Morphable Counters: Enabling Compact Integrity Trees For Low-Overhead Secure Memories

Gururaj Saileshwar[§], Prashant J. Nair[†*], Prakash Ramrakhyani[‡], Wendy Elsasser[‡], Jose A. Joao[‡], Moinuddin K. Qureshi[§]

[§]*Georgia Institute of Technology* [†]*IBM Research* [‡]*Arm Research*

{gururaj.s,moin}@gatech.edu Prashant.Nair.J@ibm.com {Prakash.Ramrakhyani,Wendy.Elsasser,Jose.Joao}@arm.com

***Abstract*—Securing off-chip main memory is essential for protection from adversaries with physical access to systems. However, current secure-memory designs incur considerable performance overheads – a major cause being the multiple memory accesses required for traversing an integrity-tree, that provides protection against man-in-the-middle attacks or replay attacks.**

In this paper, we provide a scalable solution to this problem by proposing a compact integrity tree design that requires fewer memory accesses for its traversal. We enable this by proposing new storage-efficient representations for the counters used for encryption and integrity-tree in secure memories. Our *Morphable Counters* are more cacheable on-chip, as they provide more counters per cacheline than existing split counters. Additionally, they incur lower overheads due to counter-overflows, by dynamically switching between counter representations based on usage pattern. We show that using Morphable Counters enables a 128-ary integrity-tree, that can improve performance by 6.3% on average (up to 28.3%) and reduce system energy-delay product by 8.8% on average, compared to an aggressive baseline using split counters with a 64-ary integrity-tree. These benefits come without any additional storage or reduction in security and are derived from our compact counter representation, that reduces the integrity-tree size for a 16GB memory from 4MB in the baseline to 1MB. Compared to recently proposed VAULT [1], our design provides a speedup of 13.5% on average (up to 47.4%).

***Index Terms*—Memory Security, Replay Attack, Merkle Tree, MAC, Intel SGX, Split Counters, Encryption, Compression.**

I. Introduction

Securing system main-memory from physical attacks is important for building trusted data-centers. Numerous attacks [2], [3], [4], [5], [6] have demonstrated that adversaries with physical access can take control of a system through unauthorized reads and modification of main-memory contents. Commercial solutions like Intel's Software Guard Extensions (SGX) [7], [8] take an important step towards secure memories, by providing data encryption, integrity and replay attack protection for small regions of main memory. However, extending such mechanisms to secure entire memories comes at the cost of considerable performance overheads [1], [9], [10].

Securing commodity memory requires security metadata on each data access. For data encryption, a counter needs to be fetched from memory. For verifying data integrity, a cryptographic hash of data (MAC) is fetched from memory. Furthermore, to prevent replay attacks, an integrity tree is traversed generating several additional memory accesses. These accesses stress the memory bandwidth causing performance slowdown. While recent proposals [1], [10] address the overheads of accessing MACs, integrity-tree traversal continues to be a bottleneck as it can generate several memory accesses.

Prior proposals optimize integrity tree traversal by caching tree entries in on-chip caches [7], [11], [12]. However, they side-step the real problem – the large size of the integrity-tree. For example, protecting a 16 GB memory with an SGX-like integrity-tree design would require a tree as large as 292MB, that is hardly cacheable on-chip. Therefore, in this paper we explore compact integrity tree designs with higher cacheability, to reduce memory accesses for integrity-tree traversal.

State-of-the-art integrity-trees are constructed over the entire footprint of the encryption counters [13]. A smaller footprint obtained by packing more encryption counters per cacheline, can shrink the base of the integrity-tree and reduce its size. Additionally, commercial designs like SGX use integrity-trees that are designed as a tree of counters [7], where the tree-arity (fan-in per node) is determined by the number of counters per cacheline-sized entry. As the arity dictates the ratio by which each tree-level reduces in size, packing more counters per tree-entry can further reduce the tree height. Thus, to enable compact integrity trees, we investigate counter organizations that can provide a large number of counters per cacheline.

Prior work [14] proposed split counters for encryption, to accommodate more counters per 64-byte cacheline. This design can pack up to 64 counters per cacheline, by sharing a large major counter among 64 smaller minor counters. The minor counter is incremented when the corresponding data cacheline is updated in memory, while the major counter is incremented when any minor counter overflows. A recent work [1] also proposed using a similar split counter design for integrity-tree counters, to pack more counters per cacheline-sized tree entry.

However, it is impractical to store more than 64 counters per cacheline by reducing minor counter size. This is because smaller counters can overflow frequently. On an overflow, all the n-minor counters in an entry are reset after incrementing the major counter. This requires extra memory reads and writes – for re-encrypting n-child data cachelines on an encryption counter overflow, and for updating hashes of n-child entries on an integrity tree counter overflow. For example, packing 128 counters per cacheline results in 3-bit minor counters that can overflow in just 8 writes. As each overflow requires 256 extra memory accesses, this can cause significant slowdown.

All existing counter organizations are limited to 64 counters per cacheline, as they statically provision an equal number of bits for all the counters. However, we observe that applications utilize counters in distinct patterns which allow efficient counter

*This work was performed when Prashant J. Nair was affiliated with Georgia Institute of Technology.

designs that overflow less frequently. Leveraging this, we propose *Morphable Counters (MorphCtr)*, that dynamically changes the counter representation based on the usage pattern – allowing storing more than 64 counters in a cacheline while limiting the re-encryption overheads incurred due to overflows.

Our analysis of overflowing counters shows that applications either use less than a quarter of the counters in a cacheline or use all the counters in a cacheline. For example, usage of encryption counters depends on the write-intensity per data cacheline. As applications write uniformly to most cachelines within a write-heavy page, frequently overflowing encryption counters see uniform usage within a cacheline. However, usage of integrity-tree counters depends on write-intensity per page. As write-heavy and cold pages can be interspersed in memory, integrity-tree counters see sparse usage of few counters within a cacheline. Furthermore, higher levels of the tree do not suffer counter overflows, as writes do not propagate beyond the level of the tree that completely fits in the on-chip cache.

When 64 or less counters are used out of 128 counters in a cacheline, MorphCtr uses a representation called *Zero Counter Compression (ZCC)* that performs utility-based allotment of space to non-zero minor counters in a cacheline. ZCC uses a bit-vector to track the non-zero minor counters in a cacheline and distributes the rest of the bits in the cacheline only to those counters. For example, with 16 non-zero counters, each counter gets 16-bits while with 32 non-zero counters, each gets 8-bits and so on. Thus, MorphCtr provides large overflow-tolerant counters while packing 128 counters per cacheline, when at most a quarter of the counters within a cacheline are used.

When more than 64 out of 128 counters are used, MorphCtr uniformly allocates 3-bits per minor counter. In this regime, most workloads tend to use all the counters in a cacheline, with low dynamic-range in counters. Leveraging this, MorphCtr represents each minor counter in the cacheline as the sum of a common base and an offset. When any offset crosses its max-value, the counters are *re-based* instead of being reset, i.e. the base is moved forward by the smallest offset and all offsets are reduced by that value. This provides the largest offset room to grow, without changing other minor counter values, thus avoiding an overflow and subsequent re-encryption overheads.

Compared to split counters, MorphCtr provides a higher density of counters per cacheline, while incurring fewer overflows. Thus, it is a practical alternative to reduce storage and performance overheads of secure memories. For example, using MorphCtr for encryption can reduce the memory footprint of the encryption counters by 2x. This is because MorphCtr-128 packs 128 counters per cacheline as compared to split-counters with 64 counters per cacheline (SC-64) in the baseline. Additionally, as encryption counters form the base of the integrity-tree, this also reduces the integrity tree size by 2x.

Furthermore, using MorphCtr for the integrity-tree counters enables a compact 128-ary MorphTree. As the arity dictates the ratio by which each tree-level reduces in size, a 128-ary integrity tree has each level smaller by 2x compared to a 64-ary design obtained with SC-64 counters in the integrity tree. Thus,

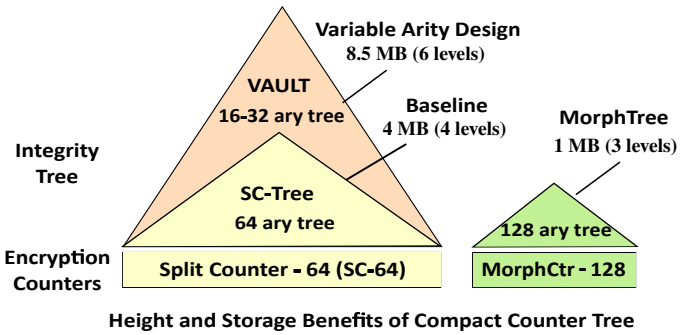

Fig. 1. Using MorphCtr-128 for encryption and integrity-tree results in a compact 128-ary integrity-tree (MorphTree) that allows low-overhead traversal. MorphTree is 4x smaller than Baseline and 8.5x smaller than VAULT.

using MorphCtr-128 for both encryption and integrity-tree results in a 4x smaller integrity-tree compared to our baseline which uses SC-64 for both, as shown in Figure 1. As a more compact tree is also better cacheable on-chip, MorphCtr-128 reduces the memory accesses for the integrity-tree traversal and improves performance compared to SC-64.

Overall, this paper makes the following contributions:

1) **Morphable Counters**, an overflow-tolerant counter design capable of high density of counters per cacheline, that is 2x more storage-efficient than split counters.
2) **MorphTree**, a 128-ary integrity-tree using MorphCtr. Combined with MorphCtr for encryption, this reduces integrity-tree size by 4x compared to SC-64 baseline.
3) **Zero Counter Compression**, a scheme for reducing the counter overflow frequency, when few counters in a cacheline are used – by compressing zero value counters and expanding the non-zero counters.
4) **Re-basing** instead of resetting minor counters, to avoid overflows and the subsequent re-encryption overheads, when all the counters in a cacheline are used.

We evaluate Morphable Counters with 28 memory intensive workloads from SPEC2006 and GAP benchmark suites and compare our design with a baseline using split counters with 64 counters per cacheline (SC-64) for both encryption and integrity tree. Using MorphCtr-128 improves performance by 6.3% on average (up to 28.3%) and reduces energy-delay product by 8.8% on average. These benefits come without any extra storage or reduction in security and stem from a compact integrity-tree design enabled by merely interpreting counters differently. For a 16 GB memory, the SC-64 baseline requires an integrity tree that is 4MB in size (4 levels), while our 128-ary tree using MorphCtr is only 1MB (3 levels).

Compared against VAULT [1], a recent proposal using split counters in the integrity-tree, MorphCtr-128 provides speedup of 13.5% on average (up to 47.4%). VAULT uses a variable arity of split counters (16 or 32 counters per cacheline) to limit counter overflows, resulting in a large integrity tree (8.5MB size, 6 levels). We propose an orthogonal approach to reduce counter overflows, that allows a compact 128-ary integrity-tree (8.5x smaller, 3 levels shorter), improving performance.

II. Background and Motivation

We first provide background regarding secure memory design. We then describe the performance overheads with secure memories and motivate our compact integrity tree design.

A. Secure Memory Design

1) Attack Model: We assume that adversaries have physical access to the system. Similar to prior works [1], [9], we assume the processor to be within the trusted computing base, with the off-chip main-memory and memory-bus vulnerable to unauthorized reads, modification or replay attacks. In this context, providing memory security (like SGX) involves data confidentiality with counter-mode encryption, ensuring data integrity with Message Authentication Codes (MACs) and replay attack protection using integrity-trees.

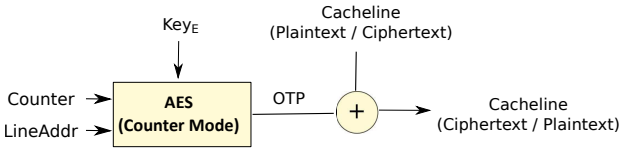

Fig. 2. Counter-Mode Encryption

2) Data Encryption with Counters: To prevent attackers from reading memory contents, data is encrypted with Counter Mode Encryption [14], [15]. As shown in Figure 2, a plaintext data cacheline is encrypted through an XOR with a One Time Pad (OTP). Similarly, decryption is done through an XOR of the cipher-text with the OTP. The OTP is a secret bit-string generated by passing a per-line counter through a block cipher like Advanced Encryption Standard (AES), with a secret key. The counter is incremented on each cacheline-write to ensure temporal variation in the encrypted data. While these counters are stored in the memory, they are cached on-chip [7], [11] to avoid extra memory accesses for the counter and allow the OTP to be pre-computed in parallel with the data-access.

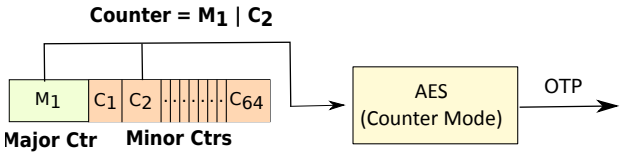

Fig. 3. Split counter design for counters

A *split counter* [14] design, further packs a large number of counters in a cacheline, to reduce the storage and the memory traffic of counters. This is achieved by encoding the counter value as a concatenation of a large major counter and a smaller minor counter, as shown in Figure 3. To avoid counter reuse[1] on a minor counter overflow, the major counter is incremented and all the minor counters in the cacheline are reset. This requires additional memory reads and writes to re-encrypt all the data cachelines associated with the minor counters. To limit the subsequent performance overhead, prior work [14] used a counter cacheline with 64 minor counters, sized such that they overflow and incur re-encryption overheads infrequently.

[1]Re-using counter values after a wrap-around is a security vulnerability, as an XOR of two cipher-texts using the same OTP can leak plain-text information.

3) Data Integrity with Authentication Codes: Encrypting the entire memory (e.g. AMD-SME [16]) still leaves it vulnerable to undetected data-tampering by an adversary. To prevent this, cryptographic signatures of cacheline contents called message authentication codes (MACs) are stored in memory per data cacheline. Prior designs use MACs generated by cryptographically hashing[2] data and encryption counter, using a secret key (e.g. AES-GCM [14], Carter-Wegman [7]). On every cacheline access, the memory controller fetches the stored MAC and verifies it by recomputing the MAC using the data cacheline and the counter, to ensure no tampering has occurred.

Recent work Synergy [10] avoids additional accesses for MACs, by storing MAC in the extra chip available in an ECC-DIMM organization, obtaining MAC and data in a single memory access. Additionally, it is possible to avoid extra accesses for both MAC and Error Correction Code (ECC) by storing a 10-bit Single-Error-Correction code (SEC)[3] with a 54-bit MAC[4] in the extra-chip (a concurrent work [19] makes a similar observation) – we use this design for all our evaluations.

4) Replay Attack Protection using Integrity-Trees: Memory contents protected with MACs are still vulnerable to tampering through replay attacks. For example, an adversary can replace a tuple of {*Data, MAC, Counter*} in memory with older values without detection. Integrity-trees [7], [13], [20] prevent replay attacks using multiple levels of MACs in memory, with each level ensuring the integrity of the level below. Each level is smaller than the level below, with the root small enough to be securely stored on-chip. We restrict our discussion to Bonsai-style counter trees [7], [13], a type of integrity-trees using counters, constructed with encryption counters as their base.

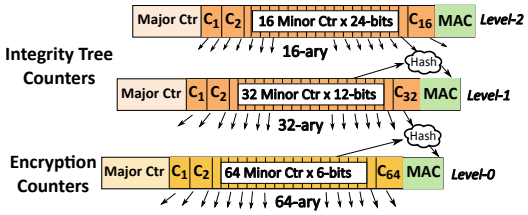

Fig. 4. Counter-tree based integrity tree design like VAULT [1].

Counter-Trees use hashing algorithms that generate MACs using integrity-tree counters. As shown in Figure 4, MACs protecting the integrity of encryption counters are co-located with the encryption counter cachelines (level-0) and generated using counters from the parent (level-1). Similarly, an integrity-tree counter cacheline (level-1) is hashed to produce a co-located MAC using a higher-level counter (level-2). The number of counters per cacheline-sized entry determines the arity, i.e. the ratio by which the size of each level reduces and decides the number of levels, which impacts the tree traversal overheads.

[2]Refer [10], [14] for more details regarding MAC implementation.

[3]Our analysis with FaultSim [17] shows system failure probability does not change whether we use 10-bit SEC per 64-byte cacheline or 8-bit SEC per 8-byte word (as in ECC-DIMM) – as the probability of two single-bit errors in the same cacheline is negligible compared to other failure modes [18]

[4]Forging a 56-bit MAC in SGX requires 2 million years [7], so forging a 54-bit MAC requires 500,000 years, that is much beyond system lifetime.

SGX uses an 8-ary counter tree with 8 counters per integrity tree entry. On the other hand, VAULT [1] uses split counters in the integrity-tree, with a 32-ary design at level-1 and 16-ary design at level-2 and beyond. Furthermore, the tree is constructed on an encryption counter base with a 64-ary split counter design. This variable-arity design ensures negligible frequency of counter overflows despite the higher write traffic at upper levels of the tree, given that writes propagate up the tree when lower level entries suffer dirty-evicts from on-chip cache. It is also possible to uniformly use a 64-ary split counter design across all the levels of the integrity-tree, to optimize for integrity-tree size rather than counter overflow frequency.

B. Performance Problem in Secure Memories

During secure execution, every memory access for data requires accessing the encryption counter. If the counter is not available in the on-chip cache and is fetched from memory, its integrity is verified by traversing the integrity-tree from the leaf to the root. The traversal continues accessing tree entries from memory until an entry is found securely cached on-chip. These extra memory accesses cause a memory-traffic bloat, that results in performance overhead during secure execution.

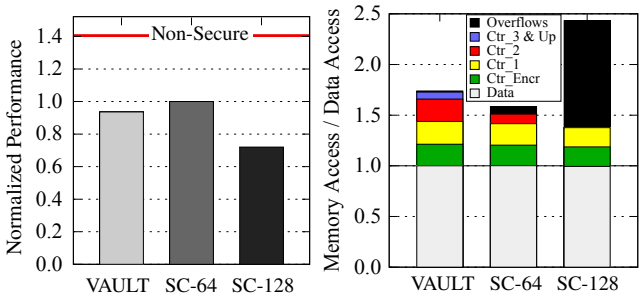

Fig. 5. Impact of increasing arity of counters used for encryption and integrity-tree. (a) Performance normalized to SC-64. (b) Memory traffic per data access.

Figure 5(a) compares the performance of secure execution as the arity of the counters used for encryption and integrity-tree changes across configurations. Each configuration is evaluated with 16 GB memory and 128KB metadata cache, using memory intensive workloads from SPEC2006 and GAP. All results are normalized to SC-64 that uses 64-ary counters throughout.

There exists a performance gap of 40% between secure execution with SC-64 and non-secure execution. To bridge this gap, scaling the integrity tree arity appears promising, but it has challenges. VAULT, a design using 64-ary encryption counters, 32-ary counters at level-1 of the integrity-tree and 16-ary at higher levels has 6.4% slowdown compared to SC-64 that is 64-ary throughout. However, further increasing the arity to 128 with SC-128 hurts performance, causing 28% slowdown.

This slowdown is caused by the additional memory accesses for fetching counters and handling counter overflows, as shown in Figure 5(b). VAULT incurs 0.7 additional accesses per data access for counters, while incurring negligible accesses due to counter overflows. SC-64 reduces the additional accesses for counters to 0.5 per data access, while incurring a modest 0.07 access per data access for handling counter overflows. While SC-128 further reduces accesses for counters to 0.4 per data access, it incurs a considerable penalty on account of counter overflows – almost 1 additional access per data access.

Increasing the integrity tree arity provides the benefit of fewer counter accesses. Higher arity trees are smaller in size, where fewer levels require memory accesses because of poor cacheability on-chip. For instance, VAULT incurs accesses for tree levels 1 to 3, while SC-64 requires accesses only for levels 1 and 2, and SC-128 requires accesses to just level-1.

However, increasing arity also causes higher re-encryption and re-hashing overheads. SC-64 and SC-128 incur more frequent counter overflows due to 6-bit and 3-bit minor counters, that can overflow in 64 and 8 writes respectively, whereas VAULT uses larger (12-bit or 24-bit) counters that overflow rarely. Furthermore, each overflow requires extra memory reads and writes to re-encrypt data or re-hash child tree entries, with the number of extra accesses being proportional to arity. SC-64 requires 64 reads and 64 writes per overflow, while SC-128 requires 128 reads and 128 writes per overflow. For SC-64, these overheads are limited compared to a reduction in counter accesses, resulting in a speedup compared to VAULT, but for SC-128 they are significant enough to cause a slowdown.

C. Goal: Compact Integrity Trees with Low-Overheads

All existing designs statically and equally size the counters within a cacheline. However, they are all limited to at most a 64-ary design, as uniformly smaller counters face the problem of frequent counter overflows and considerable re-encryption overheads. Fortunately, writes that are the root-cause of counter overflows, only propagate up the tree until the level that is completely resident in the on-chip cache. Studying the usage patterns of counters below this level can enable non-uniform counter organizations that are more tolerant to overflows. Thus, we investigate counter organizations that are both high density and overflow-tolerant, to get the benefit of compact integrity trees without the re-encryption and re-hashing overheads.

III. Morphable Counter Design

First, we analyze the problem of overflows in split counters and then explain our morphable counter design – how it incurs fewer overflows despite providing more counters per cacheline.

A. Overflow Problem with Split Counters

It is impractical to pack more than 64 counters per cacheline with split counters because small counters can overflow rapidly resulting in frequent re-encryptions. However, this overhead depends on the usage pattern of counters within a cacheline.

Figure 6 shows the "time to overflow", i.e. number of writes a counter-cacheline can tolerate before an overflow, for split counter designs with 64 x 6-bit minor counters (SC-64) and 128 x 3-bit minor counters (SC-128). Time to overflow varies as the fraction of counter-cacheline used changes (assuming uniform writes to the counters used). For example, split counters overflow rapidly if a small fraction of the counters in the cacheline see a majority of the writes. While SC-64 overflows

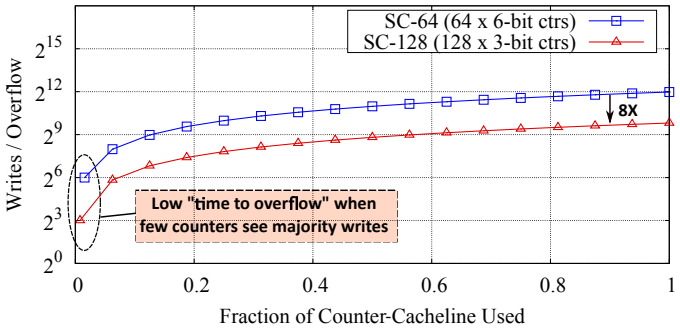

Fig. 6. Number of writes tolerated before an overflow, for split counters. Assuming uniform writes to the fraction of counters used in a cacheline.

every 64 writes in the worst-case, SC-128 can overflow with just 8 writes to the cacheline. Therefore, it is important to address this scenario while designing overflow tolerant counters building on the split counter design. In general, SC-128 design tolerates 8x lesser writes before an overflow compared to SC-64, because its counters are 3-bits smaller in size.

To better understand the application write patterns driving these counter overflows, we analyze the counter values at the time of overflow while using an SC-64 counter design for encryption and integrity-tree. Figure 7 shows the histogram of the "fraction of counter-cacheline used" when it encounters an overflow, averaged across 28 memory intensive workloads from SPEC2006 and GAP benchmark suites.

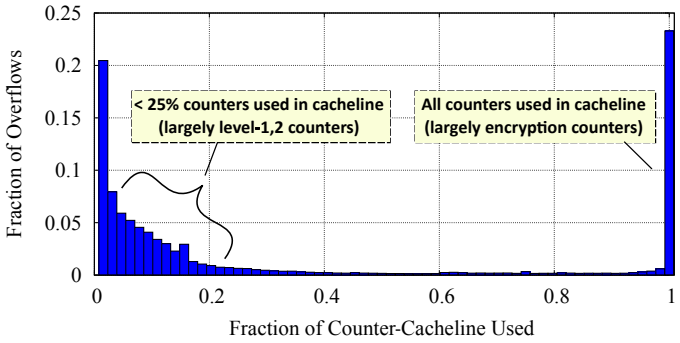

Fig. 7. Histogram of "fraction of counter-cacheline used" when an overflow occurs for SC-64, averaged over 28 workloads from SPEC2006 and GAP.

Most of the overflows in SC-64 occur either when applications sparsely use few counters in a cacheline, or when they use all the counters in the cacheline. For example, 27 out of 28 workloads we analyzed incurred more than 75% of overflows, either when they used less than a quarter of the counters in a cacheline, or when they used all the counters in a cacheline.

Sparse counter usage is common in level-1 counters, where the counter values depend on the frequency of writes to a physical page of data in memory. Because hot pages can be interspersed in memory with cold pages, there is a sparse utilization of counters within a cacheline at this level. On the other hand, encryption counters are more prone to uniform utilization especially in streaming applications, that uniformly write to all cachelines within a write-heavy page. Optimizing for these patterns of counter usage can help improve the write-tolerance of split counters and enable a high-density counter organization that is also tolerant to overflows.

B. Zero Counter Compression: For Sparse Counter Usage

We observe that when counters within a cacheline are used sparsely, few counters seeing intense writes drive the overflows, while many counters remain unused. Conventional counter designs equally allot bits to all counters irrespective of their usage. However, we observe that overflows may be reduced if zero counters are compressed to make space for larger sized non-zero counters. With this insight, we propose a Morphable Counter representation using *Zero Counter Compression (ZCC)* that enables utility-based sizing of counters.

1) Design: Figure 8 shows the counter cacheline organization with morphable counters using ZCC. The 512-bit counter cacheline is split into 4 fields: a 57-bit major counter, a 7-bit format field (specifying ZCC or Uniform), a 384-bit field storing the minor counters and a 64-bit MAC for the cacheline.

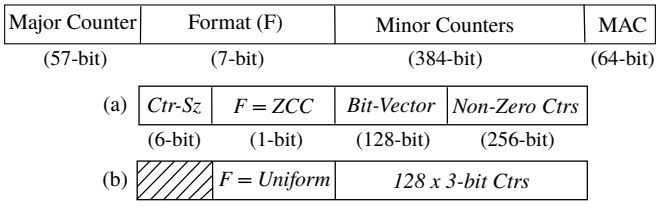

Fig. 8. Organization of Morphable Counter cacheline. Counters are represented in (a) Zero Counter Compression or (b) Uniform formats.

In ZCC-format, minor counter size depends on the number of non-zero counters. A 128-bit *Bit-Vector* tracks which counters are non-zero and the rest of the 256-bits are equally distributed only among the non-zero counters. The non-zero counter size is stored in the 6-bit *Ctr-Sz* field. In uniform-format, all the minor counters are uniformly sized as 128 x 3-bit counters.

ZCC permits large counter sizes when only a few counters in a cacheline are used. For example, up to 16 non-zero counters each counter gets 16-bits, up to 32 non-zero counters each gets 8-bits and so on (7-bits up to 36, 6-bits up to 42, 5-bits up to 51 and 4-bits up to 64). When more than 64 counters are used, the design adopts the uniform format without the bit-vector, allotting 3-bits for each of the 128 minor counters.

The counter value used for encryption or hash generation is obtained by adding the major and minor counter ($Counter = MajorCtr + MinorCtr$). On an overflow, to avoid re-using counter values, the major counter is incremented by the value of the largest minor counter in the cacheline ($MajorCtr \mathrel{+}= (Largest\_MinorCtr + 1)$) and all minor counters are reset to 0. While the major counter grows at a faster rate than in conventional split counter design, it is large enough to never overflow in system lifetime, as shown in Section V.

2) Operation: Counter cachelines are encoded with ZCC in both the on-chip cache and main-memory – decoding is required to obtain the counter value for encryption and decryption. Interpreting if a counter is zero only requires indexing into the bit-vector, as the corresponding bit is set to 0 for such counters. For a non-zero counter, the value can be obtained as shown in Figure 9(a), using the counter-size from the *Ctr-Sz* field and indexing into the *Non-Zero Ctrs*

field of the cacheline. As these are relatively simple operations compared to a cryptographic operation like AES required for encryption or decryption, decoding can be completed with negligible impact on the latency of encryption or decryption.

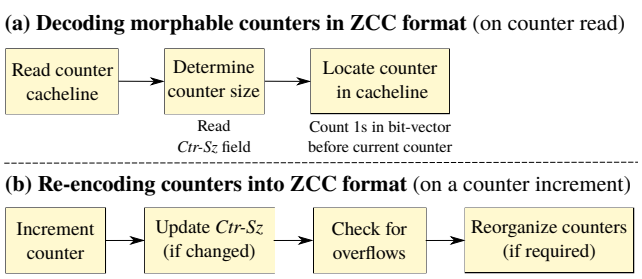

Fig. 9. Decoding and re-encoding morphable counters in ZCC format.

On a counter increment, a check is performed to ensure that it did not overflow. If the counter increment increased the number of non-zero counters, there is a possibility of reduction in the minor counter size. In this case, re-encoding the counters to ZCC requires re-organizing the *Non-Zero Ctrs* field to reflect the updated counter size as shown in Figure 9(b). However, as counter re-organization is performed infrequently and only on a write, its latency is not on the critical path of execution.

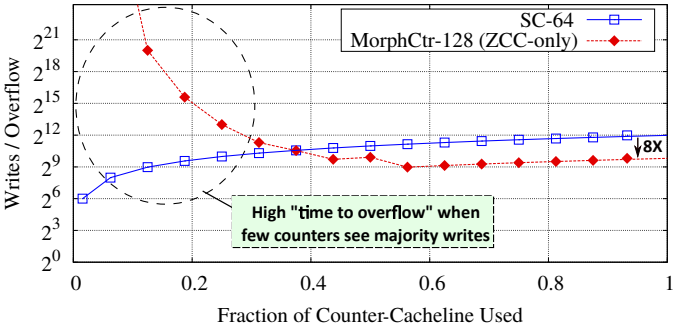

Fig. 10. Benefit in "time to overflow" with Morphable Counters using ZCC, compared to split counters. ZCC tolerates a higher number of writes before an overflow when less than 25% of counters in a cacheline are used.

3) Benefits: Figure 10 shows the number of writes before an overflow for morphable counters with MorphCtr-128 compared to SC-64. MorphCtr-128 has a higher "time to overflow" when at most a quarter of the counters in a cacheline are used. This is because when only a few counters in the cacheline are used, ZCC provides large counters by distributing the available bits among the non-zero counters. For example, when 32 or fewer counters are used, each counter gets at least 8-bits. On the other hand, SC-64 statically provisions 6 bits per minor counter. However, when a majority of the counters in a cacheline are used, MorphCtr-128 tolerates 8x lesser writes compared to SC-64, because it only provisions 3-bits per counter.

To evaluate benefits with real applications, we compare the overflows per million memory accesses for MorphCtr-128 against SC-64 and SC-128 configurations as shown in Figure 11. On average, SC-128 incurs 7.4x higher overflows compared to SC-64, whereas MorphCtr-128 incurs 1.4x fewer overflows than SC-64 and 10.2x fewer overflows than SC-128.

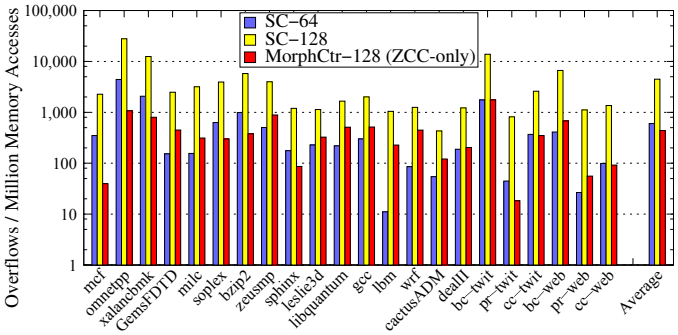

Fig. 11. Overflows per million memory accesses for SC-64, SC-128, and MorphCtr-128 using ZCC, for workloads from SPEC2006 and GAP.

ZCC considerably reduces the overflows of the sparsely used counters in the integrity-tree – therefore, MorphCtr-128 incurs fewer overflows than SC-128 for all workloads. Moreover, for workloads with inherently sparse data accesses (like *mcf, omnetpp, xalancbmk*), ZCC encoding for MorphCtr-128 helps reduce overflows of the encryption counters as well, resulting in fewer overflows compared to SC-64 for these workloads. For other workloads with streaming data accesses (like *libquantum, gcc, lbm*) that use a large fraction of the encryption counter-cacheline, ZCC is not as effective. This causes MorphCtr-128 to incur more overflows than SC-64 for streaming applications.

IV. MorphCtr - Handling Uniform Counter Usage

When a majority of the counters in a cacheline are used, MorphCtr-128 uses 3-bits per counter. As a result, it tolerates 8x fewer writes before an overflow, compared to SC-64 with 6-bit counters. Fortunately, workloads with such counter usage have uniform accesses (e.g. streaming workloads) and tend to use all the counters in a cacheline, without any zero counters. For such patterns, it is possible to avoid re-encryption overheads on a minor counter overflow – by *re-basing minor counters*, i.e. moving the major counter ahead by the smallest minor counter and reducing all minor counters by that value.

	Major Counter	Minor Counter (3-bit)				Effective Value (*major + minor*)			
Overflowing Minor Counter	100	5	6	8	7	105	106	108	107
After Overflow (Existing Design)	108	0	0	0	0	108	108	108	108
Avoiding Overflow With Rebasing	105	0	1	3	2	105	106	108	107

Fig. 12. Avoiding overflow and re-encryption with minor counter re-basing. Existing design resets all minor counters, requiring re-encryptions (all effective values changed); Re-basing only changes effective value of overflowing counter.

All existing designs incur re-encryption costs when a minor counter overflows, as they reset all the minor counters to 0 while incrementing the major counter. This changes the effective counter value for all counters in the cacheline as shown in Figure 12, requiring subsequent re-encryptions. However, if all the minor counters are non-zero, it is possible to re-base them, i.e. move the major counter ahead by the smallest minor counter and reduce all the minor counters by that value. This provides room for incrementing the previously overflowing counter, without changing the effective counter values for the other counters. Thus re-basing minor counters can avoid a counter overflow and also the associated re-encryption costs.

Minor Counter Re-basing (MCR) as described, is applicable to all existing counter designs up to 64 counters per cacheline. However, with 128 counters per cacheline, we observe a differing behavior among the two sets of 64 counters at the encryption counter level, given that each set corresponds to a different 4KB physical page. To support re-basing over two separate sets of 64 counters, we propose a double-base format for MCR in MorphCtr-128 – one base per set of 64 counters.[5]

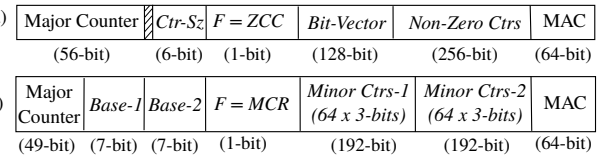

Fig. 13. Organization of morphable counter cacheline supporting (a) ZCC and (b) MCR (Minor Counter Re-basing) formats.

1) Design: Figure 13(b) shows the organization of a MorphCtr-128 cacheline in the MCR format. Each cacheline stores two 7-bit bases and two sets of 64 x 3-bit minor counters. The additional space required for the second base comes from storing a smaller 49-bit major counter, compared to a 56-bit major counter in ZCC format as shown in Figure 13(a). The effective counter value is a 56-bit value in both formats, i.e. $(MajorCtr + MinorCtr)$ for ZCC, and $((MajorCtr \,\|\, Base) + MinorCtr)$ for MCR.

2) Operation: When MorphCtr has more than 64 non-zero counters per cacheline, the format switches from ZCC to MCR. To ensure counter values are unchanged, initial values of *Base-1* and *Base-2* in MCR are the same as the lower 7-bits of *MajorCtr* in ZCC. In MCR, rebasing is attempted when a minor counter at the maximum value needs to be incremented – the base is moved forward by the smallest minor counter, and all the counters are decreased by the same value. Now the counter can be incremented without an overflow. If the smallest minor counter is zero, then re-basing is not possible – then all the counters in the set are reset to 0 and the base is incremented ($Base \mathrel{+}= (LargestMinorCtr + 1)$), accompanied by 64 re-encryptions. In case either base overflows, then both bases and all minor counters are reset to 0, the major counter is incremented by two[6] along with 128 re-encryptions, and the format switches back to ZCC.

3) Benefits: Rebasing reduces overflows in addition to the benefits obtained with ZCC, as shown in Figure 14. On average, ZCC+Rebasing reduces overflows by 1.6x compared to SC-64, while ZCC-only reduces by 1.4x. For many streaming workloads (like *gcc, lbm, libquantum*), ZCC+Rebasing significantly reduces overflows to a level similar as or below SC-64. However, for some outliers (e.g. *GemsFDTD*), ZCC+Rebasing incurs higher overflows as the counter usage pattern is neither sparse nor uniform. While our scheme is sub-optimal for such patterns, these patterns fortunately, occur rarely.

[5]For page sizes larger than 4KB (e.g. 8KB, 2MB etc.), a single-base design (using major counter as the base) works as well as the double-base design.

[6]$MajorCtr \mathrel{+}= 2$ ensures that the effective counter value at the time of simultaneous overflow of *Base* and *MinorCtr* is not re-used

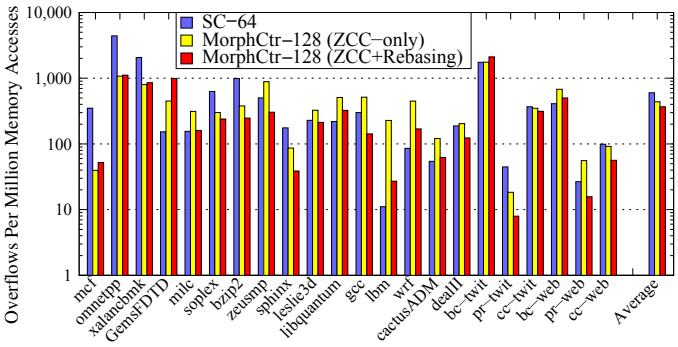

Fig. 14. Overflows per million memory accesses for SC-64 and MorphCtr-128 (ZCC-only and ZCC+Rebasing), for workloads from SPEC2006 and GAP.

V. Security Analysis

Ensuring no counter reuse: The security of encryption and integrity-checks is retained with morphable counters, as we only change the counter representation, without any modification to the encryption or MAC algorithm. By ensuring that the effective counter value is always forward-moving and never repeated, morphable counters prevents counter re-use. While the effective counter value can grow 2x faster than the minor counter (due to $MajorCtr \mathrel{+}= 2$), it is sufficiently large (56 bits) that it will not overflow in system lifetime (30+ years before overflow, with 1 counter increment per 100 CPU cycles).

Resilience to Denial of Service: Morphable counters can tolerate 500+ writes before an overflow, when counters are written uniformly as shown in Figure 10. However, a pathological write pattern can cause an overflow in 67 writes, by writing once to 52 counters out of 128 (reducing the counter size to 4-bits), followed by 15 writes to a single counter. In fact, the baseline split counter design is even more vulnerable, as it can overflow every 64 writes. Such frequent overflows can flood the memory system with accesses for re-encryptions and MAC updates. However, other programs can be shielded from any performance impact in this scenario with fairness-driven memory scheduling policies [21], [22], [23] that can throttle the overflow-handling accesses of the pathological application and maintain serviceability of other applications.

Potential side-channels: Morphable counters do not leak any information in addition to existing side-channels. While overflow frequency can reveal information about memory access patterns, this is available by monitoring address-bus or plaintext counter-values even in baseline. Data-confidentiality is further unaffected, as only counter-encoding is changed, with counter-values in counter-mode encryption of data unchanged.

VI. Experimental Methodology

Simulation Framework: We use USIMM [24], a memory system simulator for evaluating system performance and power. USIMM enforces the JEDEC DDR3 protocol and uses power parameters from industrial 4Gb x8-DRAM chips to accurately model memory power. For our simulations, we warm-up the counters for 25 billion instructions[7] before

[7]We verify that counter overflows have stabilized by comparing the counter overflow rate with simulations for 50 billion instructions.

measuring performance over 5 billion instructions. We evaluate performance by comparing Instructions Per Cycle (IPC) for our proposal against baseline design. We also characterize the bloat in memory traffic due to secure execution by evaluating the ratio of total memory accesses to data accesses.

TABLE I
BASELINE SYSTEM CONFIGURATION

Number of cores	4
Processor clock speed	3.2GHz
Processor ROB size	192
Processor fetch / retire width	4
Last Level Cache (Shared)	8MB, 8-Way, 64B lines
Metadata Cache (Shared)	128KB, 8-Way, 64B lines
Memory size	16 GB
Memory bus speed	800MHz
Banks x Ranks x Channels	8 x 2 x 2
Rows per bank	64K
Columns (cache lines) per row	128
OS Page Allocation Policy	Random

Secure memory model: Our baseline (SC-64) uses split-counters [14] with a 64-ary design for both encryption and integrity tree, resulting in a 64-ary integrity-tree. We also compare with VAULT [1], that has a variable arity of counters – 64-ary split counters for encryption, 32-ary for level-1 of the integrity tree and 16-ary for upper levels. Our proposal uses 128-ary MorphCtr-128 (ZCC + Rebasing), henceforth referred as just MorphCtr-128, for both encryption and integrity-tree. All configurations assume a Synergy [10] configuration, without any overhead for accessing MACs. Similar to recent works [1], [9], we use a 128KB dedicated metadata cache shared across 4 cores, for caching counters (encryption and integrity-tree).

TABLE II
(I) MEMORY ACCESSES PER KILO INSTRUCTIONS (PKI) PER CORE, (II) MEMORY FOOTPRINT FOR 4 CORES.

Suite	Workload	Read-PKI	Write-PKI	Footprint (GB)
SPEC2006	mcf	69	2	7.5
	omnetpp	18	9	0.6
	xalancbmk	4	3	1.1
	GemsFDTD	19	8	3.1
	milc	19	7	2.3
	soplex	28	6	1.0
	bzip2	5	1.4	1.2
	zeusmp	5	1.9	1.9
	sphinx	14	1.4	0.1
	leslie3d	16	5	0.3
	libquantum	24	10	0.1
	gcc	48	53	0.7
	lbm	28	21	1.6
	wrf	4	2	1.6
	cactusADM	5	1.5	1.6
	dealII	1.7	0.5	0.2
GAP	bc-twit	61	24	9.3
	pr-twit	94	4	11.2
	cc-twit	89	7	7.0
	bc-web	13	7	12.0
	pr-web	16	3	12.2
	cc-web	9	1.5	7.8

Workloads: We evaluate our design using workloads from SPEC2006 [25] and GAP [26] benchmark suites. As our proposal optimizes memory accesses, we focus on memory-intensive workloads from SPEC2006 (>1 memory access per 1000 instructions). From GAP, we use 6 important workloads (Page Rank, Connected Components, Betweenness Centrality kernels with Twitter and Web data-sets). We run the benchmarks in rate mode, i.e. each of the four cores running the same copy of the benchmark. Additionally, we evaluate 6 mixed workloads obtained with a random combination of benchmarks.

VII. RESULTS & DISCUSSION

A. Impact on Performance

Figure 15 compares the performance of MorphCtr-128 (using ZCC and Rebasing) with SC-64 and VAULT, all normalized to SC-64 (baseline). While VAULT suffers slowdown of 6.4%, MorphCtr-128 achieves a speedup of 6.3%, compared to SC-64.

Performance depends on the size of the integrity-tree in each configuration. A more compact integrity-tree has better cacheability of entries on-chip, resulting in lower overheads due to integrity-tree traversal and better performance. The baseline SC-64 using 64-ary counters has an integrity-tree that is 4 MB in size. On the other hand, VAULT requires a larger 8.5 MB integrity-tree because it uses lower arity (16 or 32-ary) split counters in the integrity-tree, while using 64-ary split counters for encryption. As a result, VAULT suffers slowdown of 6.4%.

In comparison, MorphCtr-128 uses 128-ary counters, that reduce the footprint of encryption counters by 2x. Consequently, the area on which the integrity tree is constructed is smaller by 2x. Additionally, each level of the tree is smaller by 2x as the tree-arity is double that of SC-64. This reduces the tree size additionally by 2x. In combination, this results in an integrity-tree size of 1 MB, that is 4x smaller in size than SC-64. This is the main driver behind the speedup of 6.3%.

The benefits of MorphCtr-128 are more pronounced for workloads like *mcf, omnetpp*, and *xalancbmk* with high memory traffic and random data accesses. These workloads incur considerable memory traffic for integrity-tree traversal in the baseline because there is limited reuse of the cached tree-entries without temporal locality in data accesses. In such scenarios, the compact tree design with MorphCtr-128 provides considerable speedup by reducing the number of levels to be traversed. Similar speedup is also seen with graph workloads from GAP with the Twitter dataset (*bc-twit, pr-twit, cc-twit*), which perform random accesses across large working sets.

For workloads with high spatial locality like *libquantum*, *gcc* etc., MorphCtr-128 performs as good as the baseline. This is because these workloads have limited integrity-tree traversal with high spatial re-use of cached counters. Similarly, there is no impact on performance for non-memory intensive workloads, with infrequent memory accesses and integrity-tree traversal.

B. Analyzing memory traffic bloat

We analyze the memory traffic bloat due to the accesses for the counters and handling their overflows, which is the main driver of performance overheads. Figure 16 shows the memory traffic bloat (memory accesses per data access) for VAULT, SC-64, and MorphCtr-128. The memory traffic is split into accesses for (1) program-related data, accesses to multiple levels of counters, i.e. (2) Ctr_Encr used for encryption, (3-5) Ctr_1, Ctr_2, Ctr_3 & Up (different levels of the integrity tree) and (6) Overflow handling – memory accesses for re-encryption of data and updates to MACs on counter overflows.

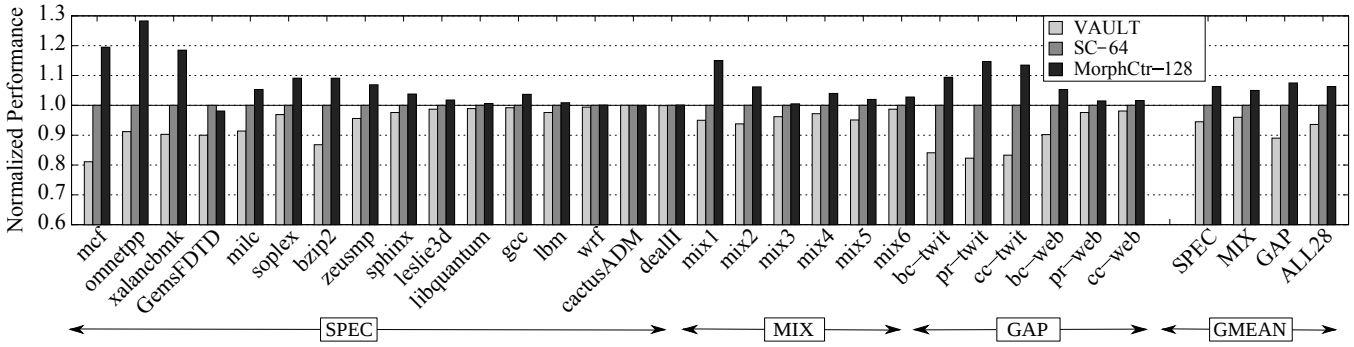

Fig. 15. Performance (IPC) of morphable counters (MorphCtr-128) compared with SC-64 (baseline) and VAULT, all normalized to SC-64. MorphCtr-128 improves performance by 6.3% using a more compact integrity-tree, while VAULT suffers slowdown of 6.4% due to a larger integrity tree.

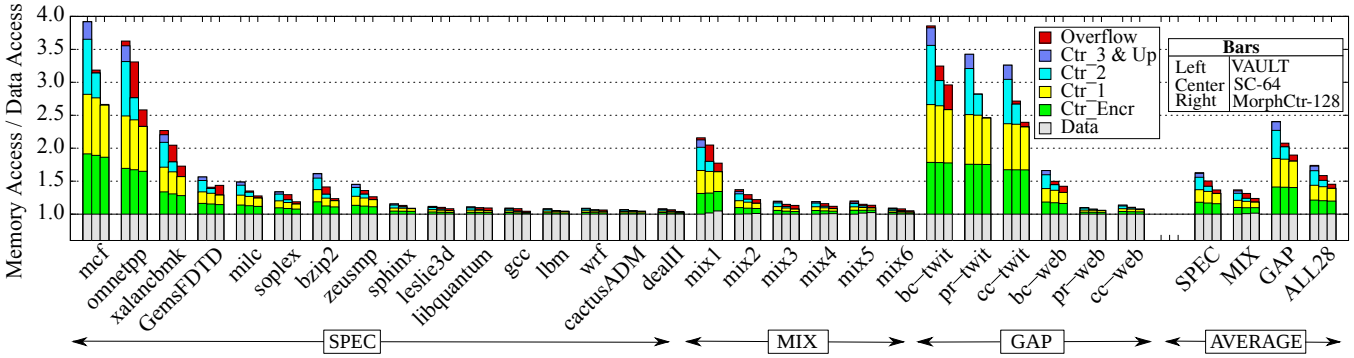

Fig. 16. Memory traffic for morphable counters (MorphCtr-128), compared with SC-64 and VAULT, normalized to the data traffic. MorphCtr-128 with a 3-level integrity tree reduces the memory traffic by 8.8% compared to SC-64 (4-level tree), while VAULT (6-level tree) incurs 9.7% higher traffic compared to SC-64.

MorphCtr-128 provides the benefit of fewer memory accesses for metadata, requiring 0.5 extra accesses per data access compared to SC-64 that needs 0.6 accesses. This reduction is due to the compact integrity-tree in MorphCtr-128, that is one level shorter and better cacheable on-chip compared to SC-64. While SC-64 require 0.5 counter accesses per data access (accessing Ctr_Encr, Ctr_1 and Ctr_2), MorphCtr-128 only requires 0.4 accesses (accessing only Ctr_Encr and Ctr_1).

Additionally, overflowing counters result in extra memory accesses for re-encrypting data and updating MACs in child tree-entries. MorphCtr-128 incurs 0.07 accesses per data access for handling overflows, similar to SC-64 (0.06 accesses), despite having 2x arity and consequently, 2x memory accesses required per overflow. This is because of a reduction in the frequency of counter overflows by 1.6x in MorphCtr-128 compared to SC-64, due to a combination of ZCC and Counter-Rebasing.

Random access workloads (like *mcf* and *omnetpp*) see benefits of both higher arity tree of MorphCtr-128 and higher overflow tolerance of ZCC, resulting in a reduction in accesses for counters and overflow handling. On the other hand, uniform access workloads (like *libquantum* and *gcc*) have limited potential for reduction in counter accesses and do not benefit from higher arity of MorphCtr-128. However, they do not suffer the overflow handling overheads of smaller counters either, as Counter Rebasing ensures low overflow frequency.

Only *GemsFDTD* suffers a slowdown (2%) with MorphCtr-128, as the reduction in counter memory accesses is offset by a larger increase in accesses for handling overflows. Here, both ZCC and Rebasing are unable to limit the overflow frequency, as the counter usage pattern is neither sparse nor uniform.

C. Understanding Slowdown of VAULT

As shown in Figure 15, VAULT suffers a slowdown of 6.4% compared to our baseline SC-64. This is because of 9.7% higher memory traffic, as shown in Figure 16. VAULT uses conservative low-arity split counters in the integrity tree (32-ary counters for level-1, 16-ary for upper levels), that result in a large 8.5 MB tree with 6 levels. While these counters are sized such that the overflow handling overheads are negligible, the resulting integrity-tree has many levels requiring multiple accesses during traversal. Thus, VAULT requires an additional 0.74 accesses per data access for counters, while only incurring 0.01 accesses per data accesses for handling overflows. On the other hand, SC-64 incurs lesser memory traffic by trading off a slight increase in accesses for overflow handling (0.07 accesses per data access) for much fewer counter accesses (0.5 accesses per data access).

D. Understanding Overflow Tolerance of MorphCtr-128

Using a high-arity counter design in the integrity-tree may seem counter-intuitive, given how these entries protect a large span of memory. However, the ZCC format in MorphCtr-128 is able to provide the illusion of larger counters despite a high-arity organization, by exploiting the fact that less than 25% of the counters in these integrity-tree entries are actually used. Furthermore, the upper levels of the tree do not have frequent counter overflows, as writes do to propagate beyond the tree-level that is completely resident in cache. When streaming access patterns cause ZCC to fail, MCR format leverages the uniform counter usage that is common with these access patterns to avoid overflows in MorphCtr-128.

E. Impact on Height of the Integrity-Tree

Figure 17 shows the number of levels in the integrity-tree, using MorphCtr-128 compared with VAULT and SC-64, for a system with 16GB memory. VAULT has counters with variable arity, using 64-ary counters for encryption, 32-ary counters at level-1 of the integrity-tree and 16-ary counters at higher levels. As a result, it has a large integrity tree with 6 levels. With higher arity counters, the number of levels in the integrity-tree decreases, thus reducing the worst-case overhead of integrity-tree traversal. For example, SC-64 (64-ary throughout) has 4 levels, whereas MorphCtr-128 (128-ary) only has 3 levels. Additionally, both 64-ary and 128-ary designs have smaller footprints at each level of the tree, resulting in better cacheability of tree-entries and fewer memory accesses for integrity-tree traversal on average.

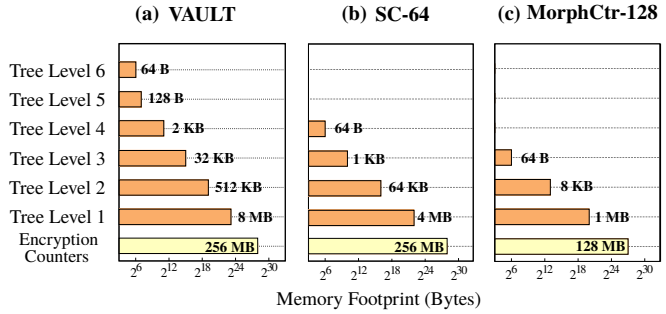

Fig. 17. Reduction in number of integrity tree levels as arity increases. VAULT a variable-arity tree (16 to 32-ary) has 6 levels, SC-64 (64-ary) has 4 levels, MorphCtr-128 (128-ary) only has 3 levels.

F. Impact on Storage Overheads

The higher arity design of MorphCtr-128 reduces the overall storage overheads for encryption counters and integrity-tree compared to SC-64, as shown in Table III. SC-64 with a 64-ary design throughout, incurs 1.6% storage overhead for its encryption counters (1/64 of data footprint) and 0.025% for the integrity-tree (approximately 1/64 of encryption counter footprint). As MorphCtr-128 provides a 128-ary design, it requires 2x lesser storage, incurring 0.8% overhead for the encryption counters. In addition, the integrity-tree with MorphCtr-128 is 4x smaller, because of multiplicative benefits of smaller encryption counter base and higher arity in the integrity-tree. In comparison, VAULT has a higher integrity-tree overhead (8.5x larger than MorphCtr) due to its conservative integrity-tree arity, whereas its encryption counter overhead is similar to SC-64. Commercial-SGX has much higher storage overheads in comparison to these designs, because of its 8-ary counter design for both encryption and integrity-tree counters.

TABLE III
STORAGE OVERHEADS FOR 16GB MEMORY.

Configuration	Storage Overheads	
	Encryption Counters	Integrity-Tree
Commercial-SGX	2 GB (12.5%)	292 MB (1.8%)
VAULT	256 MB (1.6%)	8.5 MB (0.05%)
SC-64	256 MB (1.6%)	4 MB (0.025%)
MorphCtr-128	128 MB (0.8%)	1 MB (0.006%)

G. Impact on System Power and Energy

MorphCtr reduces system energy, incurring fewer memory accesses that consume energy. Figure 18 shows the power, execution time, energy, and Energy-Delay Product (EDP).

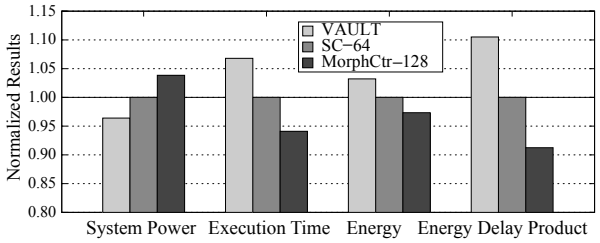

Fig. 18. Power, Execution Time, Energy and EDP for VAULT, SC-64 and MorphCtr-128, normalized to SC-64.

MorphCtr-128 reduces execution time by 6% compared to SC-64. However, it incurs 4% higher power consumption as it performs the same work in a shorter time. Despite the increased power, it provides energy savings of 2.7% due to the reduced execution time and improves the system energy-delay product (EDP), which is a product of energy and execution time, by 8.8%. On the other hand, VAULT suffers from 3.2% higher energy and 10.5% higher EDP compared to SC-64.

H. Sensitivity to metadata cache size

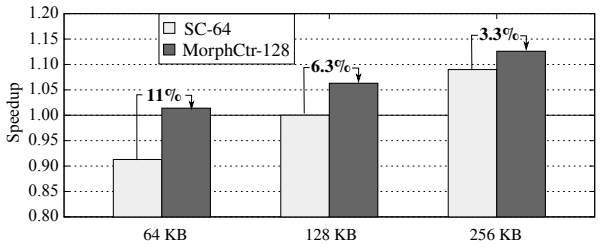

Fig. 19. Performance of SC-64 and MorphCtr-128 as metadata cache varies from 64KB to 256KB, normalized to SC-64 with 128KB metadata cache.

Figure 19 shows speedup with MorphCtr-128 vs SC-64, as the metadata cache size varies. Commercial systems design the metadata cache as a part of the memory controller [7], making a large metadata cache difficult due to area constraints. With smaller metadata caches, there is a larger memory access bloat for counters – hence MorphCtr-128 provides higher speedup as the cache size decreases. Thus, MorphCtr-128 provides a speedup of 3.3% with 256KB cache, 6.3% with 128KB cache and 11% with 64KB cache. In fact, MorphCtr requires half the metadata cache, to provide equivalent performance as SC-64.

I. Sensitivity to MAC organization

For data integrity, secure memories store and access MACs of data cachelines. In this paper, we assume a MAC organization like Synergy [10] that stores *In-Line MACs* and provides MAC in the same access as data. However, an unoptimized design could store MACs separately (*Separate MACs*) like other prior works [7], incurring an extra memory access for MAC on each data access. Figure 20 compares the performance of SC-64 and MorphCtr-128 using Separate MACs and In-Line MACs (i.e. Synergy), all normalized to SC-64 with In-Line MACs.

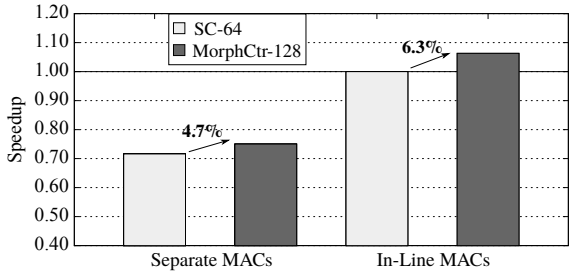

Fig. 20. Performance of SC-64 and MorphCtr-128 for Separate-MACs and In-Line MACs configurations, normalized to SC-64 with In-Line MACs.

In Separate MACs configuration, MACs cause significant memory access bloat in addition to counters. Therefore, both SC-64 and MorphCtr-128 suffer slowdown of 29% compared to similar designs with In-Line MACs. As MorphCtr-128 only reduces the memory traffic bloat due to counters compared to SC-64, it provides a smaller speedup of 4.7% in Separate MACs configuration, in contrast to speedup of 6.3% in In-Line MACs configuration. Other proposals optimizing MAC accesses [1], [11], [12] would see speedups in this range.

VIII. Related Work

To our knowledge, this is the first work proposing compact representations for encryption and integrity-tree counters, to reduce overheads of integrity-tree traversal and counter overflows. A concurrent work [19] proposes delta-encoding for encryption counters, but only in the context of reducing overflows. Prior works have explored alternate encoding of data (compression) in the memory-system and optimizing integrity-tree traversal in secure memories – we discuss these below.

A. Compressed memory systems

1) Compressed caches and main-memory: Prior works have exploited data patterns, to propose compression for caches [27], [28], [29], [30], [31], main-memory [32], [33], [34], [35], [36], [37] and even 3D-DRAM [38], [39] – to unlock additional storage, bandwidth or energy savings.

Our work does not depend on data values and hence is orthogonal to all of these prior works. Instead, we propose alternate representations for counters, whose values depend on write patterns of data. In fact, our proposal works even for an application writing random data values to its working set. Furthermore, compression of data in cache or main-memory may be used along with our proposal to enjoy additive benefits.

2) Low-latency compression algorithms: FPC [40] uses a dictionary-based approach for compressing small values in 32-bit words. DZC [41] represents zero value cachelines with a single-bit, to avoid expending energy for reading all the bits. BDI [42] represents the cacheline as a base value and an array of deltas, exploiting low-dynamic range in the data values. BPC [43] uses a bit-plane transformation on 32-bit words to improve the compression ratio with other schemes.

In a similar spirit, we use counter encoding that exploits sparse or uniform patterns in counter values. However, while prior proposals aim to reduce space occupied by data, we store larger counters in the same space to avoid overflows.

B. Integrity Verification in Secure Memories

1) Alternate integrity-tree designs: All counter based trees like TEC-Tree [44], Parallelizable Authentication Trees [45], SGX Tree [7] and VAULT [1] have tree arity dependent on the number of counters or nonces per tree entry. Using morphable counters to obtain more counters per entry can enable higher tree-arity with minimal overheads for these designs.

On the other hand, MAC-Trees or Merkle Trees [13] that are constructed as a tree of MACs are limited to 8-ary irrespective of the counter design used. This is because the arity depends on the number of MACs per tree-entry and only 8 x 64 bit-MACs can fit in a cacheline sized entry. Smaller 32-bit MACs that provide a higher arity, do not provide sufficient security.

2) Proposals optimizing integrity-tree traversal: Prior works [11], [12], [46] have proposed caching integrity-tree entries in the last-level cache along with data or using metadata type-aware replacement policies for efficient caching. These proposals are orthogonal to our work as they do not address the size of the integrity-tree which we focus on. Combining these with our compact integrity tree design can ensure low-overhead integrity-tree traversal as memories scale to larger sizes.

Alternate designs push integrity-tree traversal off the critical path, with counter value prediction [47] or speculative usage of unverified counters (e.g. PoisonIvy [9], ASE [48]). However, they only address the latency overheads of integrity-verification and still incur the bandwidth overheads. Whereas, our design also reduces the bandwidth overheads with a compact integrity-tree and can be combined with these proposals.

Recent smart-memory solutions [49], [50] provide low-overhead replay-attack protection, but require custom memory modules. In contrast, our proposal is compatible with commercial approaches for securing commodity DRAM like SGX.

IX. Conclusion

In this era of cloud computing, remote data-centers store sensitive information like credit card details, bitcoin keys, etc. in main memory. While it is critical to protect such data in memory from adversaries, it is also important to ensure that the security mechanisms have low overhead to facilitate adoption.

In this paper, we enabled a secure memory design with morphable counters, a compact 128-ary counter organization that requires lesser storage compared to all prior works that are limited to at most 64-ary. Using morphable counters, we designed a compact integrity-tree design that is more amenable to caching, improving performance by 6.3% compared to our 64-ary baseline and by 13.5% compared to VAULT. These benefits come without any extra storage or reduction in security and are derived from re-designing the counter organization that reduces the storage overhead of secure memory.

X. Acknowledgment

We thank Roberto Avanzi and Milos Prulovic for their technical inputs, Sanjay Kariyappa and Poulami Das for helpful discussions, and the anonymous reviewers of ISCA-2018 and MICRO-2018 for their valuable feedback that helped improve this paper. This work was supported by NSF Grant 1526798.

REFERENCES

[1] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures," in *ASPLOS*, 2018.

[2] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *CACM*, 2009.

[3] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin, "Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors," in *HPCA*, 2017.

[4] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ISCA*, 2017.

[5] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," *Black Hat*, 2015.

[6] M. Becher, M. Dornseif, and C. N. Klein, "FireWire: all your memory are belong to us," *Proceedings of CanSecWest*, 2005.

[7] S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors," *IACR Cryptology ePrint Archive*, 2016.

[8] V. Costan and S. Devadas, "Intel SGX Explained." *IACR Cryptology ePrint Archive*, 2016.

[9] T. S. Lehman, A. D. Hilton, and B. C. Lee, "PoisonIvy: Safe speculation for secure memory," in *MICRO*, 2016.

[10] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "SYNERGY: Rethinking Secure-Memory Design for Error-Correcting Memories," in *HPCA*, 2018.

[11] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *HPCA*, 2003.

[12] J. Lee, T. Kim, and J. Huh, "Reducing the Memory Bandwidth Overheads of Hardware Security Support for Multi-Core Processors," *IEEE Trans. Comput*, 2016.

[13] 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 *MICRO*, 2007.

[14] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving Cost, Performance, and Security of Memory Encryption and Authentication," in *ISCA*, 2006.

[15] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors," in *MICRO*, 2003.

[16] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," *White paper*, 2016. [Online]. Available: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[17] P. J. Nair, D. A. Roberts, and M. K. Qureshi, "FaultSim: A Fast, Configurable Memory-Reliability Simulator for Conventional and 3D-Stacked Systems," in *ACM-TACO*, 2015.

[18] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *SC 2012*.

[19] S. F. Yitbarek and T. Austin, "Reducing the overhead of authenticated memory encryption using delta encoding and ECC memory," in *DAC*, 2018.

[20] R. C. Merkle, "Protocols for Public Key Cryptosystems," in *S&P (Oakland)*, 1980.

[21] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *MICRO*, 2010.

[22] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX Security*, 2007.

[23] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," in *ACM Sigplan Notices*, vol. 45, no. 3, 2010.

[24] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "Usimm: the utah simulated memory module," *University of Utah, Tech. Rep*, 2012.

[25] "SPEC CPU2006 Benchmark Suite," in *Standard Performance Evaluation Corporation*. [Online]. Available: http://www.spec.org/cpu2006/

[26] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[27] J. Yang and R. Gupta, "Energy efficient frequent value data cache design," in *MICRO*, 2002.

[28] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *MICRO*, 2000.

[29] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *ISCA*, 2004.

[30] J.-S. Lee, W.-K. Hong, and S.-D. Kim, "Design and evaluation of a selective compressed memory system," in *ICCD*, 1999.

[31] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *SC*, 2009.

[32] B. Abali, H. Franke, D. E. Poff, R. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith, "Memory expansion technology (MXT): software support and performance," *IBM JRD*, vol. 45, 2001.

[33] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *ISCA*, 2005.

[34] G. Pekhimnko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: a low-complexity, low-latency main memory compression framework," in *MICRO*, 2013.

[35] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "MemZip: Exploring unconventional benefits from memory compression," in *HPCA*, 2014.

[36] J. Kim, M. Sullivan, S.-L. Gong, and M. Erez, "Frugal ecc: Efficient and versatile memory error protection through fine-grained compression," in *SC*, 2015.

[37] D. J. Palframan, N. S. Kim, and M. H. Lipasti, "COP: To Compress and Protect Main Memory," in *ISCA*, 2015.

[38] S. Kim, S. Lee, T. Kim, and J. Huh, "Transparent Dual Memory Compression Architecture," in *PACT*, 2017.

[39] V. Young, P. J. Nair, and M. K. Qureshi, "DICE: Compressing DRAM Caches for Bandwidth and Capacity," in *ISCA*, 2017.

[40] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for L2 caches," *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, vol. 1500, 2004.

[41] L. Villa, M. Zhang, and K. Asanović, "Dynamic zero compression for cache energy reduction," in *MICRO*, 2000.

[42] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *PACT*, 2012.

[43] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-plane compression: Transforming data for better compression in many-core architectures," in *ISCA*, 2016.

[44] R. Elbaz, D. Champagne, R. B. Lee, L. Torres, G. Sassatelli, and P. Guillemin, "Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks," in *CHES*, 2007.

[45] W. E. Hall and C. S. Jutla, "Parallelizable authentication trees," in *SAC*, 2005.

[46] T. S. Lehman, A. D. Hilton, and B. C. Lee, "MAPS: Understanding Metadata Access Patterns in Secure Memory," in *ISPASS*, 2018.

[47] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, "High Efficiency Counter Mode Security Architecture via Prediction and Precomputation," in *ISCA*, 2005.

[48] W. Shi and H.-H. S. Lee, "Authentication Control Point and Its Implications For Secure Processor Design," in *MICRO*, 2006.

[49] S. Aga and S. Narayanasamy, "InvisiMem: Smart Memory Defenses for Memory Bus Side Channel," in *ISCA*, 2017.

[50] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories," in *ISCA*, 2017.