2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10071100

Plutus: Bandwidth-Efficient Memory Security for GPUs

Rahaf Abdullah
North Carolina State University
Raleigh, USA
rmabdul2@ncsu.edu

Huiyang Zhou
North Carolina State University
Raleigh, USA
hzhou@ncsu.edu

Amro Awad
North Carolina State University
Raleigh, USA
ajawad@ncsu.edu

***Abstract*—Graphic-Processing Units (GPUs) are increasingly used in systems where security is a critical design requirement. Such systems include cloud computing, safety-critical systems, and edge devices, where sensitive data is processed or/and generated. Thus, the ability to reduce the attack surface while achieving high performance is of utmost importance. However, adding security features to GPUs comes at the expense of high-performance overheads due to the extra memory bandwidth required to handle security metadata. In particular, memory authentication metadata (e.g., authentication tags) along with encryption counters can lead to significant performance overheads due to the memory bandwidth used to fetch the metadata. Such metadata can lead to more than 200% extra bandwidth usage for irregular access patterns.**

In this work, we propose a novel design, Plutus, which enables low-overhead secure GPU memory. Plutus has three key ideas. The first is to leverage value locality to reduce authentication metadata. Our observation is that a large percentage of memory accesses could be verified without the need to bring the authentication tags. Specifically, through comparing decrypted blocks against known/verified values, we can with high confidence guarantee that no tampering occurred. Our analysis shows that the probability of the decryption of a tampered (and/or replayed) block leading to a known value is extremely low, in fact, lower than the collision probability in the most secure hash functions. Second, based on the observation that many GPU workloads have limited numbers of dirty block evictions, Plutus proposes a second layer of compact counters to reduce the memory traffic due to both the encryption counters and integrity tree. Third, by exploring the interesting tradeoff between the integrity tree organization vs. metadata fetch granularity, Plutus uses smaller block sizes for security metadata caches to optimize the number of security metadata memory requests. Based on our evaluation, Plutus can improve the GPU throughput by 16.86% (up to 58.38%) and reduce the memory bandwidth usage of secure memory by 48.14% (up to 80.30%).

I. Introduction

GPUs can significantly accelerate workloads that demand high throughput such as machine learning [21], graph analytics [24], and scientific computing [29]. The significant acceleration power of GPUs has driven their wide adoption by different cloud service providers. While the users' applications are offloaded to remote untrusted clouds, ensuring the security of the execution and data, through confidential computing, becomes a necessity. Because users no longer have control over the executing environment, physical attacks to leak and/or tamper with data become even more plausible. In fact, even within the same GPU board, it is common to integrate memory stacks and modules developed by a third-party vendor. Thus, GPU vendors need to avoid memory vendor lock-in, by ensuring memory security support from the GPU chip side, hence protecting their reputation while integrating the cheapest and/or the highest-performance options. Accordingly, supporting GPUs with a *trusted execution environment (TEE)* is crucial in this era. To create an isolated execution environment, three components are provided: encryption keys that are hardware-managed, remote attestation, and memory security.

TEEs for CPUs, such as Intel Software Guard Extension (SGX) [16] and ARM TrustZone, constrain the trusted computing base (TCB) to the processor chip only, while any other components located off the processor chip are considered untrusted. As a result, data protection is required for any data flows off-chip. For GPU TEEs, there have also been recent proposals such as Graviton [32], HIX [10], and others [9], [18], [35], [36]. However, memory security relies on metadata stored in memory, which leads to high bandwidth consumption. These bandwidth overheads of secure GPU memory could become an obstacle to enabling secure execution while ensuring high performance.

Memory security in CPUs provides three main guarantees: ***confidentiality***, ***integrity*** and ***replay prevention*** [7], [27], [34]. Confidentiality through data encryption protects data privacy by cryptographically hiding it from any external entity, e.g., physical access to the memory by any entity except the processor chip provisioned with the encryption key. Data integrity prevents active attacks that introduce changes to the original data. While message authentication code (MAC) satisfies this goal, it fails to ensure the freshness of data, which makes the system vulnerable to replay attacks. Accordingly, the freshness of data is typically guaranteed through integrity trees, such as Merkle Trees (MT) or Tree of Counters (ToC). Adopting these from CPUs directly to GPUs results in significant performance losses due to architectural differences between GPUs and CPUs [36], [37]. Hence, many prior works exploited the distinct behaviors and characteristics of either the GPU architecture itself [36] or the nature of applications running on GPUs [18], [35], to organize security metadata in better ways that reduce security overheads and leave potential for speedup.

Graviton [32] and HIX [10] trust the off-chip memory

and do not support any memory security model. However, subsequent works emphasize the necessity of securing the GPU memory. Na et al. [18] show that most of the memory accesses in GPUs conform to patterns that render a number of contiguous blocks in memory to have the same encryption counter value. Based on their observation, they maintain a common set of counters in the GPU boundary to reduce counter traffic, which is consequentially reflected into reduced Bonsai Merkle Tree (BMT) traffic. Yuan et al. [36] observe that using the physical or virtual address directly to address security metadata blocks leads to useless traffic for fetching metadata corresponding to data in other memory partitions. Accordingly, they propose using local partition addresses for metadata organization. Additionally, they recommend using sector granularity as the protection granularity for metadata to eliminate any extra data traffic for the sake of security checks. Adaptive security support for Heterogeneous Memory on GPUs [35] exploits the mostly read-only data of GPU benchmarks, to skip the need for fetching counters and performing integrity verification in case of a counter miss. In summary, most of the prior works focus on optimizing counters traffic due to its significance and direct effect on BMT traffic too, but they track counters in a coarse granularity which leads to missed optimization opportunities. In addition, MAC traffic still consumes significant bandwidth, especially for applications with poor spatial and temporal localities. In this work, we propose a novel scheme, Plutus, to address these critical challenges.

Plutus builds upon three key ideas. ① Unlike CPUs, additional latency in memory accesses is more tolerable than increased bandwidth. However, we observe that certain encryption schemes, e.g., AES-XTS, provide more resistance to malleability; for each cipher block, e.g. 128-bit, produced by one run of the encryption algorithm, any bit flip in this cipher block will lead to a completely unrelated corresponding part in the plaintext. Note that this is the inverse feature of the avalanche effect desired upon encryption (one-bit change in plaintext leads to many-bit changes in ciphertext). Accordingly, by leveraging the malleability resistance of AES-XTS, we can guarantee no tampering with high probability by comparing the decryption result with recently verified/known values. Since it is malleability resistant, tampering results in a known/correct value is extremely unlikely; in fact, we prove that its probability is less than finding collisions in secure MACs. Thus, by counter-intuitively checking if not been tampered with, through checking if matches any recently verified value, we can verify authenticity without fetching MAC. This is in contrast with checking if tampered with by computing MAC. ② We observe that for GPU workloads, the neighboring encryption counter values are both too small and close to each other. Thus, Plutus employs a compact two-level counters scheme to reduce the counter and BMT traffic. Specifically, Plutus uses small counters along with a small BMT to reduce the counter traffic. In the infrequent cases when the small counters overflow, the original counters and BMT are consulted. ③ Plutus explores the tradeoff between the integrity tree organization and metadata fetch granularity to optimize the metadata traffic. In GPUs, a finer granularity than a cache block is generally used to access the main memory. Specifically, *sectors* which are 32B typically can be independently read/written to the memory module, even though a full 128B block is reserved in the cache.

To the best of our knowledge, none of the prior works explored eliminating MAC fetches even though they showed it contributes to a large portion of the overheads [8], [18], [31]. Similarly, even though the common counter scheme [18] exploited the spatial similarity of counters, none explored how the small values of counters can be used to build a first-level integrity verification using a very small tree. Prior works, e.g., PSSM [36], studied the trade-offs of whether to maintain security metadata (e.g., MAC and counter) per sector, and hence avoiding over-fetching but incurring high-storage and bandwidth overheads, or maintaining the metadata per block and hence simply disabling sectored caches. PSSM ignored the harmony between such granularity and the integrity tree verification. We observe that using block granularity to fetch security metadata reduces the integrity tree depth and hence reduces the number of integrity tree misses, but can over-fetch metadata (i.e., a 128B integrity tree node block) to allow verification from the next level. This striking tension between metadata fetch granularity (smaller is better) and the tree depth (shallower is better) requires careful examination. To the best of our knowledge, this is the first paper to study the integrity tree organization vs. metadata fetch granularity trade-offs in the context of GPUs. To evaluate Plutus, we use GPGPU-Sim v4.0 [13] with a diverse set of workloads from the Rodinia-3.1 [4], Parboil [26], Lonestargpu-2.0 [14] and Pannotia [3] benchmark suites. Our evaluation shows that Plutus improves the GPU throughput by 16.86% (up to 58.38%) and reduces the memory bandwidth usage of secure memory by 48.14% (up to 80.30%), compared to PSSM.

The rest of the paper is organized as the following. First, the background concepts are discussed in Section II. Second, in Section III, the motivation behind Plutus is explained. Then, the design and evaluation results are discussed in Section IV and Section V, respectively. Section VI summarizes the related prior work. Finally, we conclude the paper in Section VII.

II. Background

A. Memory Security

Memory security aims to attain different types of guarantees: ***confidentiality***, ***integrity***, and/or ***replay protection***. confidentiality is achieved through encryption while integrity usually is protected through message authentication codes (MAC), whereas replay prevention can be added through integrity trees over data or some of the security metadata [20]. Since processor memory is typically accessed using cache-line size (e.g., 64 bytes), most memory security implementations operate on that granularity; decrypt/encrypt a memory block, and verify/protect its integrity.

1) Memory Encryption: Memory encryption limits the plain data visibility to the trusted computing base (TCB) only and hence makes it unintelligible for physical attacks or bus snooping attacks. Two encryption approaches can be used for this purpose: ***direct encryption*** or ***counter-mode encryption***. In direct encryption, the encryption algorithm, e.g. AES, is directly applied to the data, which introduces the encryption/decryption latency to the critical path for read/write operations. This affects the performance negatively in CPUs while GPUs are latency-tolerant due to their massive thread-level parallelism, they can hide the latency by serving different warps as shown in prior works [37]. However, if no extra tweaks are included in the encryption process, this opens the door for dictionary-based attacks. Thus, more secure modes allow using a tweak as a part of the encryption process. However, the tweaks can be used differently, for instance in the commonly used counter-mode encryption (CME) mode, shown in Figure 1, an encryption pad is generated by encrypting the tweak, and the pad is consequently used to complete encryption/decryption using bitwise XOR operation. Such a solution offers high performance as it can hide the latency of pad generation with fetching the ciphertext. However, since CME relies on XOR'ing a pad with the plaintext to generate the ciphertext, it suffers from ***malleability***. In other words, flipping certain bits in ciphertext would lead to flipping the exact same bits in the plaintext upon decryption. On the other hand, another option commonly used with storage, where encryption latency is negligible, is the Tweakable block-cipher with ciphertext stealing (XTS) mode. In XTS mode, a tweak is used, however, the plaintext is also fed to the AES engines rather than merely XORed with a pad. Accordingly, the XTS malleability is merely at the cipher block size (typically 16B) of the 64B, rather than at bit resolution as in CME. Whether XTS or CME is used, tweaks are generally implemented as a combination of address (for spatial uniqueness) and counters (for temporal uniqueness).

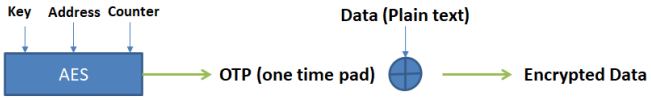

Fig. 1. Counter-Mode Encryption

Counters are generally grouped into memory blocks that could be cached in the processor chip to capture spatial and temporal locality [27], [33], [34]. Counters could be organized in two different ways: monolithic counters as used in Intel SGX [16] [7], where 56-bit counters, one per cache block, are grouped into groups of eight, in case of 64-byte cache blocks. The state-of-the-art counters organization is split-counters [33], where each counter composes a minor counter per data block, and a major counter is shared among a group of data blocks.

2) Message Authentication Code (MAC): Memory integrity means ensuring that the message has not been altered or tampered with in transit to the processor or while residing in memory. In other words, the read data is identical to the most recent one written by the processor. This is partially attained by MACs, which verify that the data has been produced by the processor. In other words, it has not been tampered with. MACs can detect spoofing and splicing attacks. However, MAC values alone do not protect against replay attacks, therefore, integrity trees are usually used for this purpose [7], [17], [20]. MAC is a cryptographic hash calculated over certain data with the use of a secret key [30]. The algorithm used for the generation is one-direction, which makes it impossible to get the actual message over which this MAC has been computed even with the availability of the secret key. The output, which is the MAC, is a fixed-size value. As the size of the MAC increases, the MAC collision rate decreases and hence having a higher level of security. In Intel SGX for CPU trusted execution environment (TEE), 56-bit MAC is used [7]. While in the partitioned sectored security metadata work (PSSM) [36], a 32-bit MAC is used to mitigate MAC storage overheads, especially after assigning a MAC per smaller granularity (sectors).

3) Integrity Trees: are used for preventing replay attacks. In general, there are two variations of integrity trees: Merkle Tree (MT) [17] and Tree of Counters (ToC) [16], also known as a parallelizable integrity tree. Both are shown in Figures 2 and 3. MTs protect the integrity of the memory data by a tree of MACs/hashes that ends with one node called the root, which is securely kept in the processor chip and never leaves it. On each memory access, a path of MAC/hash values up to the root is verified, and since the root never leaves the processor boundary, it reflects the most recent state of the memory and any tampering will be detected. Originally, MT is constructed over the data itself leading to a relatively large tree. Another version of MT is called Bonsai Merkle Tree (BMT) [20], where the leaves of the tree are the encryption counters. In addition, the encryption counters are used as tweaks for the MAC generation as part of replay attack prevention. ToCs use encryption counters as their leaves too. Unlike MT, ToC uses other counters, called versions, in the upper levels of the tree instead of hashes/MACs. Each node contains a MAC value calculated over its data and its parent version counter. Hence, no cumulative calculations are done which allows parallel updates and verification.

To reduce the performance overheads associated with the tree traversal, both of the update and verification processes stop at the first node that hits in the cache, which is already verified. For updates, when those nodes get evicted from the cache, they propagate the update to their parents. This is referred to as the lazy update scheme while the updates that go always to the root are part of the eager update scheme.

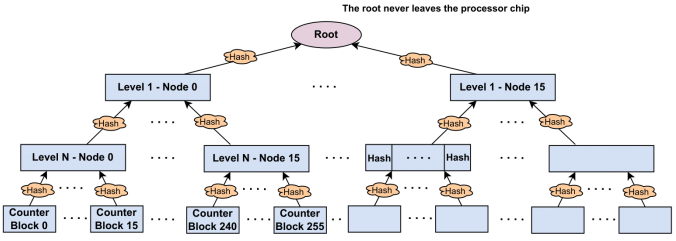

Fig. 2. Bonsai Merkle Tree over Encryption Counters

B. Our Baseline

Our baseline is built over the state-of-the-art proposed designs for GPUs equipped with secure memories. Plutus adopts the PSSM [36] design as a baseline, where each memory partition is equipped with its own security engines for MAC generation and encryption using counter mode encryption (CME), and hence each partition has its own BMT that is built over its local counter blocks. In addition, every memory partition integrates caches per each of counters, MACs, and BMT nodes, each of 2kB size, hence, 192kB in total (in case of Volta architecture [11]) for the whole GPU. While PSSM uses a truncated MAC of 4 bytes, Plutus uses an 8 bytes MAC to provide a higher level of security.

Each memory block is 128B divided into 4 sectors serving as the granularity for attaining security metadata. In other words, each data block needs 4 counters and 4 MACs, one per 32B sector. Counters are organized using split-counter scheme [33] modified by PSSM to comply with the sectored metadata caches as shown in Figure 4. Sectored metadata caches are used by PSSM to reduce useless metadata traffic. MAC caches benefit from a such design on both read and write transactions, while counters and BMT still need to get the whole block on reads for integrity verification, which uses the whole block as its hashing granularity. More sensitivity analysis for block sizes and BMT hashing granularity is provided in Section V.

III. Motivation

A. Performance Overheads of Security Metadata

To investigate the overheads of a secure GPU memory, the performance of different memory-intensive GPUs' benchmarks has been analyzed by comparing a design equipped with a security model against a baseline with no security. The security model adopts the PSSM [36] design, as described in Section II-B. The simulation details are provided later in Section V. As shown in Figure 6, for the benchmarks in our study, significant overheads are introduced due to the added security protection. The reason is the bandwidth contention due to security metadata, as shown in Figure 7. Each last-level cache (LLC) miss requires extra metadata to verify it, which could miss in their caches [27] too and result in extra memory requests competing on the available memory bandwidth.

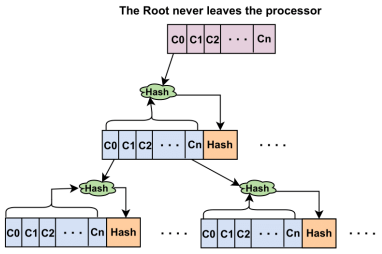

Fig. 3. Tree of Counters (SGX-Style Parallelizable Merkle Tree)

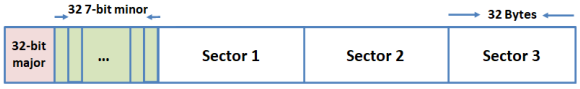

Fig. 4. A Counter Block in PSSM Sectored Design

Some previous works like common counters [18] and adaptive security support for heterogeneous memory on GPUs [35] addressed the traffic caused by counters and proposed designs to reduce it by exploiting read/write characteristics of GPUs applications. However, MAC traffic is still significant as pointed out by these works, and as can be observed from Figure 7. While minimizing MAC code size or computing a MAC for a large memory region [35] can reduce MAC misses in applications with good spatial locality. It is not the case for applications with random accesses in addition to reducing the security level by increasing the collision rate. Hence, a more efficient way that keeps the same security level is needed for optimizing the MAC bandwidth consumption in GPUs.

B. Data Value Locality

GPU workloads exhibit data similarity and homogeneity [5], which could be exploited for optimizing security metadata overheads. To study such patterns, we conduct an analysis of the data access patterns of these benchmarks. Data read/written from/to memory is studied in several ways. For a data sector of 32B, which is the access granularity, it is divided into multiple values of a certain size, including 32-bit, 64-bit, and 32-bit with masking its 4 least-significant bits. For each of these granularities, the reuse pattern among all data accesses is explored by caching the most recent values seen in previous accesses. The number of hits in this temporal value cache proves that there is a high value-reuse locality in GPU workloads' data accesses. The reuse patterns studied for the 32-bit values are explained in more detail below while represented in Fig. 8:

32-bit Values per Sector: If all 32-bit values - there are 8 in a 32B sector - hit in the value cache, reuse is counted.

32-bit Values as Two Parts: Each sector is divided into two halves. For now, let's assume that to consider a sector half reused, three out of the four small values should hit in the value cache, then if both halves are marked as reused values, the whole sector is marked as reused. Later in IV, we describe why we have such restrictions for considering a value reused.

32-bit Values as Two Parts with Masking the 4 least-significant bits: The 4 least-significant bits of studied chunks are masked to capture nearby values. Then, it is processed as in the previous case.

Figure 9 presents the reuse percentages for the aforementioned different scenarios when a value cache of size 2kB, keeping 512 different 32-bit values, is used per memory partition in Volta GPU architecture. For every memory read or write, values are inserted into this value cache if they are not already. On reads, before inserting values, the read value is checked for reuse. As can be observed from the figure, a lot of data values are identical to previously read/written values. Later, in Section 4.3, we show how these value similarities could be

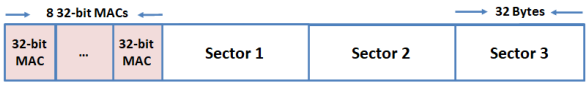

Fig. 5. A MAC Block in PSSM Sectored Design

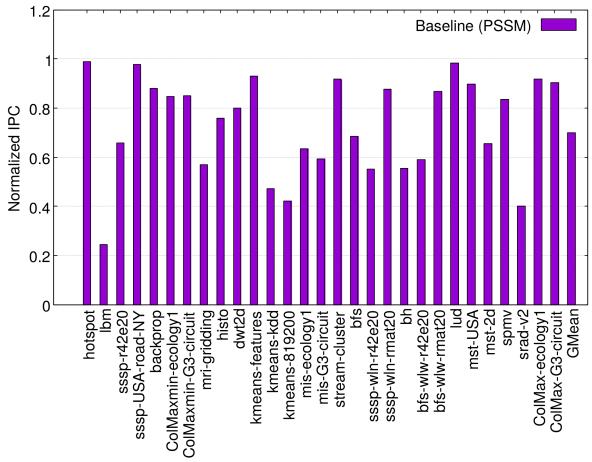

Fig. 6. Performance Overheads of Secure Memory

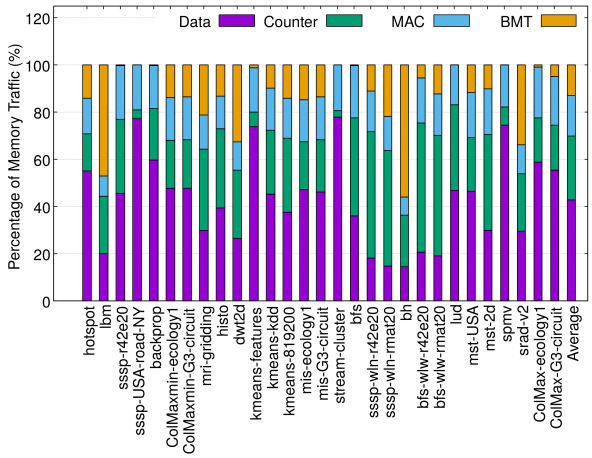

Fig. 7. Bandwidth Overheads of Secure Memory

leveraged to securely verify the integrity of data without the need to fetch its corresponding MAC.

C. Read-Only or Infrequently Updated Data

As many of the state-of-the-art works [18], [35] pointed out, most of the GPU data are read-only or updated infrequently. Counter values corresponding to such type of data either zeros or small values. Prior works tried to track if any writes happen to a region like 16kB. As long as no writes happen to this whole region, counter value zero is used without any memory requests for counters or BMT. However, on the first write received by this region, the whole region is no more considered read-only, and all new accesses have to get the original counters from memory for security purposes. However, such coarse-grain decisions could miss part of the optimization chances, especially in random access applications. However, tracking finer granularity requires more storage overhead and some performance penalties if the tracking metadata is stored in off-chip memory. Fig. 10 shows the breakdown of memory requests as read or write requests for the used benchmarks.

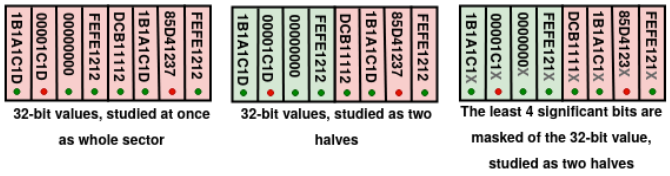

Fig. 8. Patterns of Value-Reuse for Considering Data Unit as Reused

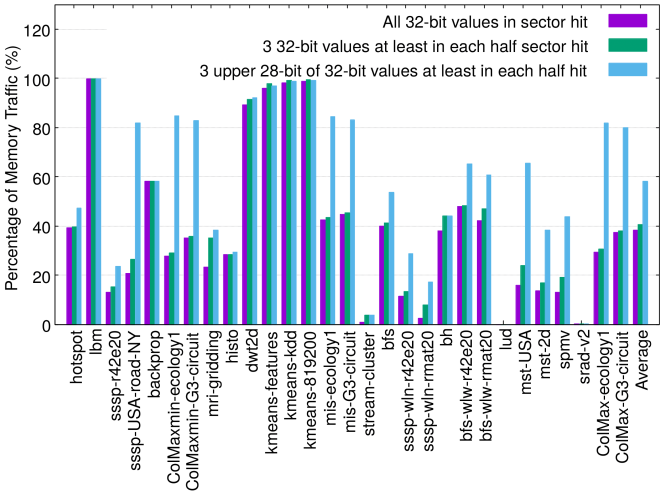

Fig. 9. 32-bit Values Reuse Percentage

IV. Plutus Design

A. Threat Model

Plutus threat model is similar to prior works that address memory security either in CPUs or GPUs [18], [20], [27], [33], [35], [36]. Basically, the trusted computing base (TCB) includes only the processor chip (GPU) and any internal components residing on it. Every component out of the processor chip is not trusted and is considered vulnerable to external or internal attacks. Thus, caches, internal wiring, and memory controllers are all parts of the trust base, and attackers cannot tamper with any data in these structures. As in prior works, our threat model only considers external physical attacks, while different side-channel attacks e.g., timing, power, electromagnetic, and template attacks [9], [12], [19] are all excluded in our threat model. Moreover, access pattern leakage (including due to accesses to security metadata blocks) is beyond the scope of our threat model, however, can be addressed using solutions such as Path-ORAM [25].

B. Security Guarantees

In our design, the following specific design details are used in our implementation as our baseline:

- **Encryption:** AES-XTS encryption is used to guarantee that any alteration of the ciphertext completely randomizes the plaintext. As diffusion is needed for our model, refer to IV-C, this justifies the choice of AES-XTS over CME, in which tampering is localized. Although CME is better at hiding encryption/decryption latencies by overlapping them with data fetches, while direct encryption exposes it to the critical path but this is not an issue in GPUs where thread-level parallelism hides it as indicated by [37]. Along with the addresses, counters are still

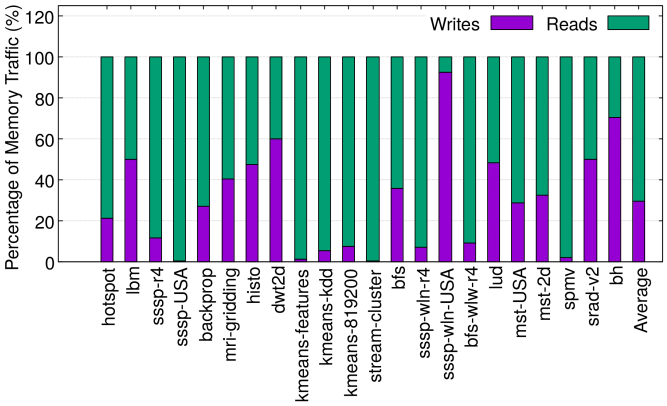

Fig. 10. Read-Write Breakdown of Memory Traffic

used as tweaks to provide temporal uniqueness of the generated ciphertexts for same plaintexts. They follow the state-of-the-art split-counters [33] design in a sectored fashion as proposed by PSSM [36], where a counter per 32B sector is used.

- **Message Authentication Codes (MAC):** are assigned one per data sector, each of 8 bytes. However, Plutus scheme is agnostic about the MAC code size.
- **Bonsai Merkle Tree (BMT):** An 16-ary BMT is built over the encryption counters to guarantee their freshness at the time they are fetched from the off-chip memory. In our implementation, we adopt the lazy update.

In our baseline, security metadata is organized in memory blocks/cachelines of 128B same as the data blocks. They are cached to allow for faster operations and less overheads [27].

C. Value-Based Integrity Verification

Plutus introduces a novel way for verifying data integrity without the need for fetching MACs from the off-chip memory, regardless of the MAC code size, thus, reducing the bandwidth consumed by security metadata. The Plutus method of integrity verification relies on the data itself. It works by bookkeeping the most recently seen values in the memory controller (e.g., a value cache) and using them to verify data arriving from the off-chip memory. The key insight is to leverage the property of cryptography, i.e., counter-mode encryption with AES-XTS, that the plaintext of a tampered ciphertext does not exhibit frequent value locality as observed in Section III.B. In other words, the tampered plaintext has a much smaller possibility to hit in the value cache than untampered plaintexts. Therefore, if a plaintext hits in the value cache, it can be considered untampered. For value cache misses, which are rare cases in normal execution, MACs are used then to check the integrity. The detailed security analysis is presented later in this section. Note that the replay attacks using previous values are still detected by counter verification using the integrity tree.

How does Plutus' Integrity Verification Work?

When a memory request is issued from the L2 cache to the memory controller, it could be read or writeback. The request flow in both cases is illustrated in Figure 11. Let us start with writes. As a writeback request ① arrives from the L2 cache

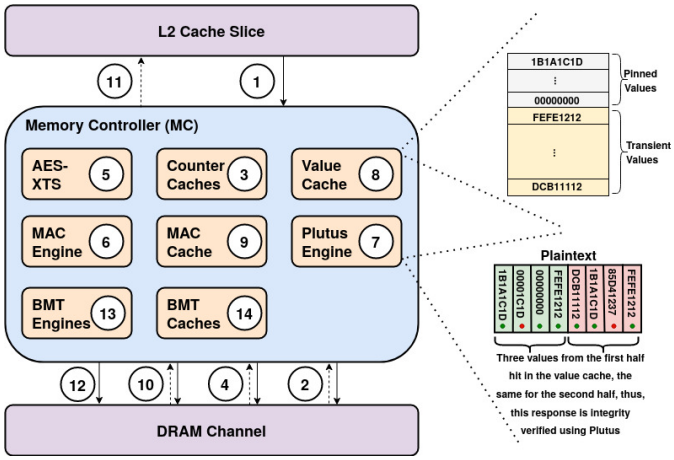

Fig. 11. Memory Requests Flow in Plutus

and as a preparation for leaving the trusted boundary, the data is divided into values of size M bits (explained later), and added to the list of recently seen values ⑧. At the same time, the counter for this piece of data is brought to the memory controller ④ and verified ⑬ in case it is not cached already ③. Usually, the MAC will be computed too, but in Plutus it is deferred until it is needed. Once the counter is available, the data and its counter are sent to both the encryption engine ⑤ and the MAC engine ⑥. Concurrently, the data is processed by Plutus' engine ⑦ to see if the data is assured to pass the value-based integrity verification the next time when brought to memory (explained later). If so, there is no need to update the normal MAC. Otherwise, as in conventional designs, the new MAC is computed and written to the MAC cache ⑥. For MAC cache misses, the cache line is brought to the cache and updated ⑩. Finally, the encrypted data is passed to the memory ⑫.

For a read, when the request arrives at the memory controller ①, the data needs to be verified and decrypted before sending it to the requesting core. Thus, the counter assigned to the read sector, if not cached ③, is fetched ④ and verified ⑬. Then, it is supplied to the decryption engine ⑤ as part of the AES-XTS tweak. Generally, the decryption process goes concurrently with the data integrity verification using MACs. In Plutus, we delay or eliminate MAC verification as we hope to verify the data integrity via data value reuse and avoid any extra memory requests due to MAC. Although this could introduce some serialization to the security operations and add some extra latency, GPUs can hide such latency by scheduling other threads/warps by leveraging thread-level parallelism. Thus, after getting the plaintext of the block, the data is sent to Plutus Engine **to perform integrity verification using the book-kept recently-seen values following the following steps, which are shown in Figures 11 and 12:**

1) The data is divided into values of M bits as shown in the bottom right of Figure 11.
2) Each M-bit value is probed in the value cache ⑧. The number of hits decides if it is sufficient to consider the

data unit verified or not.

3) If the data is proved to be integrity protected, it is passed to the requesting core directly ⑪. Otherwise, the MAC has to be checked either from the MAC cache ⑨ or by fetching it from memory ⑨, then after MAC verification, data is sent to the requesting core ⑪.

Why is Plutus Secure?

To verify a unit of data of N (e.g., N=32) bytes, it is divided into a collection of M-bit values. Thus, as a total, there are $\frac{N*8}{M}$ values in one unit of data. Each M-bit value probes the value cache, which keeps K recently seen values.The cached values are categorized into two types, pinned ones that never get replaced or removed after pinning, and transient values that could be replaced using the least recently used replacement policy. Values start as transient, and while in the cache, might get promoted to a pinned value based on their frequency of accesses. A small counter is associated with a cached value and it gets incremented on each hit on this value. Once the counter reaches a certain threshold, the value becomes a pinned value. Pining some values, the most frequent-accessed ones, is useful for write requests as illustrated later. In our model, a quarter of the value cache is reserved for these pinned values.

Given the space of M-bit values, 2^M, the chance of hitting in the value cache is $\frac{K}{2^M}$ for an M-bit value. The reason is that given the property of cryptography, i.e., AES-XTS, any tampering on the ciphertext would be diffused to uniform changes in the plaintext. In other words, as long as the probability to see a tampered value hitting in the value cache is sufficiently low, the values hitting the value cache can be considered tamper-free or integrity verified. Next, to consider the whole unit of data not tampered with, there should be a minimum number of its M-bit values hitting in the value cache, such that the probability of a tampered data unit passing the value-based check would be similar to or lower than the MAC collision rate.

The answer to this problem could be studied using the binomial distribution shown in equation 1 [1]. The binomial distribution finds the probability to have x success events out of n independent trials, given the success probability for a single trial is p. In the MAC problem, we study the event of the data being tampered with while the verification passes. The n trials are represented by the number of M-bit values (i.e., n = 4) in the studied part of the data, whether it is a whole sector or part of it. The probability of success, p, for a single trial, is to have a value that hits in the value cache and it's actually a tampered value. It has been derived earlier in this section, $p = \frac{K}{2^M}$. The answer to our question is x which is the minimum number of M-bit values that hit in the value cache. As it has been proved by Gueron in [7] that a forgery success probability of $\frac{1}{2^{56}}$ is sufficient, we need to select x such that the inequation in equation 1 holds.

Besides x in equation 1, we also need to consider the granularity that this probability P_x should be checked. For Plutus, as mentioned previously, AES-XTS is used for encryption. Therefore, Plutus considers the size of the block cipher of the AES-XTS encryption algorithm, i.e., 128 bits,

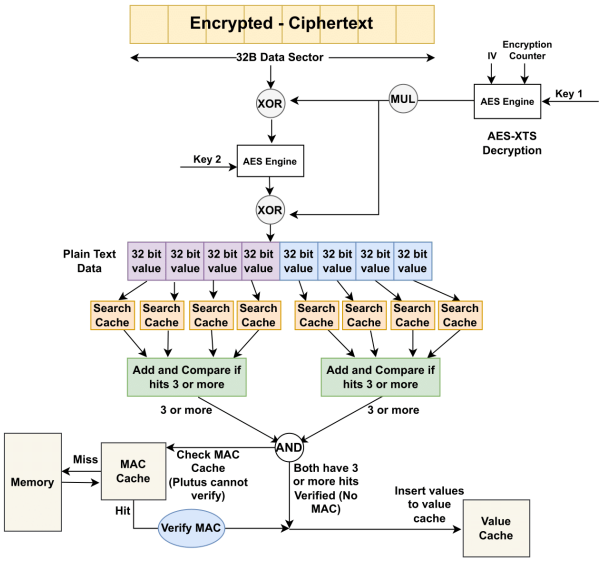

Fig. 12. Value-Based Integrity Verification Steps

as the granularity for value-based integrity verification. Thus, considering a system where memory access granularity is A bits, $\frac{A}{128}$ data units are studied together. If each of those units meets the requirement for Plutus' integrity verification, which is discussed earlier, the whole data access is considered verified. Otherwise, value-based integrity verification cannot give a strong guarantee for not being forged. In such cases, the corresponding MAC needs to be fetched from memory for regular integrity verification.

$$P_x = \binom{n}{x} p^x (1-p)^{n-x} <= \frac{1}{2^{56}} \quad (1)$$

Design Implementation

For our implementation, where Nvidia Volta GPU architecture [11] is used, cache blocks are of size 128 bytes sectored blocks with a memory access granularity of 32 bytes, i.e., A =256 bits. Each memory access consists of two 128-bit blocks, requiring two rounds of value-based integrity verification. For the value size M, 32-bit is used in Plutus implementation where the upper 28 bits of the value are used for the matching test. Based on the acceptable forgery success probability and with a 256-entry value cache, by solving the inequality 1 for x, out of the four 32-bit values in every 128 bits, at least three of them need to hit in the value cache. Obviously, both halves (every 128 bits in one memory access) need to satisfy this to skip the need for original MAC access. The exact flow for Plutus verification for 128-bit value is shown in Fig. 12.

How does Plutus Know if Writes are Verifiable at Next Read?

For a dirty data unit, if the number of hits in the value cache satisfies the inequation 1 and the hits are in the pinned region of the value cache, then the data is guaranteed to be verifiable on next fetch from memory as shown in the right of Fig. 11.

Security Impact: similar to compression, value prediction, and counter value analysis, timing side-channel attacks are possible if MAC values show significant reuse. However, the

leakage is limited to how common are the values in certain applications. Eliminating such side-channel is possible through randomly obfuscating MAC blocks and issuing random accesses. As mentioned in our threat model, access pattern leakage (including accessing MACs and data blocks) is beyond the scope of our threat model, and can potentially be addressed through optimized Oblivious RAM implementations [25].

D. Compact Mirrored Counters

To reduce the memory traffic due to counters and BMT, Plutus leverages the fact that many cache lines are seldom updated in GPU workloads. Although there are prior works exploiting a such feature of GPU applications like Common Counters [18] and Adaptive security support for Heterogeneous Memory on GPUs [35]. They do track memory changes in large chunks which results in many lost chances for optimizations.

With limited updates, counters for many cache lines are low and their major counters are 0 if all minor counters are low. As a result, the standard split counter organization leads to wasted space as many bits in the major counter and minor counters are zeros. To address this issue, Plutus introduces mirrored counters in a compact way. For each cache line/sector, a compact counter, e.g., 2 or 3 bits, is used for encryption until it saturates, at that time, its value is propagated to the original copy of counters, which follows the standard split counter design [33]. To select whether the compact or the standard split counter is to be used, a one-bit flag is also maintained per cache line/sector, as shown in Fig. 11. When a compact counter is used, its major counter is 0, and does not need to be fetched. Note that as multiple minor counters share a major counter, if a minor counter overflows, the major counter is incremented and all the cache lines/sectors sharing the major counter need to use the split counters instead of compact ones.

Based on the fact that a large portion of GPUs data is read-only or rarely updated [35] [22], it suffices to capture these infrequent updates using smaller counters in trade-off more compaction and thus better cacheability and spatial locality. Compared to prior works [18] [35], which track memory changes in large chunks, Plutus uses the same fine granularity as the original counters, one per data sector, but with a fewer number of bits. Thus, it captures as much as possible of cases that still can be served in an optimized way. These minimized counters could be designed in different ways and sizes. Plutus studies three different designs as discussed in the evaluation section V. **These designs are:**

- **2-bit compact Counter per 32B sector:** Thus, it provides 4X compaction compared with original counters but overflows on the third write.
- **3-bit compact Counter per 32B sector:** It provides more flexibility for writes compared to a 2-bit counter and still provides good compaction.
  The drawback of the above two organizations is that for applications that experience heavy write accesses to the point where most of the compact counters get saturated, it might end up with two accesses to memory for getting the counter, first to compact ones, realizing that it's saturated, then accessing the original counters.
- **3-bit Adaptive Compact Counter:** To eliminate the severe effect of having to check two blocks before getting the counter, an adaptive scheme is used. In this design, a counter sector of 32B has 64 3-bit compact counters in addition to a counter that keeps track of the number of saturated counters in this block of compact counters as shown in Figure 13. A third layer of access is needed, however, it's too compact to the point where its misses are negligible. This layer assigns a bit for each compact counter block, this bit is set to one once the number of saturated counters reaches a certain threshold, in Plutus this threshold is 8, based on the observation by prior work [22] that mostly less than 25% of counters in a block is accessed. Hence, half this number is used as a threshold to convert to using the original counters for this block of compact counters. However, to make later decryption for data corresponding to those counters possible, all non-saturated compact counters are copied to the original ones, and the access to this compact counter block is stopped by the control layer and no re-encryption is needed. As this adaptive scheme provides 2X compaction, only two original counters blocks are needed to synchronize original counters to compact counters updates, given that the memory access granularity is 32B.

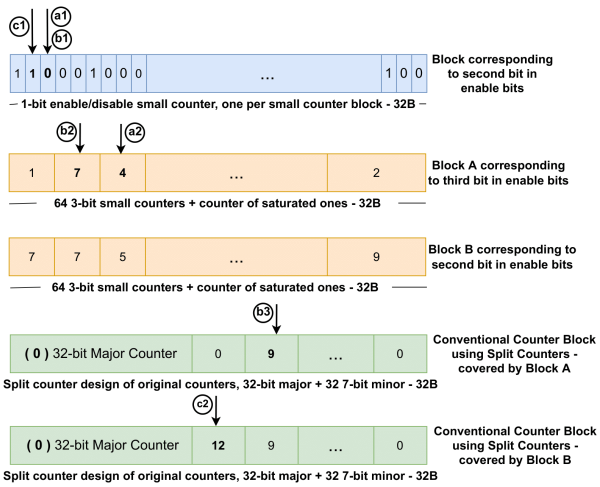

Fig. 13. Counter Access Flow in Plutus Design
For access **(a)**, it checks the enable bit corresponding to its small counter block, since 0 means still enabled, it accesses the small counter, reads the value is less than the overflown value of 7, which means it is a valid value. Same with access **(b)**, but b's counter is overflown (7), which means it needs a second access to the original counter (9). However, for access **(c)**, it reads 1 from enable bits, which indicates that direct access to original counters should be made (12).

The security guarantees of the counters are kept by having a small BMT over these counters to validate their integrity before using them by the processor. Due to the small number of blocks to be protected by this BMT compared to the original ones, it has better locality and cacheability which makes its traffic to the off-chip memory small.

E. Achieving Finer Granularity Security Metadata Blocks

The previous work, PSSM [36], suggested using sectored caches for security metadata to avoid fetching extra metadata uselessly. It works for MAC read and write accesses, but not for counters and BMT, where the whole block still has to be brought completely to the memory controller to accomplish the integrity verification. Basically, this comes from using the whole counter block as a unit for generating the BMT hash/MAC. Once a counter misses in the counter cache and is fetched from off-chip memory, its integrity has to be verified via BMT traversal. To generate the parent hash/MAC of the counter in the BMT, the unit used for the hashing algorithm is the 128B block. As a result, another three sectors have to be fetched to the memory too. The same applies to BMT intermediate node verification.

In Plutus, we propose finer granularity counter blocks for building the BMT to eliminate any useless memory traffic for both counters and BMT. When the size of the BMT leaf is reduced, e.g., from 128B to 32B, one memory access, instead of four, is needed to fetch the leaf for verification purposes. However, as one intermediate node covers the same number of leaves but of smaller sizes, more intermediate nodes are needed to build the tree and, hence, extra storage. Note that more intermediate nodes may or may not lead to a taller tree, as one can adjust the number of children of the root as long as it does not exceed the arity. For example, an 8-ary tree, with 128 leaves would have a height of 4 (the number of nodes at each level is 128-16-2-1) while an 8-ary tree with 512 leaves has the same height (the number of nodes at each level is 512-64-8-1).

Three BMT design options are studied, shown in Fig. 14:

- **Security metadata block size as 128 B:** This is the design used by prior works. It leads to a more compact tree. However, it consumes high memory bandwidth and affects performance adversely.
- **Counter/MAC blocks of 32B while the BMT nodes use 128B:** Only the leaf level differs from the previous design, whereas the rest of the tree still has the same structure. Due to the increased number of leaf nodes - each previous hash value now is replaced by four-, the tree size increases and the height might as well.
- **All security metadata (Counter / MAC / BMT) use 32B blocks:** In addition to using 32B counters, BMT nodes are shorthanded into 32B with a fourth of the previous arity. The size of this BMT design is the same as the one above. However, this one grows vertically and consequently has a larger height whilst the above one is more flattened.

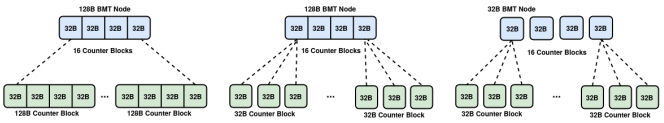

Fig. 14. The Different Design Options for the BMT

F. Hardware Overheads

The hardware overheads of Plutus are as the following. First, value-based integrity verification requires a value cache of 256 32-bit entries, 1kB, including the counter used for pinning values. Compact counters require two caches, one for counters and another for BMT nodes, each of 2kB, so 4kB in total. For the finer granularity security metadata blocks, the storage required for BMT increases due to the decreased arity. It goes from 145.125kB to 1.33MB.

V. Evaluation

A. Simulation Environment

We use GPGPU-Sim v4.0 [13] to evaluate Plutus. The configuration of the baseline GPU is shown in Table I, which is modeled based on Nvidia Volta architecture.

TABLE I
BASELINE GPU CONFIGURATION

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

A device memory of 4GB is assumed to be the range of protected memory. The cacheline size is 128B. As shown in Table I, the data caches, both L1 and L2 are sectored by default in Nvidia Volta architecture. And by applying PSSM design, metadata caches for counters, MACs, and BMTs are also sectored and use partition local addresses. The detailed configurations of metadata caches are shown in Table II.

TABLE II
METADATA CACHES AND SECURITY CONFIGURATION

MAC Cache, Counter Cache, BMT Cache	Each 2kB/memory partition, 128B blocks, 4-way sectored, 256 MSHRs, allocate-on-fill policy
MAC Latency	40 cycles
Encryption Engine AES-XTS Latency	1 pipelined AES/memory partition 40 cycles
Value Cache	1kB, Fully-associative, 25% pinned

We use a collection of benchmarks from Rodinia-3.1 [4], Parboil [26], Lonestargpu-2.0 [14] and Pannotia [3]. They are selected based on their memory bandwidth utilization. Both high and medium memory-intensive applications are included in our evaluations. An application that uses more than 50% of available memory bandwidth is considered high memory intensive, and if not while using more than 20%, it's classified as medium memory intensive. We simulate the first two billion instructions from each of these selected benchmarks unless they have less number of instructions. For Plutus scheme, to enable data-based integrity verification, a value cache of size 256 32-bit entries (28 bits value and 4-bit frequency use counter), 1kB, is used per partition. For the compact mirror counters, two new caches are used, for the new counters and

their BMT, each of the same size of the original metadata caches, 2kB per partition.

B. Experimental Results

First, the performance improvement of each proposed technique is evaluated separately. Then, the combination of these schemes is simulated.

1) Value-Based Data Integrity Verification: The data integrity verification based on data-reuse locality is evaluated and the results are shown in Figure 15. The figure shows the number of instructions per cycle (IPC) normalized to a system with no security support. The comparison is made with our baseline PSSM [36]. From the figure, we can see that value-based integrity verification improves the performance by 4.94% on average and up to 19.89%.

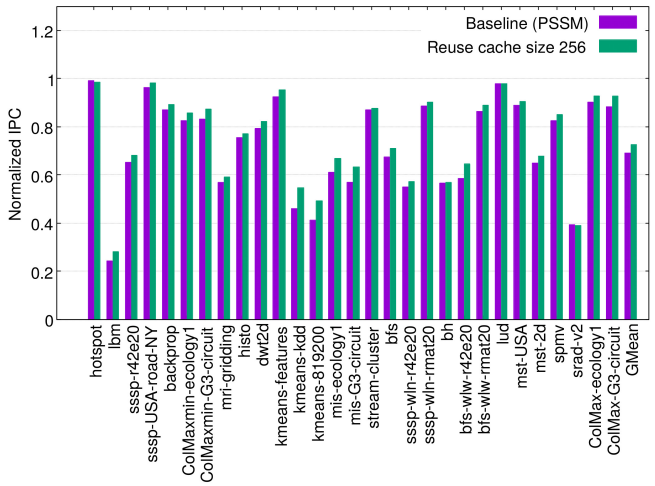

Fig. 15. Performance Improvement due to Value-Based Integrity Verification

We observe that although some applications have high value-reuse rates, they do not show high performance improvement. One reason is that writes cannot be verified using value reuse unless they hit only with pinned entries to guarantee the ability to verify them the next time they are fetched from off-chip memory. Also, in some cases, when two data units (sectors) share the same MAC sector, if they happen to be requested by the processor closely in time, and one is value-verified while the other is not, then, the benefit is nearly nullified.

2) Finer-grain Security Metadata Blocks: Figure 16 compares the performance among three different designs presented in Fig. 12. As can be seen in the figure, the two designs with finer-grain metadata blocks have much better performance compared to the baseline. The third design, where all security metadata have blocks of size 32B, has the best performance, 10.57% on average with up to 74.85% over the baseline.

3) Compact Mirrored Counters: As discussed in Section IV.D, we explore three design options using different compact mirrored counters and Figure 17 shows their performance results. From the figure, we can see that 2-bit compact counters tend to overflow quickly, which results in extra traffic for accessing two layers of counters. In comparison, 3-bit compact

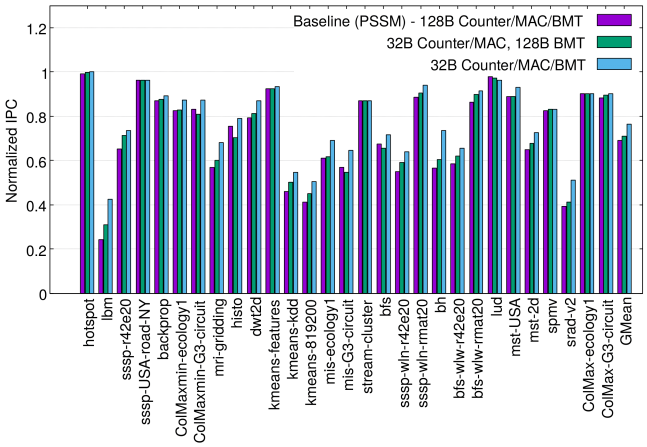

Fig. 16. Performance Improvement due to Different Metadata Block Granularities

counters perform better in many of the benchmarks. The adaptive compact counter scheme learns from the application behavior and achieves the best performance, 2.07% on average and up to 8.28% performance improvement.

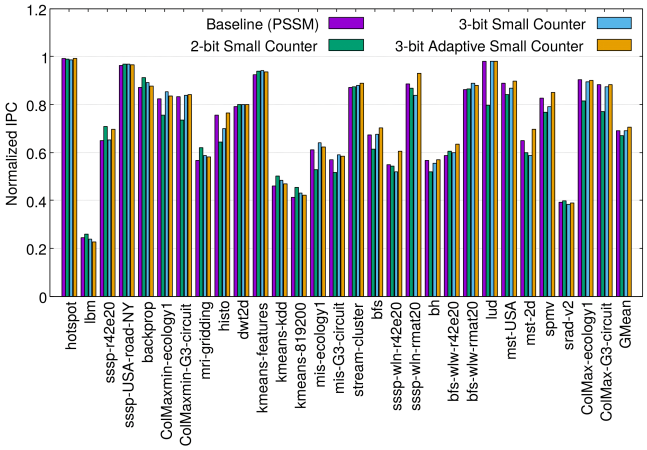

Fig. 17. Performance Improvement due to Different Compact Counter Designs

4) Plutus Overall: By combining all three schemes, Plutus achieves high performance as shown in Figure 18. It improves the IPC by 16.86% over the PSSM design, and by 8.97% over common counters combined with PSSM, where both use CME while Plutus uses AES-XTS. As in Fig. 19, Plutus reduces security metadata traffic by 48.14%.

Given the recent works that address the overheads of counters and integrity trees of other types of accelerators, such as MGX [8], TNPU [15], and softVN [31], Plutus remains effective. Figure 20 shows the performance of Plutus when the integrity tree traffic is totally eliminated.

5) Sensitivity to value cache size: As Figure 21 shows, a value cache size larger than the value used in the calculations of inequality 1 does not bring much improvement; 256 entries per partition can capture most of the repeated values.

C. Power Saving

Figure 22 shows the overall average power consumed by Plutus scheme normalized to consumed power in a system

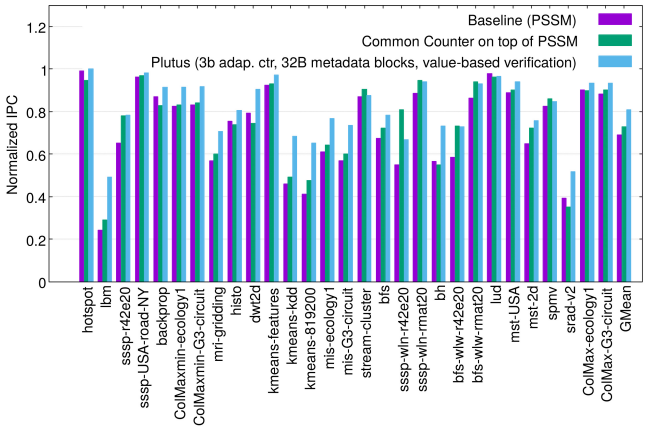

Fig. 18. Overall Performance Improvement of Plutus Design

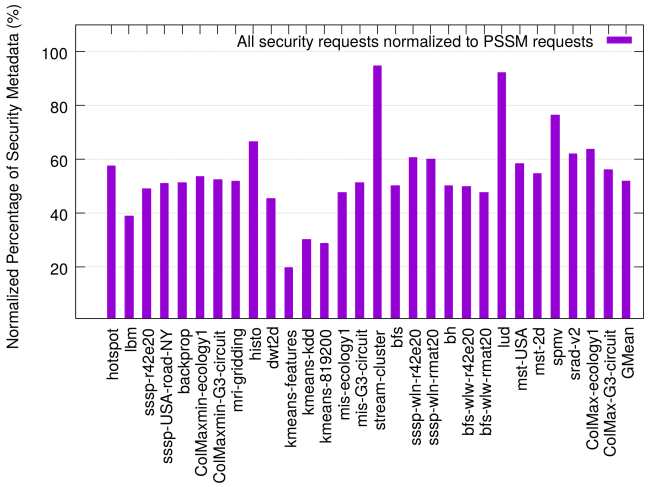

Fig. 19. Security Metadata Bandwidth Consumption Reduction in Plutus

with no security. Plutus reduces these security overheads from 36.9% in 8B-MAC-PSSM scheme to 17.8%.

VI. Related Work

Memory Protection: In the last two decades, there has been a significant amount of work targeting the confidentiality and integrity protection of data residing in an untrusted memory [6], [17], [18], [20], [22], [23], [27], [28], [33]–[37].

Memory Encryption: For **confidentiality**, the authors of [27] introduced using counter-mode encryption (CME) instead of direct Advanced Encryption Standard (AES) to hide decryption latency. Both [27], [34] pointed out that caching counter values used for encryption helps reduce the time needed for generating the OTPs. Yan et al. proposed split counters to be used for CME instead of monolithic counters. The counters in this design consist of two parts, minor counters dedicated to each data block, and a major counter shared by several blocks. It reduces per-counter size which improves the on-chip caching of counters. Common Counters [18] studies the uniformity of counter values in GPUs applications and they figured out that blocks are mostly updated in a uniform manner. From this finding, they proposed a scheme that stores a set of

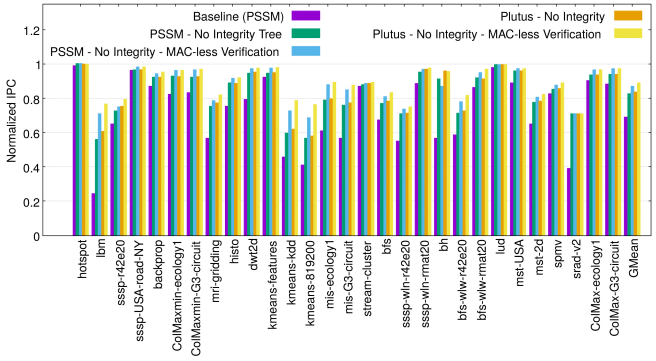

Fig. 20. Performance Comparison between Plutus and Other Memory Security Designs

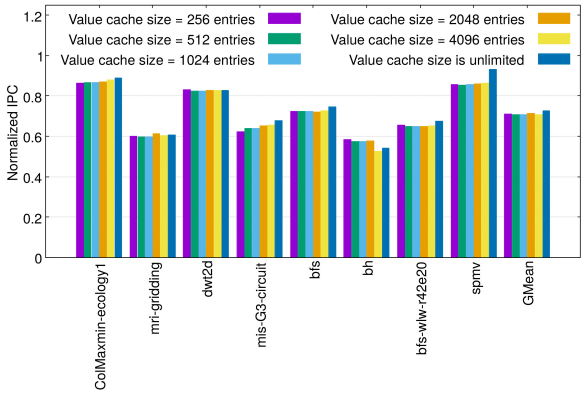

Fig. 21. Performance Sensitivity to Value Cache Size

common counter values in the trusted boundary, these values are the ones common among a large number of data blocks. This improves the counter fetch latency since, in most cases, counter values are get from the on-chip set of counters. PSSM [36] proposed partitioned and sectored security metadata that fits GPUs design more than non-sectored metadata since having security data per sector avoids the need for fetching all block sectors for the sake of security checks. Adaptive security support for Heterogeneous Memory on GPUs [35] exploits some of the GPUs workloads' features as read-only data and streaming accesses to propose optimization for the security handling of such data. It introduces a predictor for read-only regions, and another one for streaming accessed regions along with memory access trackers for analyzing such accesses. Read-only regions use one shared counter stored on-chip, hence counter traffic is eliminated for such regions. They also proposed dual-MAC granularity to optimize for streaming accessed regions. SoftVN [31] manages counters in the software layer in a coarse granularity for a group of memory locations that share the same update pattern, reducing the number of kept counters to one. However, this design has some restrictions to work as the updates should happen in sequential order and one at a time per a group of tracked locations. Also, both MGX [8] and TNPU [15] are optimizing counters as well as integrity tree storage plus traffic for specific accelerators such as DNN, by holding one counter value per

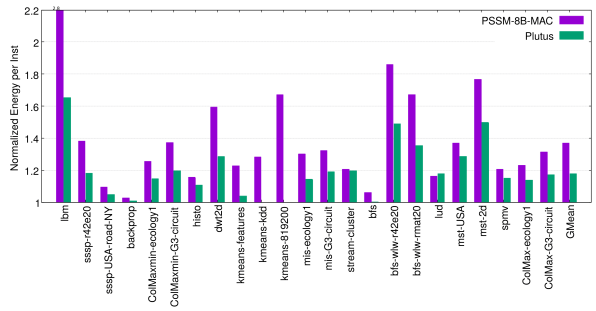

Fig. 22. Normalized Energy per Instruction

whole data structure and using computations to get VNs based on the regularity of access patterns in the applications used for such accelerators. However, they don't optimize for MAC traffic, here Plutus bandwidth-less integrity verification could be applied. Thus, MGX is orthogonal to our work, and it can be enabled on the same system as Plutus.

For **Integrity Verification:** The hardware-based scheme used to prevent replay attacks, is the integrity trees. Mainly, there are two types of integrity trees, one that consists of MAC/HASH and it is called Merkle Trees [17]. The other one is built of counters, usually referred to as version counters. Basically, the original design that uses Merkle Trees for memory integrity computes MAC or hash values over the whole memory data. The root is kept secure on-chip. However, other nodes can leave the processor chip. Internal nodes can be cached to fasten the process of integrity verification by avoiding going to the off-chip memory many times for verification. In addition, once an internal node hits in the cache, the verification stops at this node and it's considered secure as if it's the root. Since this scheme uses all data blocks to construct the tree, it requires a lot of storage. Hence, Bonsai Merkle Tree [20] has been proposed. It proposes computing stateful MACs over data blocks and counters while the integrity check is only needed for counters used in MAC generation.

GPU TEEs: With the popularity of GPUs as accelerators in untrusted remote cloud systems, an increased urge to have TEEs for GPUs has been posed. Recent works such as Graviton [32], HIX [10] and Telekine [9] have suggested TEE designs for GPUs inspired by ones used for CPUs such as Intel SGX [16] and ARM TrustZone [2].

VII. Conclusion

Although trusted computing is increasingly on demand, memory security has a high cost of metadata storage and memory traffic. Since GPUs are bandwidth-intensive, secure memory with low-bandwidth overhead is highly desired. Plutus comes up with three novel techniques that optimize bandwidth usage for secure memories in GPUs. It exploits the patterns of having repeated or close data values that could be verified without MACs with an accuracy level comparable to or better than the collision rate of the state-of-the-art MAC functions. It minimizes the counter and BMT traffic with two complementary approaches. First, it introduces a layer of compact counters to achieve better cacheability and locality. Second, by exploring the trade-off between the BMT organization and metadata granularity, Plutus chooses to use small counter blocks (32B) to avoid fetching extra ones on verification. Our evaluation shows that Plutus improves the throughput of GPUs by 11.65% on average and saves around 42.14% of security metadata needed bandwidth compared to the state-of-the-art approach.

Acknowledgement

Part of this work was funded through Office of Naval Research (ONR) grants N00014-21-1-2809 and N00014-21-1-2811, and the National Science Foundation (NSF) grants CNS-1814417, 1908406, CNS-1908471, and CNS-2008339, and an AMD gift fund. 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] *Binomial Distribution*. New York, NY: Springer New York, 2008, pp. 44–45. [Online]. Available: https://doi.org/10.1007/978-0-387-32833-1_34

[2] T. Alves, "Trustzone: Integrated hardware and software security," *White paper*, 2004.

[3] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," 09 2013, pp. 185–195.

[4] 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 *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[5] E. Choukse, M. B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, and S. W. Keckler, "Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 926–939.

[6] B. Gassend, G. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, 2003, pp. 295–306.

[7] S. Gueron, "A memory encryption engine suitable for general purpose processors," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 204, 2016.

[8] W. Hua, M. Umar, Z. Zhang, and G. E. Suh, "Mgx: Near-zero overhead memory protection for data-intensive accelerators," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 726–741. [Online]. Available: https://doi.org/10.1145/3470496.3527418

[9] T. Hunt, Z. Jia, V. Miller, A. Szekely, Y. Hu, C. J. Rossbach, and E. Witchel, "Telekine: Secure computing with cloud gpus," 2020.

[10] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity gpus," 04 2019, pp. 455–468.

[11] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *ArXiv*, vol. abs/1804.06826, 2018.

[12] E. Karimi, Z. H. Jiang, Y. Fei, and D. Kaeli, "A timing side-channel attack on a mobile gpu," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, pp. 67–74.

[13] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[14] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009. [Online]. Available: http://iss.ices.utexas.edu/Publications/Papers/ispass2009.pdf

[15] S. Lee, J. Kim, S. Na, J. Park, and J. Huh, "Tnpu: Supporting trusted execution with tree-less integrity protection for neural processing unit," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 229–243.

[16] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2487726.2488368

[17] R. C. Merkle, "Protocols for public key cryptosystems," in *1980 IEEE Symposium on Security and Privacy*, 1980, pp. 122–122.

[18] S. Na, S. Lee, Y. Kim, J. Park, and J. Huh, "Common counters: Compressed encryption counters for secure gpu memory," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 1–13.

[19] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Side channel attacks on gpus," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 4, pp. 1950–1961, 2021.

[20] 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 2007)*, 2007, pp. 183–196.

[21] J. Ryoo, M. Fan, X. Tang, H. Jiang, M. Arunachalam, S. Naveen, and M. T. Kandemir, "Architecture-centric bottleneck analysis for deep neural network applications," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 205–214.

[22] 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 *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 416–427. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00041

[23] O. Shafi and J. Bashir, "Freqcounter: Efficient cacheability of encryption and integrity tree counters in secure processors," *Journal of Systems Architecture*, vol. 119, p. 102252, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762121001727

[24] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph processing on gpus: A survey," *ACM Comput. Surv.*, vol. 50, no. 6, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3128571

[25] E. Stefanov, M. V. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," *J. ACM*, vol. 65, no. 4, apr 2018. [Online]. Available: https://doi.org/10.1145/3177872

[26] J. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. Liu, and W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[27] G. Suh, D. Clarke, B. Gasend, M. van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 339–350.

[28] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," *SIGPLAN Not.*, vol. 53, no. 2, p. 665–678, mar 2018. [Online]. Available: https://doi.org/10.1145/3296957.3177155

[29] M. Taher, "Accelerating scientific applications using gpu's," in *2009 4th International Design and Test Workshop (IDT)*, 2009, pp. 1–6.

[30] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IX. New York, NY, USA: Association for Computing Machinery, 2000, p. 168–177. [Online]. Available: https://doi.org/10.1145/378993.379237

[31] M. Umar, W. Hua, Z. Zhang, and G. E. Suh, "Softvn: Efficient memory protection via software-provided version numbers," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 160–172. [Online]. Available: https://doi.org/10.1145/3470496.3527378

[32] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on gpus," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 681–696. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/volos

[33] 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'06)*, 2006, pp. 179–190.

[34] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 351–360.

[35] S. Yuan, A. Awad, A. W. B. Yudha, Y. Solihin, and H. Zhou, "Adaptive security support for heterogeneous memory on gpus."

[36] S. Yuan, Y. Solihin, and H. Zhou, "Pssm: Achieving secure memory for gpus with partitioned and sectored security metadata," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 139–151. [Online]. Available: https://doi.org/10.1145/3447818.3460374

[37] S. Yuan, A. W. B. Yudha, Y. Solihin, and H. Zhou, "Analyzing secure memory architecture for gpus," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 59–69.