C. Fourth, our SHM design is compatible with the common counters scheme. As we can see from Fig. 13, 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.

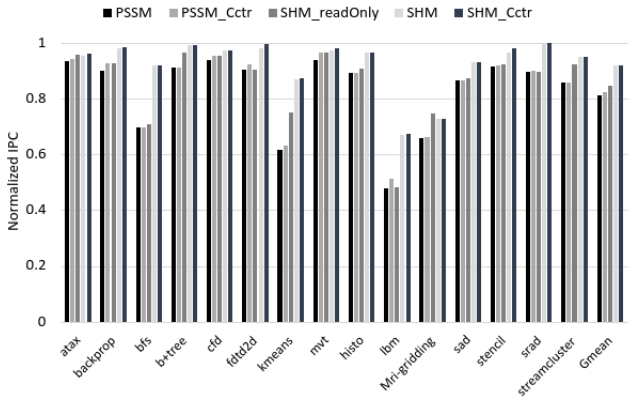

Figure 13. Performance impacts of different optimizations.

E. 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. 14. 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. 12. On the other hand, for the benchmarks with relatively low bandwidth utilization, such as atax or mvt (Table VII), 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. 5, 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. 5. 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.

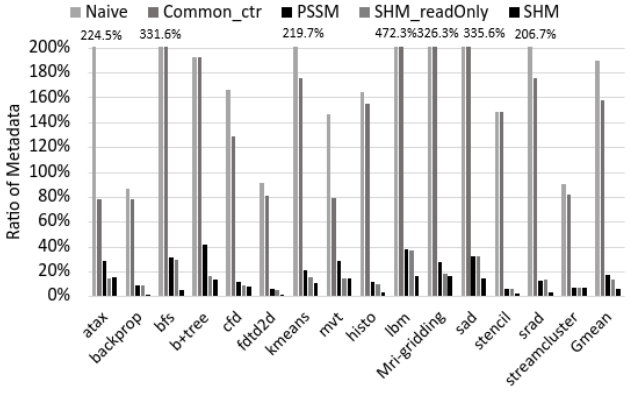

Figure 14. Bandwidth overheads due to security metadata, normalized to regular data bandwidth, of different designs.

F. Power Saving

We report the power efficiency of our SHM design, and compare it with the prior works, as shown in Fig. 15. We extend the GPUWattch [14] to model the power and energy consumption of different designs. We use CACTI_v6.5 [19] 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.

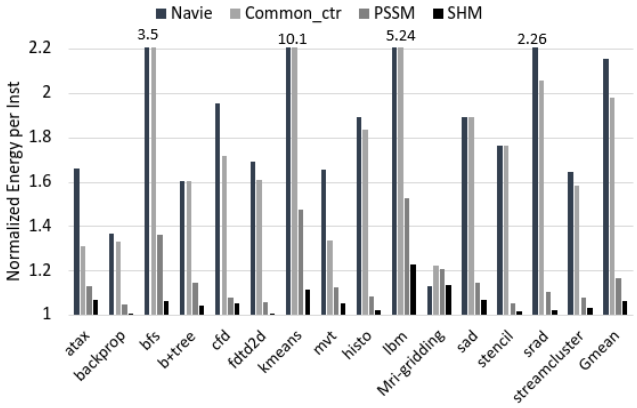

Figure 15. Normalized Energy Consumption per Instruction for different designs.

G. Using L2 as a Victim Cache

We present our results of using L2 as the victim cache for security metadata caches in Fig. 16. 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. 16, 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.

VII. Conclusions

Security metadata traffic is the key performance bottleneck for GPU secure memory. In this paper, we propose

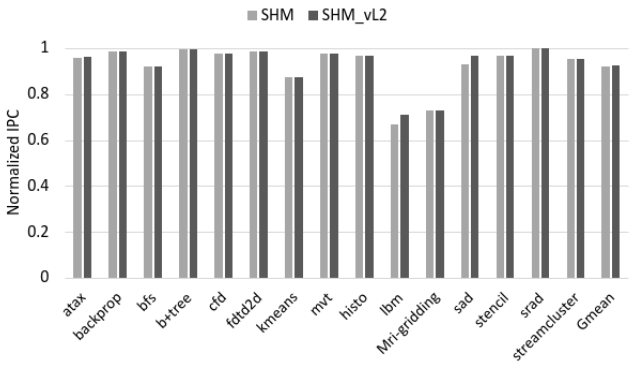

Figure 16. Normalized IPC when enabling L2 as a victim cache for security metadata.

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.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable comments. For this work, the NSCU team is funded in part by NSF grants 1717550 and 1908406, an AMD gift fund, and Office of Naval Research (ONR). The UCF team is supported in part by NSF grant 1908079, AMD gift, and UCF. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for public release. Distribution is unlimited.

REFERENCES

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*. USA: IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[2] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, "Making secure processors OS- and performance-friendly," *ACM Trans. Archit. Code Optim.*, vol. 5, no. 4, pp. 16:1–16:35, 2009. [Online]. Available: https://doi.org/10.1145/1498690.1498691

[3] I. Corporation, "Intel® 64 and ia-32 architectures software developer's manual (325462-071us)," Intel Corporation, USA, Tech. Rep., 2019.

[4] CUDAToolkit, "Cuda c++ best practices guide," 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#performance-metrics

[5] Y. Gao, H. Zhang, W. Cheng, Y. Zhou, and Y. Cao, "Electromagnetic analysis of gpu-based AES implementation," in *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*. USA: ACM, 2018, pp. 121:1–121:6. [Online]. Available: https://doi.org/10.1145/3195970.3196042

[6] S. Grauer-Gray and J. Cavazos, "Optimizing and auto-tuning belief propagation on the GPU," in *Languages and Compilers for Parallel Computing - 23rd International Workshop, LCPC 2010, Houston, TX, USA, October 7-9, 2010. Revised Selected Papers*, ser. Lecture Notes in Computer Science, K. D. Cooper, J. M. Mellor-Crummey, and V. Sarkar, Eds., vol. 6548. USA: Springer, 2010, pp. 121–135. [Online]. Available: https://doi.org/10.1007/978-3-642-19595-2_9

[7] S. Gueron, "Memory encryption for general-purpose processors," *IEEE Secur. Priv.*, vol. 14, no. 6, pp. 54–62, 2016. [Online]. Available: https://doi.org/10.1109/MSP.2016.124

[8] G. Gupta, "What is birthday attack??" 2015. [Online]. Available: https://www.researchgate.net/profile/Ganesh-Gupta-7/publication/271704029_What_is_Birthday_attack/links/54cfbdcc0cf24601c0958a1e/What-is-Birthday-attack.pdf

[9] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*. USA: USENIX Association, 2008, pp. 45–60. [Online]. Available: http://www.usenix.org/events/sec08/tech/full_papers/halderman/halderman.pdf

[10] T. Hunt, Z. Jia, V. Miller, A. Szekely, Y. Hu, C. J. Rossbach, and E. Witchel, "Telekine: Secure computing with cloud gpus," in *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, R. Bhagwan and G. Porter, Eds. USA: USENIX Association, 2020, pp. 817–833. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/hunt

[11] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds. USA: ACM, 2019, pp. 455–468. [Online]. Available: https://doi.org/10.1145/3297858.3304021

[12] 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.

[13] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated GPU modeling," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. Spain: IEEE, 2020, pp. 473–486. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00047

[14] J. Leng, T. H. Hetherington, A. ElTantawy, S. Z. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: enabling energy optimizations in gpgpus," in *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, A. Mendelson, Ed. ACM, 2013, pp. 487–498. [Online]. Available: https://doi.org/10.1145/2485922.2485964

[15] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz, "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*. USA: ACM Press, 2000, pp. 168–177. [Online]. Available: https://doi.org/10.1145/356989.357005

[16] O. Mutlu, "The rowhammer problem and other issues we may face as memory becomes denser," *CoRR*, vol. abs/1703.00626, 2017. [Online]. Available: http://arxiv.org/abs/1703.00626

[17] S. Na, S. Lee, Y. Kim, J. Park, and J. Huh, "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. [Online]. Available: https://doi.org/10.1109/HPCA51647.2021.00011

[18] H. Naghibijouybari, A. Neupane, Z. Qian, and N. B. Abu-Ghazaleh, "Rendered insecure: GPU side channel attacks are practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Canada: ACM, 2018, pp. 2139–2153. [Online]. Available: https://doi.org/10.1145/3243734.3243831

[19] N. P. J. Naveen Muralimanohar, Rajeev Balasubramonian, "Cacti 6.0: A tool to model large caches," *4HP Laboratories*, 2009.

[20] Nvidia, "Nvidia turing gpu architecture." [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[21] 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. [Online]. Available: https://doi.org/10.1109/MICRO.2006.49

[22] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors OS- and performance-friendly," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40 2007), 1-5 December 2007, Chicago, Illinois, USA*. USA: IEEE Computer Society, 2007, pp. 183–196. [Online]. Available: https://doi.org/10.1109/MICRO.2007.16

[23] B. Rogers, C. Yan, S. Chhabra, M. Prvulovic, and Y. Solihin, "Single-level integrity and confidentiality protection for distributed shared memory multiprocessors," in *14th International Conference on High-Performance Computer Architecture (HPCA-14 2008), 16-20 February 2008, Salt Lake City, UT, USA*. USA: IEEE Computer Society, 2008, pp. 161–172. [Online]. Available: https://doi.org/10.1109/HPCA.2008.4658636

[24] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. Japan: IEEE Computer Society, 2018, pp. 416–427. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00041

[25] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "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. [Online]. Available: https://doi.org/10.1109/HPCA.2018.00046

[26] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Champaign, IL USA, Tech. Rep., 2009.

[27] 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. [Online]. Available: https://doi.org/10.1145/3173162.3177155

[28] J. K. Tugnait, "Detection of active eavesdropping attack by spoofing relay in multiple antenna systems," *IEEE Wirel. Commun. Lett.*, vol. 5, no. 5, pp. 460–463, 2016. [Online]. Available: https://doi.org/10.1109/LWC.2016.2585549

[29] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on gpus," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, A. C. Arpaci-Dusseau and G. Voelker, Eds. USA: USENIX Association, 2018, pp. 681–696. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/volos

[30] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *33rd International Symposium on Computer Architecture (ISCA 2006), June 17-21, 2006, Boston, MA, USA*. USA: IEEE Computer Society, 2006, pp. 179–190. [Online]. Available: https://doi.org/10.1109/ISCA.2006.22

[31] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in *Proceedings of the 36th Annual International Symposium on Microarchitecture, San Diego, CA, USA, December 3-5, 2003*. USA: IEEE Computer Society, 2003, pp. 351–360. [Online]. Available: https://doi.org/10.1109/MICRO.2003.1253209

[32] S. F. Yitbarek and T. M. Austin, "Reducing the overhead of authenticated memory encryption using delta encoding and ECC memory," in *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*. USA: ACM, 2018, pp. 35:1–35:6. [Online]. Available: https://doi.org/10.1145/3195970.3196102

[33] 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*, H. Zhou, J. Moreira, F. Mueller, and Y. Etsion, Eds. USA: ACM, 2021, pp. 139–151. [Online]. Available: https://doi.org/10.1145/3447818.3460374

[34] S. Yuan, A. W. B. Yudha, Y. Solihin, and H. Zhou, "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. [Online]. Available: https://doi.org/10.1109/ISPASS51385.2021.00017