Common Counters: Compressed Encryption Counters for Secure GPU Memory

Seonjin Na, Sunho Lee, Yeonjae Kim, Jongse Park, Jaehyuk Huh
School of Computing, KAIST
{sjna, shlee, yjkim, jspark, jhhuh}@casys.kaist.ac.kr

***Abstract*—Hardware-based trusted execution has opened a promising new opportunity for enabling secure cloud computing. Nevertheless, the current trusted execution environments are limited to the traditional CPU-based computation, while more and more critical workloads such as machine learning require to use GPUs. For secure GPU computing, recent studies proposed to isolate critical GPU operations from the operating system by providing trusted execution environments for GPUs. However, a missing component in the prior GPU trusted execution studies is a hardware-based memory protection technique optimized for GPUs. Start-of-art memory protection techniques for CPUs use counter-mode encryption, where per-block counters are used as timestamps for generating one-time pads. The integrity of counters is protected by a counter tree. This study investigates the hardware memory encryption for GPUs and identifies counter cache misses as one of the key performance bottlenecks. To mitigate the overheads of counter cache misses, this paper proposes a novel technique, called *common counters*, for efficient counter-mode encryption and counter integrity protection. The proposed technique exploits unique characteristics of common GPU applications. In GPU applications, a large portion of application memory is written only once by the initial data transfer from the CPU memory, or the number of writes by the GPU applications to major data structures tends to be uniform. Exploiting the uniform write property, this study proposes to use a common counter representation to complement the existing per-block encryption counters. With the common counters, the proposed technique can almost eliminate the performance overheads caused by counter cache misses, reducing the degradation to 2.9%.**

I. Introduction

Hardware-based trusted execution environments (TEEs) such as Intel Software Guard Extension (SGX) [17] and ARM TrustZone [1] provide secure execution of user applications on untrusted remote cloud servers. Such hardware TEEs open a new direction for trusted clouds by isolating and protecting user execution environments from compromised privileged software and physical attacks. Nevertheless, a critical limitation of the current TEE supports is the lack of consideration for widely used GPU-based computation. Along with a wide deployment of machine learning (ML) workloads, the GPU computation has become an essential part of computing systems, especially in clouds providing ML-based services. To protect the essential GPU computation, the scope of TEEs must be extended to GPU computation.

To address the lack of GPU TEEs, recent studies proposed hardware-based hardening techniques for such GPU execution environments [19], [50]. Graviton detaches the critical GPU management operations from the CPU-side driver, and isolates them in the GPU-side command processing unit, protecting critical GPU operations from malicious privileged software [50]. It requires to change the GPU command processing subsystem to provide such isolated GPU management inside a GPU. Alternatively, HIX proposed to protect PCIe I/O subsystems from privileged software and protect the GPU driver with a CPU TEE [19]. While HIX does not require to change GPUs, its threat model is weaker than Graviton since HIX does not provide the protection from direct physical attacks.

Although the prior studies proposed the GPU system designs for the trusted GPU computing, a critical unaddressed component is a hardware-based memory protection technique optimized for GPUs. Graviton relies on the availability of 3D stacked DRAM for GPUs, and assumes that the memory contents in the 3D stacked DRAM cannot be leaked or compromised. However, not only do many discrete GPUs use GDDRx memory, but also integrated GPUs use conventional DDRx memory. Therefore, to provide general supports for GPU TEEs, the hardware memory protection must be available for GPU memory as well. However, the existing hardware-based memory protection schemes require costly encryption and integrity verification. Figure 1 (a) depicts the secure GPU execution system proposed in Graviton [50] with the trusted stacked memory, and Figure 1 (b) shows our target GPU with conventional untrusted memory.

Recently, there have been significant improvements of hardware-based memory protection for trusted CPUs [11], [37], [38], [40], [46], [53], [54]. The state-of-art memory encryption uses a counter-based encryption, which generates a one-time pad (OTP) for a given encryption key, address, and counter. Each cacheline unit of memory has a separate encryption counter, which is incremented for every data write-back to the memory, guaranteeing its freshness. The counters are cached in a counter cache, and as long as the counter for data missed in the last-level cache (LLC) is available in the on-chip counter cache, the OTP for the missed data can be prepared before the data returns from the memory, requiring a simple XOR operation for decryption with the OTP. The integrity validation uses a message authentication code (MAC) for each cacheline data, and the verification of counter values employs a tree of counters.

However, even with the recent improvements for the processor memory encryption and integrity protection, the protection of GPU memory is yet to be investigated. In our study, we

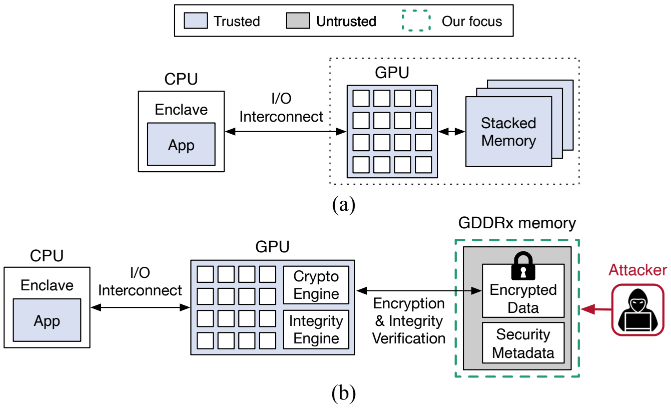


Fig. 1: (a) Trusted GPU execution in Graviton [50], exclusively designed for GPUs equipped with trusted stacked memory, and (b) the proposed memory protection technique that provides confidentiality and integrity for untrusted conventional GDDRx memory in more general systems.

employs the trusted execution model for GPUs proposed by Graviton [50], but the memory is untrusted with conventional GDDRx. To the best of our knowledge, this study is the first study to assess the performance impact of hardware GPU memory protection and to propose a novel technique to mitigate the performance degradation. Our investigation shows that counter cache misses can cause a significant performance degradation for GPU applications, which have low spatial localities for counter blocks. As GPUs are employed for their high performance, the performance overheads for security might be delaying fast and wide adoption of trusted computing for GPUs.

To mitigate the performance degradation by memory protection, this paper proposes a novel counter representation technique for GPUs, called COMMONCOUNTER, exploiting unique characteristics of GPU applications. For GPU applications, a large portion of the application memory is written once by the initial data transfer from the CPU memory. In addition, the execution of many GPU kernels tends to generate the uniform number of writes to each element of major data structures, and thus after a kernel execution, the counter value of each memory block of a common data structure tends to have the same value. Therefore, when a GPU application consists of multiple execution of kernels with data dependence, those counter values are uniformly incremented across the kernel execution flow.

Exploiting the write-once and uniform progress property of counter values in GPU applications, this paper proposes a *common counter* representation, which augments the existing per-cacheline counter management. The proposed mechanism efficiently identifies which memory chunks have the same counter values for their cacheline unit of data after a kernel execution or data transfer from the CPU-side code. During a kernel execution, if the counter for a missed cacheline can be served from a few known common counter values, the counter cache is bypassed, and the requested counter is directly provided by the common counters. Since the corresponding counter value is obtained from a small number of on-chip common counters for the majority of LLC misses, counter cache misses are reduced significantly.

To enable common counters, the GPU context for each application must use a different memory encryption key since all the counters for the application memory must be reset during the context creation. Once a GPU context is created by the trusted GPU command processor, memory pages for the context are encrypted by a per-context encryption key, and counter values for allocated memory blocks for the context are reset. This counter resetting does not violate the security requirement, since the memory pages of a new context is encrypted by a new key.

Our evaluation based on the simulation with GPGPU-Sim shows that the proposed counter management technique can almost eliminate the performance overheads by counter cache misses. When combined with the prior MAC optimization [39], COMMONCOUNTER reduces the performance degradation by memory protection to 2.9% for a set of GPU applications, while the latest improvement using Morphable counters [38] can impose 11.5% degradation with the same MAC optimization.

As the first study for the hardware-based GPU memory protection, the contributions of this paper are as follows:

- This paper applies the latest existing CPU-based memory protection techniques to the trusted GPU execution model. It identifies that counter cache misses are one of the key sources of performance degradation in secure GPU execution.
- To reduce the overheads of counter cache misses, the paper exploits read-only data abundant in GPU applications.
- The paper identifies a unique update behavior of counter values in GPU applications, where the majority of counter values often progress with a few common values within an application.
- The paper proposes an efficient mechanism to identify common counters and to provide common counters for LLC miss handling, which allows effective bypassing of the counter cache.

The rest of the paper is organized as follows. Section II describes the trusted GPU execution and the latest improvements for memory protection in CPU memory. Section III investigates the unique characteristics of GPU memory writes. Section IV proposes our new common counter architecture, and Section V evaluates the proposed technique with experimental results. Section VI discusses the potential extension. Section VII presents the related work, and Section VIII concludes the paper.

II. BACKGROUND

A. Trusted Execution for GPUs

Trusted execution environments (TEEs) for CPUs provide isolated execution environments, protected from malicious privileged software and physical attacks. Intel Software Guard Extension (SGX) allows a user application to create a TEE

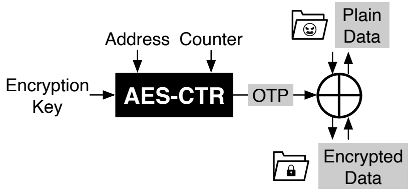


Fig. 2: Counter mode encryption.

called *enclave*, and its context including an execution environment and memory pages is protected by the hardware mechanism. To support TEEs, first, the access isolation mechanism prevents any privileged or unprivileged non-enclave code from accessing the enclave execution environments and pages. Second, the hardware memory protection mechanism supports the confidentiality and integrity of enclave memory pages, under potential physical attacks on the external DRAM. Third, the attestation mechanism allows remote or local users to validate the TEE environments and codes running in an enclave.

To provide TEEs for GPUs, recent two studies proposed two different ways with trade-offs. First, Graviton proposed a complete hardware-based isolation of a GPU from the rest of the system [50]. Each GPU is embedded with a unique private-public key pair to identify itself. In addition, the critical GPU management operations such as creating a context and allocating memory pages must be done by the command processor inside a GPU. By conducting such critical operations inside the GPU, a user application can directly and securely send commands to the GPU, bypassing any interference from the operating system. To protect the user application, the CPU-side user application runs on an enclave, and the CPU enclave and GPU establish trust during a context creation by attesting the GPU environments and by sharing a common key.

An alternative way is not to require any GPU change. HIX proposed to provide GPU TEEs without changing commodity GPUs [19]. Instead of requiring to modify GPUs to embed a unique key and to conduct management operations within a GPU, HIX proposed to secure the PCIe I/O subsystem. The GPU driver is isolated from the operating system by running it on a special GPU enclave. The PCIe routing table is locked down, so that the compromised operating system cannot arbitrarily re-route packets intended to the GPU. Although HIX allows a TEE for commodity GPUs and it can easily support other PCIe-connected accelerators, it does not provide the protection against physical attacks, since the PCIe interconnects are exposed in the motherboard.

Since this study proposes GPU memory protection against physical attacks, it is based on the approach taken by Graviton [50]. A GPU is modified to isolate itself from its driver in the operating system. However, one critical component of TEEs missing in the prior approaches is how to protect the memory of GPU. In CPU TEEs, there have been significant improvements to provide the confidentiality and integrity of memory data existing in the external unsecure DRAM. However, Graviton assumes that 3D stacked DRAM such as HBM (High Bandwidth Memory) is available for GPUs, and the data in the 3D stacked DRAM is always safe even from any possible physical attacks. However, many commodity GPUs will still be relying on much less expensive GDDRx memory [33]–[35], vulnerable from physical attacks. In addition, embedded GPUs share the same DDRx memory with CPUs [3], [9], [10]. Considering the wide-spread use of GDDRx and its vulnerability from physical attacks, it is essential to investigate the performance implication of hardware-based memory protection for GPUs.

B. Threat Model

The trusted computing base (TCB) of this study is the secure GPU chip and the GPU software running on it. In addition, as assumed by the prior study [50], the CPU-side user application initiating the GPU application is running in a CPU enclave. Therefore, the CPU chip and user software running in the enclave are also included in TCB. The study assumes that attackers can have a full control over the operating system and other privileged software, and can wield physical attacks such as probing buses, on exposed system components including CPU memory bus, PCIe interconnects, and GPU memory interconnects.

However, this study does not provide the availability of GPU computation, as a compromised OS can block the execution of the CPU-side application. In addition, side channel attacks [16], [23], [24], [32], [52] on GPUs or CPUs are out of scope of this paper. The threat model of this paper is in general identical to that of Graviton [50], except for vulnerability of GPU GDDRx memory.

C. Memory Protection for Trusted Execution

Memory protection requires to support both confidentiality and integrity of data resident in the untrusted external DRAM. When a memory block is brought into the on-chip caches, the block is decrypted and its integrity must be verified. When a modified cacheline is evicted, the memory block is encrypted and integrity meta data is also written along with the encrypted data. The common hardware encryption for confidentiality employs a counter-based encryption using one-time pad (OTP) [42], [53], [54].

As shown by Figure 2, the encryption is done by 1) creating an OTP from the encryption key, address, and counter value, and 2) by XOR'ing the OTP with the evicted cacheline. A symmetric block cipher such as AES is used to generate OTPs. For each cacheline unit of memory block, a separate counter must be maintained, and the counter value is incremented whenever a dirty eviction from the on-chip cache updates the memory block. The counter for each block guarantees the freshness of the encrypted data even though the same encryption key is used.

The integrity verification detects any compromisation of data. For the integrity verification, variants of Merkle tree are employed in prior studies and commercial processors [30]. A straightforward design creates a tree of hash values from the entire memory blocks [43]. The root of the tree never leaves

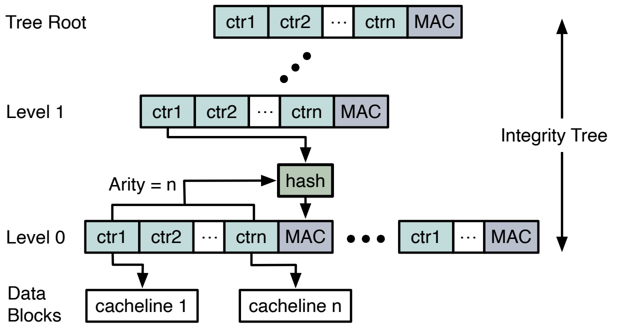


Fig. 3: Structure of integrity tree for replay-attack protection.

the secure processor, but the rest of the hash tree nodes can exist in the unsecure memory. For performance, part of the hash tree can be cached in a hash cache inside the processor. When an LLC miss brings a memory block into the processor, the computed hash value of the memory block is validated against the stored hash value in the tree. If the corresponding hash value does not exist in the hash cache, the hash node must be read from the memory and validated first by the parent node of the hash value.

Improvements for integrity trees have been investigated by combining the hash tree with the counter-mode encryption. The height of the original Merkle tree can be high because the entire data memory region must be covered by the leaf nodes of the tree. An improved Bonsai Merkle tree (BMT) creates a hash tree only of counter values of memory blocks [37]. For each memory block, a keyed hash value or message authentication code (MAC) is generated for integrity verification. In addition, the integrity tree of counters (Bonsai Merkle tree) guarantees the freshness of counters to prevent potential replay attacks. Since BMT covers a much smaller memory region just for counters than the original Merkle tree, its height is much shorter.

There have been more improvements on BMT to pack more counters in a single block of intermediate tree nodes, as shown by Figure 3. The counter block is often organized as the same size of a data cacheline. As more counters can be stored in a single counter block, not only is the efficiency of counter cache improved, but also the height of counter integrity tree is reduced. Split counters decompose a counter value into a minor counter and a major counter [53]. Within a counter block, each counter has a minor counter, and all the counters in the same counter block share a single major counter. If a minor counter hits the limit, the major counter is incremented, causing re-encryption of the corresponding data blocks. VAULT improves the counter integrity tree by adopting a different arity for each level of the tree, to balance between the compact representation of counters and the cost of re-encryption due to minor counter overflows [46]. Morphable counters proposed further compact representation of counters, packing 128 counters per 64B counter block [38]. It dynamically changes the counter representation to match the counter update behaviors of applications.

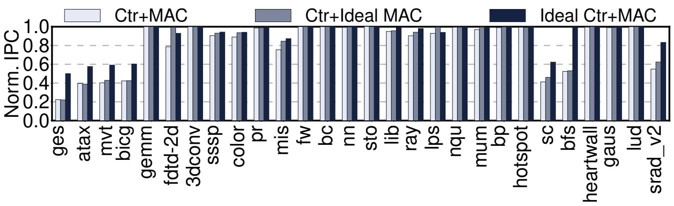


Fig. 4: Performance of the SC_128 technique on GPU, normalized to the vanilla GPU without memory protection.

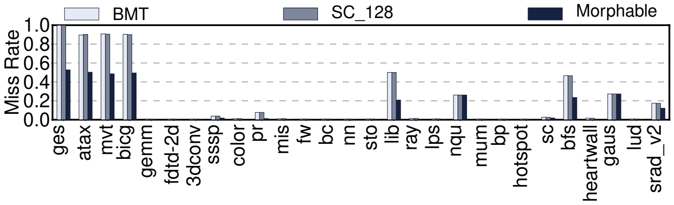


Fig. 5: Counter cache miss rates with three prior schemes.

III. Motivation

A. *Performance Implications of Secure GPU Memory*

We first perform empirical studies via GPU simulations and investigate the performance overheads of existing memory protection schemes when they are deployed on GPUs.

Methodology: We perform the performance characterizations of memory-secured GPUs, in the comparison with the vanilla GPU having no security features as the baseline. We present the detailed methodology for simulation infrastructure in Section V-A. The evaluated memory protection techniques are: (1) Bonsai Merkle Tree (BMT) [37], (2) Split counters with 128 counters per 128B cacheline (SC_128) [53], and (3) Morphable counters (Morphable) [38]. All three of the memory-secured GPUs are equipped with a 16KB counter cache, where BMT and SC_128 pack 128 encryption counters per each cacheline, while Morphable does 256 counters.

Performance implication of counter-based memory protection techniques on GPU: To better understand the performance degrading factors, we first look into the SC_128-enabled GPU and examine what causes the performance loss. Figure 4 shows the performance degradation of the SC_128-enabled GPU, compared to the baseline non-secure GPU. In the figure, the first bar (Ctr+MAC) shows the performance of SC_128 with a 16KB counter cache and MAC accesses. The second bar (Ctr+Ideal MAC) shows the performance when MAC accesses are not issued to idealize the MAC implementation, while the counter cache is still modeled. The third bar (Ideal Ctr+MAC) shows the performance when all counter cache accesses are assumed to be hits, but MAC accesses are still issued. All results are normalized to those of the GPU without memory protection.

When the vanilla SC_128 is implemented on GPU without any idealization (Ctr+MAC), the GPU experiences a significant performance loss on the memory-intensive benchmarks, including ges, atax, mvt, bicg, sc, bfs, and srad_v2, from 45.2% for

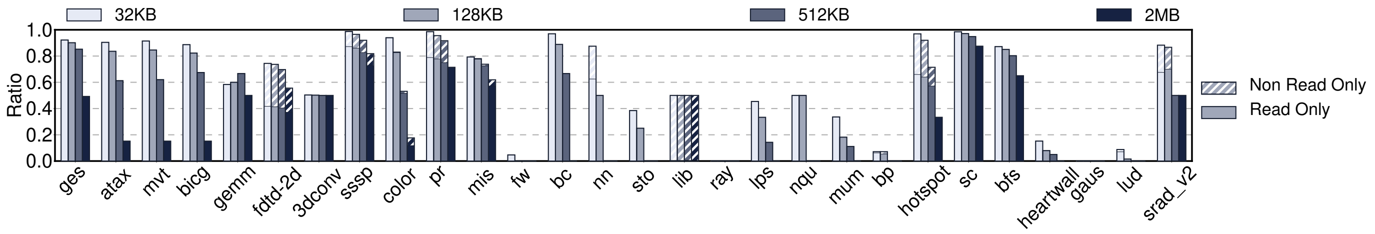


Fig. 6: Ratio of uniformly updated memory chunks over all memory chunks for the GPU benchmarks. The memory chunk size ranges from 32KB to 2MB.

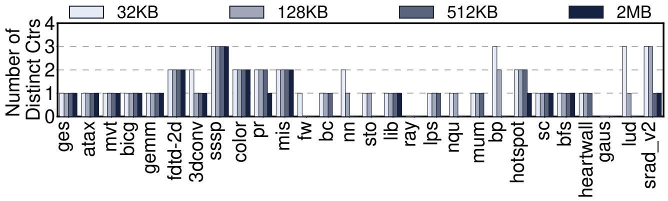


Fig. 7: Number of distinct common counters in the uniformly updated memory chunks for the GPU benchmarks. The chunk size ranges from 32KB to 2MB.

srad_v2 to 77.6% for ges. When only MAC reads and writes are eliminated without the idealization of the counter cache (Ctr+Ideal MAC), the performance improvements are minor over Ctr+MAC. The results indicate that eliminating MAC traffic alone does not effectively reduce the performance degradation as counter cache misses are still on the critical path.

If the GPU never experiences counter cache misses but generates MAC accesses (Ideal Ctr+MAC), we observe a substantial performance improvement for applications with high performance degradation by memory protection (123.9% for ges, 45.8% for atax, 47.1% for mvt, 42.7% for bicg, 51.0% for sc, 90.2% for bfs, and 51.9% for srad_v2). However, there are still significant performance gaps compared to the unsecure GPU. It is because once counter cache misses are reduced, the bandwidth consumption of MAC traffic becomes the next performance bottleneck, calling for the optimization of MAC implementation on top of the improved counter cache technique. A recent prior work, Synergy [39], proposes to embed MAC in the ECC chip of memory, which enables the parallel read of data and MAC, and effectively eliminates the MAC-induced performance overhead. In this paper, we will report the results with the combined architecture of the proposed common counters and Synergy-based MAC improvement.

Figure 5 compares the counter cache miss rates of the three prior studies. Since the counter arity is the same for BMT and SC_128 as 128, their counter cache miss rates are the same, which engenders the similar performance implications. Morphable suffers less from counter cache misses since the arity is 256; however, the performance is still highly influenced as we will show in Section V.

B. Write Characteristics of GPU Applications

Motivated by the observation in Section III-A, the main objective of this paper is to reduce the performance overhead incurred by counter cache misses. To achieve this goal, we leverage a unique characteristic of GPU applications on memory writes and develop an efficient encryption engine for secure GPUs. The observation that GPUs commonly move data in the streaming manner motivated us to identify a unique property of GPU applications, leveraged to design the memory-secured GPU architecture without compromising performance significantly. The GPU applications tend to create memory writes to the allocated memory regions in a considerably uniform manner. Much of the allocated region in a GPU context often receives the same number of writes, either 1) once for initial memory copy from the host, or 2) more-than-once yet an equivalent number of writes if the application writes data, sweeping through the allocated memory.

Methodology: For this analysis, we perform experiments using real GPUs, using NVBit [49] to perform the binary instrumentation of GPU applications and collect memory traces. NVBit allows to capture all the load and store memory accesses and record the accessed virtual addresses. However, NVBit lacks the capability of 1) profiling whether an access misses at the L2 cache, and 2) identifying whether a write-back for a dirty cacheline occurs to serve a cache miss. Therefore, we analyze the write behavior with memory access traces from GPU cores, not with L2 miss traces.

To investigate how much uniformity exists in the memory access counts, we divide the memory space for a context into a set of fixed-size data chunks. For each chunk, we trace how the cachelines in the chunk are updated. If the cachelines in a chunk are updated in a uniform manner, the chunk is denoted as *uniformly updated chunk*. In this section, we analyze the ratios of uniformly updated chunks over the total memory size.

We use two types of scenarios. First, the *GPU benchmark* analysis uses the benchmark suite used in the main evaluation of this paper (Section V). In this section, we run them on a real system, instead of using the simulator. Second, we evaluate seven *real-world applications* for more complex setups.

GPU benchmark analysis: Figure 6 reports the ratios of uniformly updated chunks over all chunks, by varying the chunk size from 32KB to 2MB. Each bar is decomposed into two categories; 1) Read-only (solid) represents the memory chunks updated only by the initial writes from the host, and 2) Non read-only (dashed) shows the chunks with more than one writes for each cacheline in a chunk. On average, when 32KB chunk size is used, 61.6% of all the chunks are uniformly

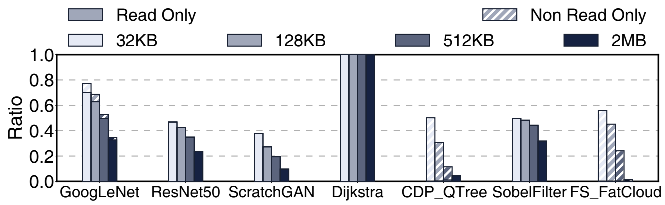


Fig. 8: Ratio of uniformly updated data chunks for the real-world GPU applications.

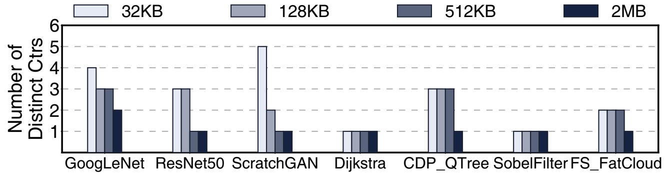


Fig. 9: Number of distinct common counters in the uniformly updated data chunks for the real-world GPU applications.

updated chunks. When using 2MB chunk size, 27.5% of the chunks are uniformly updated. As the chunk size increases, the likelihood of all cachelines in a chunk having the same counter values decreases since the chunk covers more numbers of cachelines and they are likely to diverge during the kernel execution. A large subset of uniformly updated chunks has read-only data. However, for several benchmark applications (fdtd-2d, sssp, pr, lib, hotspot, and srad_v2), a significant portion of the memory chunks is updated more than once.

Figure 7 shows the number of distinct counter values for uniformly updated chunks after the entire execution for the GPU benchmarks. For the benchmark applications only with read-only data, the number of distinct counter values is one, as the uniformly updated chunks are written once. However, for the benchmark with many non-read-only data, the number of distinct counter values can be 2 and 3. Note that the figure shows the number of distinct counter values, and the actual counter values in different benchmarks vary significantly. The results show that if uniformly updated chunks exist, a few common counters can cover the counter values used by the uniformly updated chunks.

Real-world application analysis: To demonstrate the property holds for real-world GPU applications as well, we choose seven full-fledged GPU applications: 1) two DNN inference runs for GoogLeNet [45] and ResNet50 [15], 2) iteration run of DNN training for ScratchGAN [7], 3) Dijkstra [8] algorithm to find the shortest path, 4) conversion from 2D-map to CDP_QTree [41] using CUDA dynamic parallelism [22], 5) SobelFilter [25] algorithm for edge detection, and 6) 3D Fluid simulation for fat cloud (FS_FatCloud).

Figure 8 shows the ratio of uniformly updated data chunks over all data chunks for the real-world applications, as the size of data chunk is increased from 32KB to 2MB. While the real-world applications tend to exhibit lower ratios of uniformly updated data chunks than the GPU benchmarks, a significant portion of data chunks still exhibits the uniform-write property. In GoogLeNet, 34.5% to 84.4% of all chunks are uniformly updated chunks, depending on the chunk size. ResNet-50 and ScratchGAN have more complex model structures than GoogLeNet, and thus have lower ratios of uniformly updated memory chunks. On average, when 32KB chunk size was used, 59.6% of all chunks are uniformly updated. When using 2MB chunk size, 29.3% are uniformly updated. While GoogLeNet, ResNet-50, ScratchGAN, Dijkstra, and SobelFilter are mostly read only, CDP_QTree and FS_Fatcloud are mostly non read-only.

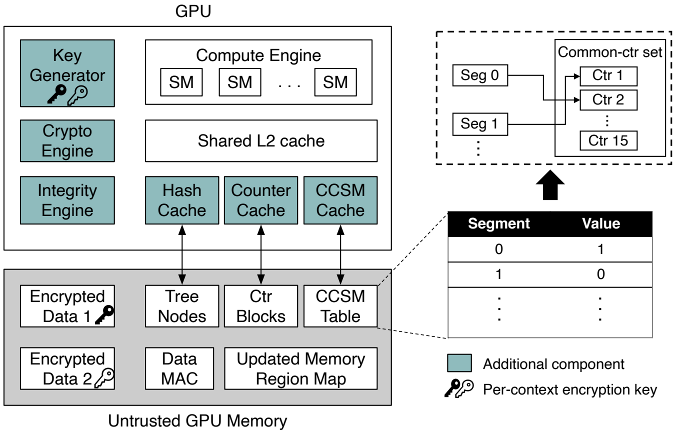


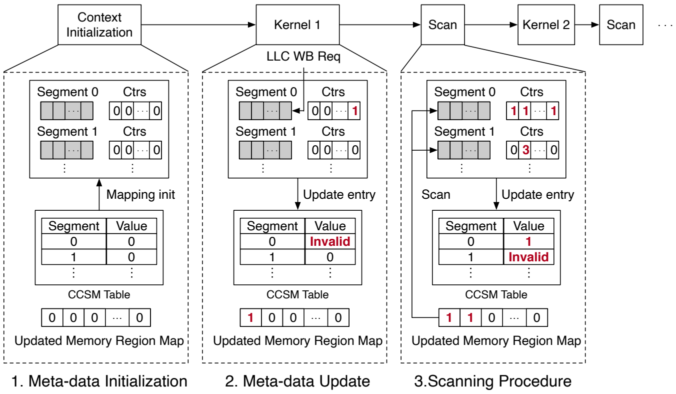
Fig. 10: Proposed memory-secured GPU architecture.

Figure 9 presents the number of distinct counter values for uniformly updated chunks after the program execution for the real-world applications. Compared to the GPU benchmarks, there are more distinct counter values, up-to 5. With the GPU benchmark and real-world application analyses, we conclude that common GPU applications tend to write the memory uniformly and the number of distinct counter values is small, which can be stored in a small amount of on-chip storage. In Section IV, we will introduce an efficient on-chip counter representation that leverages the discussed property and significantly curtails the footprint of counter values needed to reside in the on-chip counter cache.

IV. Architecture

A. Overview

To reduce the performance loss by counter cache misses in GPUs, this paper exploits the uniform memory write behaviors of common GPU applications as discussed in the prior section. The majority of a GPU application memory has one of a few possible counter values, if the counters for the application are reset during the application initialization. The proposed technique, dubbed CommonCounter, provides an efficient mapping of memory address to common counter values shared by many memory chunks. The GPU maintains a few common counters (*common counter set*) for each context. For an LLC miss, the encryption engine can use one of the common counter values for the miss request, if the requested address is for the memory chunk whose counter is one of the common



1. Meta-data Initialization
2. Meta-data Update
3.Scanning Procedure

Fig. 11: Execution flow of the common counter mechanism.

counters. Therefore, if the requested miss can be served by a common counter, it does not access the counter cache.

To support such shared common counters for many memory chunks, several changes are necessary. The necessary hardware modifications for COMMONCOUNTER are delineated in Figure 10. First, each context should have a separate memory encryption key. When the secure GPU command processor allocates new pages for a context, the processor re-encrypts the memory pages with the key and resets the counter values to 0. Second, the GPU hardware maintains a status table, called *Common Counter Status Map (CCSM)*, which tells the encryption engine whether the requested address can be served by a common counter,

Note that the common counter representation complements the existing per-cacheline encryption counters, and thus the per-cacheline counters and counter integrity trees must still be maintained. Common counters simply return the counter value of a memory block with highly compact counter representation, when many pages have the same counter values. Therefore, if the GPU kernel updates a memory block, the corresponding memory chunk can no longer be served with the common counters, since the counter values in the memory chunk diverge from the moment. Therefore, for an LLC eviction with updated data, the original counter value must be updated as well. In addition, the CCSM marks that the memory chunk can no longer use common counters.

Figure 11 depicts the execution flow of the proposed COMMONCOUNTER protection scheme. Common counters are identified in two events. The first event is the initial data transfer from the host memory. When the new data are copied from the host memory, they arrive at GPU in ciphertext encrypted by the shared key between the CPU application and GPU. The GPU decrypts the encrypted data, and the GPU memory are updated with the new data. Many memory pages of a GPU application may not be further updated after the initial update, which we call *initial write once*. After the data transfer, the common counters are identified by scanning actual counter values of the updated memory chunks. In such cases, the counter values are commonly all incremented by one, as the memory copy only updates the memory pages once. Second, when a kernel execution is completed, all the counter values of updated memory pages by the kernel are checked. By scanning counters, the memory chunks with the same counters are identified, and the status map (CCSM) is updated for the next kernel execution accessing the updated memory.

Security guarantee: Using common counters does not affect the confidentiality and integrity of the memory protection. The counter-based encryption and integrity tree maintenance are conducted in the same way as the conventional mechanism. COMMONCOUNTER only finds the common counter values set by the existing mechanism and provides a compressed representation for them. It requires to reset counters for each new context. To avoid the potential leaks by counter resets, a new encryption key is used for each reset.

To support such shared common counters for the majority of memory pages, it requires the following extra meta-data:

- **Per-context encryption key:** When a new GPU application creates the context, a memory encryption key is generated for the new context. The hardware memory encryption engine changes the memory encryption key depending on the context for the addresses of requested data.
- **GPU-wide Common Counter Status Map (CCSM):** For a chunk of memory, the status map tracks whether the memory chunk uses a common counter. For the rest of paper, the unit of the mapping managed by CCSM is denoted as *segment*. Note that the segment size does not need to match the page size for virtual memory support. This mapping table is addressed by physical address. In this paper, we use 128KB segment size, and 4 bits per segment for CCSM. If the value of a CCSM entry is `invalid`, the segment does not use a common counter.
- **Per-context common counter set:** For each context, a set of common counter values are maintained. To minimize the storage overheads, we use only 15 common counters. When the GPU executes a context, the common counter set must be loaded into the on-chip storage for quick accesses. The value of an entry in CCSM is an index to the common counter set for the context.
- **Updated memory region map:** This map records the updated memory region during data transfers from CPU or kernel execution. It is simply to reduce the scanning overheads of counter values after data transfer or kernel execution. We adopt a coarse-grained map, using 1 bit per 2MB region.

B. Trusted GPU Execution

Our trusted GPU model follows the same design as proposed by Graviton [50], although our model assumes vulnerable external DRAM. In the trusted GPU model, the CPU-side user application runs in an SGX enclave, and establishes trust with the GPU. During the initialization, the user application attests the GPU itself by verifying the signature used by the GPU with a remote CA (Certificate Authority). Once the attestation is completed, the user enclave and GPU share a common key.

The common key will be used for the subsequent encrypted communication between the user enclave and GPU. The GPU-side command processor is responsible for conducting the critical management operations, and it is part of TCB (trusted computing base). Part of the GPU memory (*hidden memory*) is reserved for the security meta data, which is visible and accessible exclusively by the secure command processor and crypto engine.

Context initialization: After the initial setup, the user enclave can safely request the creation of a new GPU context. The GPU-side command processor creates and initializes the context. Note that the GPU-side command processor works independently from any privileged CPU software including the operating system. Thus, the operating system cannot maliciously intervene the operation. For memory allocations, the GPU command processor updates the GPU page table of the application. During page table updates, the command processor ensures that different GPU contexts do not share physical pages, enforcing the memory isolation among contexts.

For the proposed memory protection technique, additional steps are necessary. For the context creation, a new memory encryption key is generated. When the context is scheduled to run in the GPU, its memory must be encrypted and decrypted by the key. When a memory page is allocated for the context, the memory is re-encrypted by the new key. However, this initial re-encryption step does not incur any additional cost, since even in the current GPU, the memory contents of newly allocated pages must be scrubbed for security. In our scheme, the scrubbing step generates the same memory writes with zero values encrypted by the hardware engine. In addition, the corresponding CCSM entries for newly allocated pages are reset not to use common counters.

C. Updating Common Counter Status

To enable common counters, what counter values are being commonly used for the memory must be identified. During the step, CCSM and the common counter set must be updated. The command processor initiates the common counter update task for two events: 1) the completion of data transfer from CPU to GPU memory, and 2) the completion of a kernel execution. For the two types of events, the counter values of updated memory segments are scanned. For new common counters found during the step, the common counter set is updated to insert or modify the corresponding entry. In addition, CCSM is updated for the segment to point the new one.

Tracking updated memory region: Since scanning all the counter blocks for the entire physical memory causes significant overheads, the data transfer or kernel execution maintains a coarse-grained *updated memory map* with 1 bit per 2MB region. For 32GB memory, only 16KB memory is used, and its spatial locality is very high since only a small portion of the memory is updated. They are cached in the last-level cache. Although further optimized tracking of updated memory region can be adopted, this study uses this naive approach as it does not cause any significant overheads.

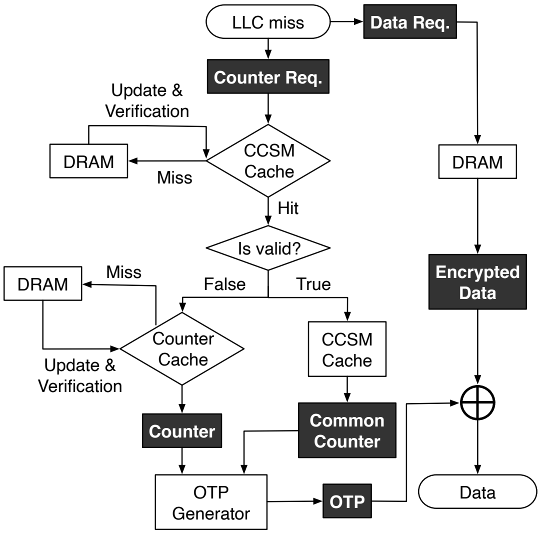


Fig. 12: LLC miss handling flow.

During the scanning step of counter blocks, the updated memory region is first checked. Only if the region has been updated, the corresponding counter blocks are scanned to find common counters. If all counter values are equal in a segment, the corresponding CCSM entry is updated to point to the new common counter value. If the new value does not exist in the common counter set, the new value is added to the set.

D. Using Common Counters

This section describes how common counters are used for LLC misses, and how writes are handled. For fast accesses to CCSM, a small cache is added to the GPU.

CCSM cache: As the common counter status map (CCSM) must cover the entire physical memory of a GPU, the entire map cannot be stored in an on-chip storage. Instead, the proposed technique uses a cache dedicated for the CCSM data. Compared to the counter cache, the efficiency of CCSM cache is higher by orders of magnitude. For counter caches, a 128B counter cache block covers 16KB and 32KB data memory, when 128-ary and 256-ary counter schemes are used respectively. However, a 128B CCSM cache block covers 32MB data memory, which offers 2,048× better caching efficiency than counter blocks with 128-arity. The improvements by the proposed common counters stem from this storage efficiency. In this study, we use a small CCSM cache, of which size is 1KB.

Miss handling flow: Consider Figure 12. For an LLC miss, a data request is sent to the memory controller, and simultaneously, the missed address is first checked in the CCSM cache to identify the CCSM status. If the corresponding entry is not in the CCSM cache, the CCSM entry must be retrieved from the *hidden* memory of the GPU. However, such misses in the CCSM cache are rare due to the high mapping efficiency. Once the CCSM entry is checked, the miss handler knows whether or not it can use a common counter for the address.

If the CCSM entry has a valid index to the common counter set of the context, the common counter value from the set is used, since it is guaranteed that the common counter value is equal to the actual counter value in the counter block. If the CCSM entry marks the corresponding segment as `invalid` (all 1's), the counter cache is accessed to find the corresponding counter entry. If a counter cache miss occurs, it should be handled as the baseline memory encryption engine does. Once the counter value becomes available either from the common counter set or the counter cache, OTP for the miss is generated. When the data arrives from the memory, the data is decrypted with the generated OTP.

Handling writes: We discuss the write flow assuming write-back/write-allocate LLCs, although other write policies can be supported. For a write, two meta data updates are necessary. 1) First, similar to the conventional counter-mode encryption, the corresponding counter entry must be updated, when the dirty cacheline is eventually evicted from the last-level cache to memory. The common counters are a shortcut for counter values, but the actual counter for each memory block is still maintained. 2) Second, the CCSM entry is now updated to `invalid`, so that any subsequent reads/writes to the same address are not allowed to use the common counter, since the counter value is now incremented. Note that the CCSM entry for the updated memory segment will be reset to a common counter, once the kernel execution is completed and the common counter status update procedure finds the uniform counter value for the segment, as discussed in Section IV-C.

E. Hardware Overheads

Meta-data size: CCSM must be allocated in the fixed location of GPU memory, along with other security meta-data. The size of CCSM depends on the size of GPU memory, the number of common counters per context, segment size. For our evaluation, the segment size is set to 128KB, and the number of common counters per context is 15, requiring 4 bits per segment. For each 1GB GPU memory, 4KB of CCSM capacity is necessary.

On-chip storage: A common counter set requires 15×32 bits when the number of common counters per context is 15. If multiple contexts can be running simultaneously in GPUs, multiple common counter sets need to be added in on-chip storage. Unless two contexts are running simultaneously, the common counter set are saved in the context meta-data memory, and restored by the GPU scheduler.

The proposed architecture requires additional on-chip caches including 1KB CCSM cache, 16KB counter cache, and 16KB hash cache. Note that the counter cache and hash cache are also included in the prior memory protection techniques, such as SC_128 and Morphable counters. We use CACTI 6.5 [31] to estimate the area and power overhead of additional on-chip caches. We find that all of on-chip caches amount to an area of $0.11\,\text{mm}^2$, which is 0.02% of the die area of TITAN X Pascal (GP102), with 11.28 mW of leakage power consumption.

TABLE I: Configuration of simulated GPU system.

GPU Core Configuration	
System Overview	28 cores, 32 execution units per core
Shader Core	1417MHz, 32 threads per warp, GTO Scheduler
Cache and Memory Configuration	
Private L1 Cache	48KB, 6-way associative, LRU
Shared L2 Cache	3MB, 16-way associative, LRU
Counter Cache	16KB , 8-way associative, LRU
Hash Cache	16KB, 8-way associative, LRU
CCSM Cache	1KB, 8-way associative, LRU
DRAM	GDDR5X 1251 MHz, 12 channels, 16 banks per rank

TABLE II: Evaluated benchmarks.

Access Patterns	Benchmark Suite	Workload (Abbr.)
Memory Divergent	Polybench	ges, atax, mvt, bicg
	Pannotia	fw, bc
	ISPASS	mum
Memory Coherent	Polybench	gemm, fdtd-2d, 3dconv
	Rodinia	backprop (bp), hotspot, streamcluster (sc), bfs, heartwall, gaussian (gaus), srad_v2, lud
	Pannotia	sssp, pagerank (pr), mis, color
	ISPASS	nn, sto, lib, ray, lps, nqu

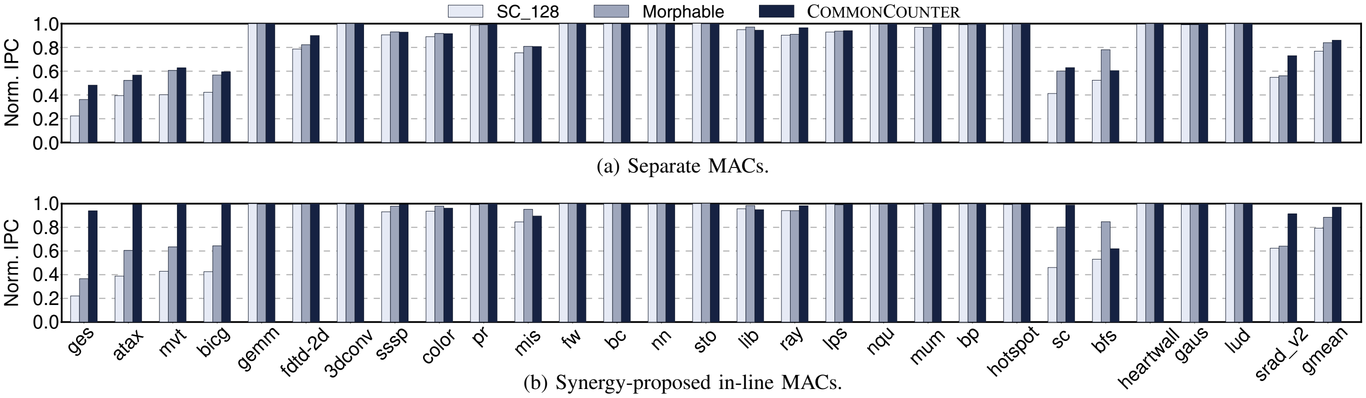
V. Evaluation

A. Methodology

Simulator: We develop the common counter architecture on a GPU simulator, GPGPU-Sim [2]. Table I elaborates the detailed GPU configuration which models NVIDIA TITAN X Pascal (GP102) [18]. The evaluated GPU model is equipped with GDDR5X DRAM, not with 3D stacked memory such as HBM2. We modify the GPGPU-sim simulator to model the memory protection techniques, which consist of encryption/decryption and integrity verification logics. We develop the CommonCounter scheme on top of SC_128 as the baseline system. In addition, we evaluate Morphable counters with a higher arity of 256 counters per cacheline [38].

Workloads: Table II denotes the list of GPU workloads we use for evaluation. We use a wide range of benchmarks, which are provided from the assorted GPU benchmark suites including ISPASS [2], Polybench [13], Rodinia [6], and Pannotia [5]. We evaluate various types of GPU workloads since they have diverse memory access patterns. The column, "Access Patterns", shows the overall memory access patterns of the corresponding applications, either 1) *memory divergent* where the memory accesses of warps are not well coalesced into a smaller number of accesses, and 2) *memory coherent* where the accesses are well coalesced and a fewer number of accesses are requested to the off-chip memory. We run each workload either to completion or up to 1B instructions [21], [48].

Data MAC verification: We consider two different approaches for data MAC verification. The first approach is to bring the data along with its MAC from the memory, which creates more pressure on the off-chip memory bandwidth. The



(a) Separate MACs.

(b) Synergy-proposed in-line MACs.

Fig. 13: Performance of the two existing memory protection schemes (SC_128 and Morphable) and the proposed CommonCounter technique with 16KB counter cache. Performance is normalized to GPU without memory protection. We evaluate the performance with two different assumptions with respect to the data MAC verification approaches.

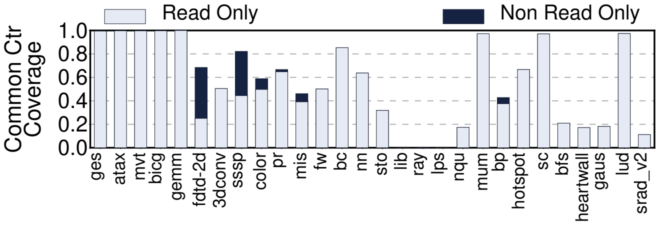


Fig. 14: Ratio of LLC misses served by common counters.

second approach is the technique proposed in an inspiring prior work, Synergy [39], which proposes to use the ECC chip as a carrier for the MAC and virtually offer the complete elimination of MAC-induced performance overhead. CommonCounter is not designed for the MAC-induced overhead reduction, which makes the use of Synergy more favorable to our technique. We report the results for both cases.

B. Experimental Results

Performance improvement: Figure 13 shows the normalized IPC of three different memory protection schemes: (1) Split counter technique (SC_128) [53], (2) Morphable Counter (Morphable) [38], and (3) the CommonCounter technique (CommonCounter). Figure 13 (a) provides the results for the case where both data and MAC are retrieved from off-chip memory, while Figure 13 (b) shows the results when the MAC read is inlined with the ECC read as Synergy suggested.

The gap between Figure 13 (a) and (b) represents the performance loss incurred by the data MAC verification overhead, when the MAC is separately read from memory. CommonCounter reduces performance degradation for secure memory in both cases, but the performance improvements of CommonCounter over SC_128 and Morphable are higher with the Synergy MAC design. Compared to the unsecure baseline, CommonCounter reduces the performance degradation from 13.9% in (a) to 2.9% in (b) on average. The non-negligible jump from (a) to (b) suggests the integration of our CommonCounter technique with the efficient data MAC verification techniques such as Synergy would potentially offer significant performance improvement. As discussed in Section III-A, it shows the need for the reduction of both counter cache misses and MAC traffic to improve overall performance.

In Figure 13 (b), we see the SC_128 and Morphable techniques exhibit on average, 20.7%, 11.5% performance degradations, while CommonCounter incurs only 2.9% performance overhead. The significant reduction comes from the fact that the proposed cache subsystem effectively serves the per-cacheline counter requests using the common counters. The benchmark applications with high performance degradation (ges, atax, mvt, bicg, sc, and srad_v2) are improved signficantly. The performance improvements range from 46.4% and 42.4% for srad_v2 over SC_128 and Morphable respectively, to 326.2% and 156.4% for ges. Although CommonCounter provides better performance than SC_128 and Morphable for most of the workloads, CommonCounter exhibits lower performance than Morphable for lib and bfs. For the two applications, there are many LLC misses not served by common counters due to their update patterns, and the 256-arity counter block of Morphable provides better performance than the 128-arity block used for CommonCounter. However, CommonCounter can be improved by adding common counters on top of Morphable, increasing the base arity of its counter block.

The performance improvement discussed above can be attributed to the effective use of common counters. Figure 14 shows the ratio of off-chip memory accesses served by common counters over all counter requests. Each bar is decomposed into read-only data (light grey) and non-read-only data (dark grey). We notice that for benchmarks that see large performance benefits in Figure 13 (i.e., ges, atax, mvt, bicg, and sc), the memory access coverage by common counters is close to 100%, which shows the correlation between the effectiveness of common counters and the performance gains.

Sensitivity to counter cache size: Figure 15 shows how the performance varies as the counter cache size changes. In general, CommonCounter is less influenced by the counter cache size change compared to SC_128, since many LLC miss

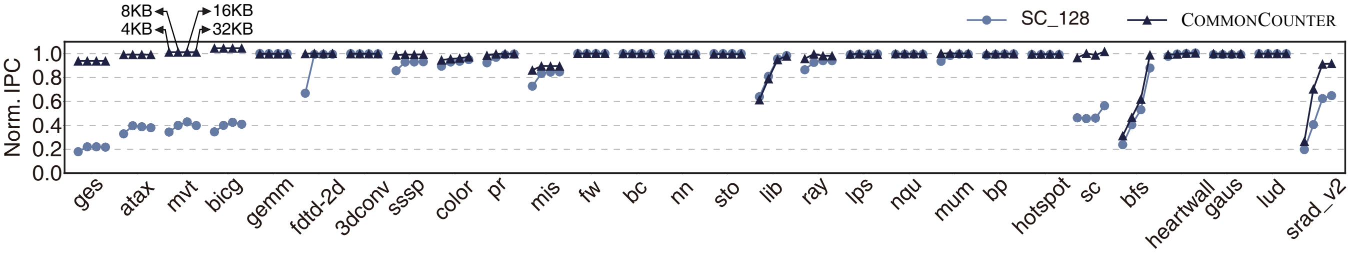


Fig. 15: Performance variations as the counter cache size ranges from 4KB to 32KB. The baseline is non-secure vanilla GPU. This result is based on the configuration that employs Synergy for data MAC verification.

TABLE III: Scanning overhead.

Workload	# of Executed Kernels	Total Scan Size	Ratio*
3dconv	254	32,256 MB	0.372 %
gemm	1	32 MB	0.090 %
bfs	24	4,108 MB	0.004 %
bp	2	390 MB	0.372 %
color	28	5,650 MB	0.081 %
fw	255	2,040 MB	0.114 %

*Scanning overhead over total kernel execution time

requests are served by the common counters, even though the associated counters are unavailable in the counter cache. For instance, when CommonCounter is used, sc experiences almost no performance loss in comparison with the non-secure GPU, as the counter cache size decreases. In contrary, in case of SC_128, sc exhibits 43.6% degradation when the counter cache size is 32KB, and the loss drastically increases up to 53.7% when the size is shrunk to 4KB. However, we observe that for some benchmarks, CommonCounter is also sensitive to the change of counter cache size. For instance, lib experiences significant performance losses when the counter cache size decreases, which is because this benchmark has very few opportunities to use common counters, as shown in Figure 14.

Scanning overhead: To identify the segments capable of using common counters instead of the regular counters, the proposed mechanism needs to scan the updated memory regions at the boundary of every kernel execution. If the scanning overhead is significant, the performance benefits that come from the use of common counters would diminish. To roughly estimate the scanning overhead, we run simulations to identify the size of updated memory regions. For the identified updated memory sizes, we measure memory region scanning latencies by using a real machine equipped with a GTX 1080. Table III shows the number of executed kernels during the end-to-end application run, the total size of updated memory regions that are subject to be scanned, and the ratio of scanning overhead out of the total execution time. The scanning overhead ratio ranges from 0.372% (3dconv) down to 0.004% (bfs), which shows that the scanning overhead is virtually negligible. Note that we incorporated the reported scanning overhead in all the performance experiments.

VI. Discussion

Integrated GPUs: Although this paper investigated a discrete GPU with its own untrusted memory, CommonCounter can be extended to integrated GPU models with moderate changes. In the integrated GPU model, since CPU cores and a GPU share the memory through shared memory controllers, the CPU cores and GPU can share not only the protected memory, but also the memory encryption and integrity protection engine. To enable common counters for such an integrated GPU model, a separate encryption key is needed for each context individually for CPU and GPU. In addition, the counters for each context are separately managed, being reset in the initialization of a context, unlike the common memory encryption and counter managements in current secure processors. Such an approach was proposed by Rogers et. al [37] to integrate the hardware memory encryption with virtual memory support. We plan to investigate the memory protection design of integrated GPUs as our future work.

Overhead for secure CPU-GPU communication: For secure communication between CPU and GPU, data transfers must be encrypted. Prior work suggested that memory copy and authentication can be effectively parallelized, which minimizes the overhead [19], [50]. Additionally, Ghosh et al. proposed a hardware acceleration technique that significantly lowers the encryption and decryption overhead [12]. In this work, the overhead of encrypted communication is not evaluated, but with HW-based acceleration for encryption [12], the overhead for the CPU-GPU communication is expected to be small compared to the kernel execution time.

Support for multiple GPUs with shared memory: If multiple GPUs running a single application context are sharing their memory, the GPUs must share the same encryption key for the context. In addition, the coherence of counter caches must be maintained as different GPUs can potentially write to shared data, updating the corresponding counters. Although such supports are also required for any multi-chip secure processors with protected shared memory, investigating it on GPUs is our future work.

Concurrent kernel execution: Supporting concurrent kernel execution is possible with common counters. Assuming that GPUs provide context isolation with virtual memory, each context needs to have a separate encryption key with common counters. The hardware-based mechanism with CCSM and update-scanning finds common counters without considering contexts as they are based on physical addresses, unaffected by multi-kernel execution.

VII. Related Work

Secure processors: Prior to the commercial trusted execution support, there had been extensive studies for hardware-based secure processor designs [4], [28], [43], [47]. XOM investigated the trusted execution in remote client systems with hardware-based encryption and integrity protection [47]. AEGIS also proposed an architecture for tamper-evident and tamper-resistant execution environments with or without the trusted kernel [43]. It proposed an effective defense against replay attacks. Secret-Protected (SP) design proposed the hardware-based isolation for a security critical module of user applications [28]. Bastion investigated multi-domain isolation with hardware-based memory protection [4]. For cloud systems running virtual machines, there have seen several studies to isolate virtual machines with hardware-based mechanism, defending user virtual machines from potentially malicious hypervisors [20], [44], [51].

Memory protection: For the confidentiality and integrity protection of memory, there have been significant improvements in the past two decades. All the hardware-based fine-grained integrity protection schemes resilient from replay attacks employ a variant of Merkle trees [30]. However, since the integrity protection is the major source of performance degradation for memory protection, there have been extensive studies to mitigate the overheads. Suh et al. proposed a log-hash integrity verification technique to reduce the bandwidth consumption for Merkle tree and One-Time-Pad (OTP) encryption to mitigate the decryption overheads [42]. Gassend et al. proposed a caching scheme for intermediate tree nodes [11]. To reduce counter access latencies, a separate counter cache structure was investigated for OTP encryption [54]. Shi et al. investigated a prediction scheme for the counter number to hide counter access latencies [40]. Yan et al. proposed split counters with major and minor counters to increase the arity of integrity tree nodes [53]. Bonsai Merkle Tree (BMT) significantly reduced the size of the integrity tree by constructing the counter integrity tree [37]. Lee et al. proposed a memory scheduling and type-aware cache insertion schemes for multi-cores with protected memory [27].

Recent studies further reduces the overheads of counter cache misses and integrity tree management. VAULT uses a different arity in each level of the counter integrity tree [46]. Morphable counters provides a compact counter representation packing more counters per counter node with changeable bit arrangements for counters [38].

GPU security: Several studies analyzed the vulnerabilities of GPU computation. CUDA Leaks [36] and Zhu et al [56] investigated potential leaks through various architectural resources. Several studies showed the potential problems of the GPU computation through residual information in deallocated memory pages in GPU [14], [26], [29], [55].

Recent studies analyzed the side-channel attacks on the GPU [16], [23], [24], [32], [52] Naghibijouybari et al. [32] demonstrated the vulnerabilities of covert and side-channel attack on the GPU. Kadam et al. proposed a randomized logic of coalescing unit [23] and optimized the coalescing mechanism to mitigate correlation timing attack on GPU [24]. Xu et al. proposed a decision tree based detection and hierarchical defense mechanism to prevent contention-based covert and side-channel attacks [52]. Hunt et al. proposed oblivious CPU-GPU communication protocol to mitigate side-channel attacks in GPU TEEs [16].

VIII. Conclusion

This study investigated the performance implication of supporting the hardware-based memory protection for GPU computation. Based on the analysis of uniform write characteristics of GPU applications, the paper proposed to use common counters, almost entirely eliminating counter cache misses. Exploiting the GPU-specific memory behavior, this study showed that the hardware-based memory protection is feasible for high performance GPU computation.

IX. ACKNOWLEDGMENT

This work was supported by National Research Foundation of Korea (NRF-2019R1A2B5B01069816) and the Institute for Information & communications Technology Promotion (IITP2017-0-00466). Both grants are funded by the Ministry of Science and ICT, Korea. This work was also partly supported by Samsung Electronics Co., Ltd. (IO201209-07864-01).

References

[1] ARM Limited, “Security Technology Building a Secure System using TrustZone Technology (White Paper),” 2009.

[2] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, “Analyzing CUDA Workloads using a Detailed GPU Simulator,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[3] D. Boggs, G. Brown, N. Tuck, and K. Venkatraman, “Denver: Nvidia’s First 64-bit ARM processor,” *IEEE Micro*, 2015.

[4] D. Champagne and R. B. Lee, “Scalable Architectural Support for Trusted Software,” in *International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[5] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, “Pannotia: Understanding Irregular GPGPU Graph Applications,” in *International Symposium on Workload Characterization (IISWC)*, 2013.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A Benchmark Suite for Heterogeneous Computing,” in *International Symposium on Workload Characterization (IISWC)*, 2009.

[7] C. de Masson d’Autume, S. Mohamed, M. Rosca, and J. Rae, “Training Language GANs from Scratch,” in *Neural Information Processing Systems (NeuroIPS)*, 2019.

[8] E. W. Dijkstra *et al.*, “A Note on Two Problems in Connexion With Graphs,” *Numerische mathematik*, 1959.

[9] M. Ditty, A. Karandikar, and D. Reed, “Nvidia’s Xavier SoC,” in *Hot Chips: A Symposium on High Performance Chips*, 2018.

[10] D. Franklin, “NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge,” *NVIDIA Accelerated Computing—Parallel Forall*, 2017.

[11] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, “Caches and Hash Trees for Efficient Memory Integrity Verification,” in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2003.

[12] S. Ghosh, L. S. Kida, S. J. Desai, and R. Lal, “A $>$ 100 Gbps Inline AES-GCM Hardware Engine and Protected DMA Transfers between SGX Enclave and FPGA Accelerator Device,” *IACR Cryptol. ePrint Arch.*, 2020.

[13] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, “Auto-Tuning a High-Level Language Targeted to GPU Codes,” in *Innovative Parallel Computing (InPar)*, 2012.

[14] A. B. Hayes, L. Li, M. Hedayati, J. He, E. Z. Zhang, and K. Shen, “GPU Taint Tracking,” in *USENIX Annual Technical Conference (ATC)*, 2017.

[15] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in *Computer Vision and Pattern Recognition (CVPR)*, 2016.

[16] T. Hunt, Z. Jia, V. Miller, A. Szekely, Y. Hu, C. J. Rossbach, and E. Witchel, “Telekine: Secure Computing with Cloud GPUs,” in *USENIX Symposium on Networked Systems Design and Implementation (NSDI))*, 2020.

[17] Intel Corp, “Intel(R) Software Guard Extensions for Linux* OS, linux-sgx,” 2016. [Online]. Available: https://github.com/intel/linux-sgx

[18] A. Jain, M. Khairy, and T. G. Rogers, “A Quantitative Evaluation of Contemporary GPU Simulation Methodology,” *ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)*, 2018.

[19] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, “Heterogeneous Isolated Execution for Commodity GPUs,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[20] S. Jin, J. Ahn, S. Cha, and J. Huh, “Architectural Support for Secure Virtualization under a Vulnerable Hypervisor,” in *International Symposium on Microarchitecture (MICRO)*, 2011.

[21] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, “Orchestrated Scheduling and Prefetching for GPGPUs,” in *International Symposium on Computer Architecture (ISCA)*, 2013.

[22] S. Jones, “Introduction to Dynamic Parallelism,” in *GPU Technology Conference Presentation S*, 2012.

[23] G. Kadam, D. Zhang, and A. Jog, “RCoal: Mitigating GPU Timing Attack via Subwarp-based Randomized Coalescing Techniques,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[24] G. Kadam, D. Zhang, and A. Jog, “BCoal: Bucketing-based Memory Coalescing for Efficient and Secure GPUs,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[25] N. Kanopoulos, N. Vasanthavada, and R. L. Baker, “Design of an Image Edge Detection Filter Using the Sobel Operator,” *IEEE Journal of solid-state circuits*, 1988.

[26] M. Kerrisk, “XDC2012: Graphics Stack Security,” 2012. [Online]. Available: https://lwn.net/Articles/517375/

[27] J. Lee, T. Kim, and J. Huh, “Reducing the memory bandwidth overheads of hardware security support for multi-core processors,” *IEEE Transactions on Computers (TC)*, vol. 65, no. 11, 2016.

[28] R. B. Lee, P. C. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, “Architecture for Protecting Critical Secrets in Microprocessors,” in *International Symposium on Computer Architecture (ISCA)*, 2005.

[29] S. Lee, Y. Kim, J. Kim, and J. Kim, “Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities,” in *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[30] R. C. Moerkle, “Protocols for Public Key Cryptosystems,” in *IEEE Symposium on Security and privacy (S&P)*, 1980.

[31] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, “CACTI 6.0: A tool to model Large Caches,” *HP laboratories*, 2009.

[32] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, “Rendered Insecure: GPU Side Channel Attacks are practical,” in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[33] NVIDIA Corp, “NVIDIA GeForce GTX 1080,” 2017. [Online]. Available: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf

[34] NVIDIA Corp, “NVIDIA Turing Architecture,” 2018. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[35] NVIDIA Corp, “NVIDIA AMPERE GA102 GPU ARCHITECTURE,” 2020. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf

[36] R. D. Pietro, F. Lombardi, and A. Villani, “CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix,” *ACM Transactions on Embedded Computing Systems (TECS)*, 2016.

[37] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, “Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly,” in *International Symposium on Microarchitecture (MICRO)*, 2007.

[38] 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 *International Symposium on Microarchitecture (MICRO)*, 2018.

[39] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, “Synergy: Rethinking Secure-Memory Design for Error-Correcting Memories,” in *International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[40] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, “High Efficiency Counter Mode Security Architecture via Prediction and Precomputation,” in *International Symposium on Computer Architecture (ISCA)*, 2005.

[41] J. Smith and S.-F. Chang, “Quad-Tree Segmentation for Texture-based Image Query,” in *ACM international conference on Multimedia*, 1994.

[42] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, “Efficient Memory Integrity Verification and Encryption for Secure Processors,” in *International Symposium on Microarchitecture (MICRO)*, 2003.

[43] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, “AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing,” in *International Conference on Supercomputing (ICS)*, 2003.

[44] J. Szefer and R. B. Lee, “Architectural Support for Hypervisor-Secure Virtualization,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[45] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going Deeper with Convolutions,” in *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[46] M. Taassori, A. Shafiee, and R. Balasubramonian, “VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[47] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, “Architectural Support for Copy and Tamper Resistant Software,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[48] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu, “The locality descriptor: A holistic cross-layer abstraction to express data locality in gpus,” in *International Symposium on Computer Architecture (ISCA)*. IEEE, 2018.

[49] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, “NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs,” in *International Symposium on Microarchitecture (MICRO)*, 2019.

[50] S. Volos, K. Vaswani, and R. Bruno, “Graviton: Trusted Execution Environments on GPUs,” in *Operating Systems Design and Implementation (OSDI)*, 2018.

[51] Y. Xia, Y. Liu, and H. Chen, “Architecture Support for Guest-transparent VM Protection from Untrusted Hypervisor and Physical Attacks,” in *International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[52] Q. Xu, H. Naghibijouybari, S. Wang, N. Abu-Ghazaleh, and M. Annavaram, “GPUGuard: Mitigating Contention based Side and Covert Channel Attacks on GPUs,” in *ACM International Conference on Supercomputing (ICS)*, 2019.

[53] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, “Improving Cost, Performance, and Security of Memory Encryption and Authentication,” in *International Symposium on Computer Architecture (ISCA)*, 2006.

[54] J. Yang, Y. Zhang, and L. Gao, “Fast Secure Processor for Inhibiting Software Piracy and Tampering,” in *International Symposium on Microarchitecture (MICRO)*, 2003.

[55] Z. Zhou, W. Diao, X. Liu, Z. Li, K. Zhang, and R. Liu, “Vulnerable GPU Memory Management: Towards Recovering Raw Data from GPU,” *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017.

[56] Z. Zhu, S. Kim, Y. Rozhanski, Y. Hu, E. Witchel, and M. Silberstein, “Understanding the Security of Discrete GPUs,” in *General Purpose GPUs (GPGPU)*, 2017.