Eager Memory Cryptography in Caches

Xin Wang[§], Jagadish B. Kotra[‡], Xun Jian[§]
[§]*Virginia Tech* [‡]*AMD Research*
xinw@vt.edu, Jagadish.Kotra@amd.com, xunj@vt.edu

***Abstract*—To protect memory values from adversaries with physical access to data centers, secure memory systems ensure memory confidentiality and integrity via memory encryption and verification. The corresponding cryptography calculations require a memory block's write counter as input. As such, CPUs today cache counters in the memory controller (MC).**

Due to the large memory footprint and irregular access patterns of many real-world applications, MC's counter cache is too small to achieve high hit rate. A promising solution is also caching counters in the much bigger Last Level cache (LLC). As such, many prior works use LLC as a second level cache for counters to back up the smaller counter cache in MC.

Caching counters in LLC introduces a new problem, however. Modern server CPUs have a long LLC access latency that not only can diminish the benefit of caching counters in LLC, but also can sometimes significantly increase counter access latency compared to not caching counters in LLC.

We note the problem lies with MC sitting behind LLC; due to its physical location, MC can only see LLC misses and, therefore, can only serially access and use counters after data miss in LLC has completed. However, prior designs without caching counters in LLC can access and use counters in parallel with accessing data. If a block's counter misses in MC's counter cache, MC can fetch the counter from DRAM in parallel with data; if the counter hits in MC's counter cache, MC can use counters for cryptography calculation in parallel with data traveling from DRAM to MC.

To parallelize the access and use of counters with data access while caching counters in LLC, we observe that in modern CPUs, L2 is typically the first place that caches data from DRAM (i.e., L2 and L3 are non-inclusive); as such, data from DRAM need not be decrypted and verified until they reach L2. So it is possible to offload some decryption and verification tasks from MC to L2. Since L2 sits before L3, L2 can access counter and data in parallel from L3; L2 can also use counters for cryptography calculation in parallel with data traveling from DRAM to L2, instead of just from DRAM to MC. As such, we propose caching and using counters directly in L2 and refer to this idea as Eager Memory Cryptography in Caches (EMCC). Our evaluation shows that when applied to the state-of-the-art baseline, EMCC improves performance of large and/or irregular workloads by 7%, on average.

***Keywords*-memory encryption and verification; counter-mode AES; cache hierarchy; network-on-chip**

I. Introduction

Due to the low cost of Cloud computing, companies are moving more and more computation from onsite to Cloud. However, moving to Cloud raises security concerns. Companies can no longer control physical accesses to the computing facilities; this opens up the possibility of physical attacks, in which malicious personnel with physical access can steal sensitive application data and/or tamper with them.

To improve memory security, secure memory systems ensure confidentiality and integrity in memory. Confidentiality refers to hiding data values from attackers; integrity refers to securely detecting malicious data tampering. Specifically, when writing back data to DRAM, CPU encrypts the data to ensure memory confidentiality and protects the data with a message authentication code (MAC) to ensure integrity; when fetching data from DRAM, CPU decrypts the data and verify integrity via the MAC.

Both decryption and verification take as input a block's write counter, which is also simply called the block's counter. Memory controller (MC) stores counters in DRAM; on an L3 miss, MC fetches the missing data block's counter from DRAM to decrypt and verify the block once the block arrives from DRAM.

Many steps of decryption and verification (and thus most of the total latency) do not need data blocks themselves; only a block's counter contributes to these steps. As such, MC can start these counter-only steps for decrypting and verifying a data block before the data block arrives from memory, as long as MC has the block's counter; correspondingly, MC caches the counters [1]. To maximize the hit rate of counters in MC's cache, prior works propose split counter designs [2][3] to make counters more cacheable.

However, the counter cache in MC is too small for many large real-world applications. The large memory footprint and irregular access patterns of many real-world applications can lead to high counter miss rate in MC's cache. High counter miss rate means frequently delaying decryption and verification and, thus, increasing total memory access latency. High counter miss rate also translates to high bandwidth overhead.

To address the low hit rate of counters in caches due to big workloads, a promising solution is to cache counters in LLC, like how LLC also caches page table blocks in LLC in existing CPUs. Prior works [4][5][6][3][7][8] use LLC as a second level cache for the smaller counter cache in MC.

Caching counters in LLC introduces a new problem, however. Server CPUs have a long LLC access latency (e.g., 23ns) that is comparable to DRAM latency (e.g., 16ns and 30ns under row buffer hit and miss, respectively). The long latency of fetching counters from LLC not only reduces the benefit of caching counters in LLC, but can significantly

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00054

increase the total latency of some memory accesses.

We note the problem lies with MC sitting behind LLC; due to its physical location, MC can only see LLC misses and, therefore, can only serially access and use counters after data miss in LLC has completed. However, prior designs without caching counters in LLC can access and use counters in parallel with accessing data. If a block's counter misses in MC's counter cache, MC can fetch the counter from DRAM in parallel with data; if the counter hits in MC's counter cache, MC can use counters to calculate in parallel with data traveling from DRAM to MC.

To parallelize the access and use of counters with data access while caching counters in LLC, we observe that in modern CPUs, L2 is typically the first place that caches data from DRAM (i.e., L2 and L3 are non-inclusive); as such, data from DRAM need not be decrypted and verified until they reach L2. So it is possible to offload some decryption and verification tasks from MC to L2. Since L2 sits before L3, L2 can access counter and data in parallel from L3; L2 can also use counters for cryptography calculation in parallel with data traveling from DRAM to L2, instead of just from DRAM to MC. As such, we propose caching and using counters directly in L2. Later, when the data block eventually arrives at L2, L2 directly uses the results computed locally at L2 to finish decrypting and verifying the data. We refer to this idea as Eager Memory Cryptography in Caches (EMCC).

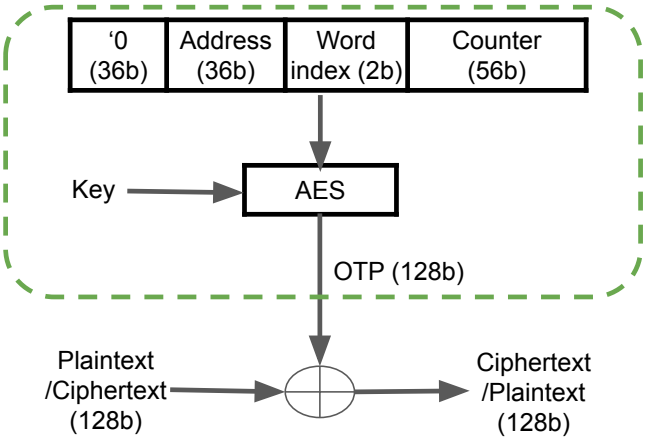
Overall, this paper makes the following contributions:

- We explore the problem of long LLC access latency when caching counters in LLC.
- We propose Eager Memory Cryptography in Caches to effectively hide the long latency of accessing counters in LLC for both when counters miss and hit in LLC.
- Our evaluations show that when applied to a state-of-the-art baseline (i.e., Morphable Counters [2]), EMCC improves performance by 7%, on average across large and/or irregular workloads.

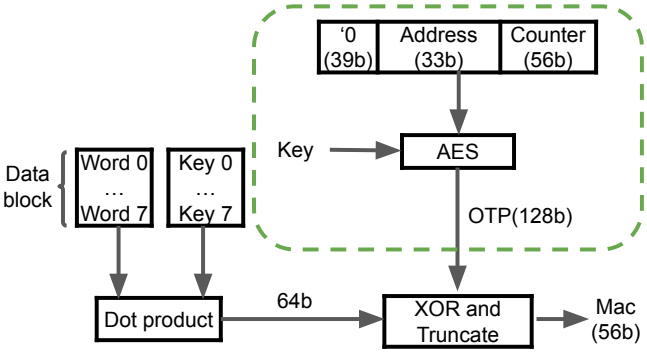
II. Background

Threat Model: Like prior works [1], [2], this paper focuses on a threat model where the attackers (e.g., disgruntled Cloud employees) have physical access to a system. Under this threat model, the CPU-memory bus (and thus off-chip memory contents themselves) is vulnerable to snooping and tampering. To assist with system-level integration test and debugging, many DDR4 memory bus probes [9] can accurately read all values and commands transmitted over the bus. An adversary with physical access to the computer can buy any of the many commercial off-the-shelf DDR4 memory bus probes [9] to read everything from memory.

To provide confidentiality and integrity against a strong adversary, capable of launching replay attacks via having physical access, prior works [2], [10] rely on counter mode encryption, coupled with Message Authentication Codes



(a) Encryption and Decryption



(b) MAC calculation.

Figure 1: Encryption and MAC calculation. Calculations in dashed boxes can start without data arriving from DRAM.

(MAC) [2], [10]. Under prior works, CPU's memory controller (MC) encrypts a memory block and protects it with a MAC whenever MC writes the block to memory. Later, when MC reads the block from memory, MC decrypts the block and uses the MAC to verify the block's integrity.

Memory encryption and verification relies on cryptography calculations, such as Advance Encryption Standard or AES. AES has a fixed size of 128 bits for both input plaintext and output ciphertext. AES has different key sizes with different number of rounds; each round performs four transformations.

Ensuring Confidentiality: To encrypt data to securely writes it to DRAM, MC calculates the ciphertext for the data by using AES to calculate an One-Time Pad (OTP) and bitwise XORing the OTP with data to produce the ciphertext; MC writes to DRAM the block's encrypted ciphertext. Specifically, OTP is calculated from counter-mode AES; counter-mode uses a nonce (number only used once), also known as the write counter, as the input to AES (see Figure 1a). Since each AES calculation always outputs 16B, each OTP is also 16B; as such, encrypting a 64B data block requires four OTPs.

To prevent an important vulnerability, where an XOR

of two ciphertexts using the same OTP can leak plaintext information (i.e., reveal the XOR of the two plaintexts), state-of-the-art protection techniques use different OTPs when writing the same data block back to DRAM. A counter is maintained for each 64B memory block; MC increases a block's counter whenever MC writes the block to memory. MC stores the counters in DRAM.

When fetching a ciphertext block from DRAM, MC decrypts the block. MC uses AES to recompute the same OTP previously used to encrypt the block and then bitwise XOR the recomputed OTP with ciphertext to obtain (and thus decrypt) the block's original plaintext.

The most time-consuming calculation under decryption is AES; the bitwise XOR after AES is fast.

Since only a block's address and counter contribute to OTP, OTP calculation can start without data. As such, to speed up decryption, MC can cache counters in MC to use them to calculate AES as soon as a data read request arrives at MC, before the data response arrives at MC. Counters are more cacheable than data because each data block is 512 bits, while each counter is 56 bits [1].

Verifying Integrity: To detect data tampering, state-of-the-art techniques use a 56-bit MAC for each data block. When writing back a block to DRAM, MC calculates the block's MAC by xoring an AES result and a Galois Field (GF) dot product result (see Figure 1b). Later, when reading a block from DRAM, MC recomputes the MAC for the block to check if it matches the MAC fetched from the memory. This MAC verification check securely detects malicious tampering and ensures integrity.

The most time-consuming calculation under integrity verification is AES since it requires multiple sequential rounds of calculation; in comparison, dot product is faster because all GF multiplications can be in parallel. Therefore, caching counters, which are the inputs for AES, also speeds up verification, in addition to speeding up decryption.

Counter Blocks: Counters are stored in DRAM in 64B blocks, called counter blocks, like 64B data blocks. Each counter block consists of eight 56-bit counters. Unlike data, counters are stored in DRAM as plaintext.

To protect against tampering of counter blocks themselves, each counter block is also protected with its own MAC. The MAC calculation for each counter block takes as input another counter value (i.e., each counter block itself is protected by another counter). State-of-the-art protection techniques [2] [10] organize the protecting counters in a tree called integrity tree.

When receiving from LLC a read request for data, if MC finds that the requested data block's counter block is not currently cached, MC must fetch the counter block from memory. In this scenario, MC must also use the counter block's counter to recompute the counter block's MAC to verify the counter block itself. To reduce the overhead of also fetching and verifying the counter block's counter, MC also caches the counter block's counter like data's counter.

Improving Counter Hit Rate: To reduce the latency overhead of memory decryption and verification, many prior works seek to improve the hit rate of counters in caches. Prior works [4][5][6][3][7][8] also cache counters in the Last Level cache (LLC) to use LLC as a second level cache for the counters to back up MC's private cache.

To improve counter hit rate, SC-64 [3] increases the number of data blocks protected by each 64B counter block from eight to 64 by packing more (i.e., 64 instead of eight) counters in each counter block. Each counter in SC-64 [3] occupies only 7 bits instead of 56 bits. The number of data blocks that each level of integrity tree nodes covers increases exponentially. For example, the number of data blocks protected by each node in the first level in the integrity tree increases from 64 to 4096 - a factor of 8^2 increase. To further improve counter hit rate, Morphable Counters [2] increases the number of memory blocks protected by each counter block from 64 to 128.

III. Characterizing the Problem

As memory system and dataset sizes keep increasing, the memory footprints of many real-world applications have increased to hundreds of GBs [11][12][13][14]. Huge datasets in memory are often used by many server workloads such as databases and data analytics [15][16][17][18][19]. Many workloads also exhibit irregular memory access patterns. Both large memory footprint and irregular accesses increase the rate of TLB misses. Many prior works have studied the problem of high rate of page table entry miss in TLBs [11][20][21][22][23][24][25][26].

Although state-of-the-art counter designs (i.e. Morphable Counters [2]) increases each counter block's coverage to 8KB, it is still similar to the coverage of a 4KB page table entry (i.e., 4KB). Because workloads with irregular access patterns have high TLB miss rates for 4KB page table entries, counter blocks also suffer from high miss rate for those workloads.

To improve counter hit rate in caches, a promising solution is also caching counters in LLC, just like how LLC also caches page table blocks. Make prior works cache counters in LLC [4][5][6][3][7][8].

To characterize the benefit of caching counters in LLC, we model in Pintool [27] Morphable Counters both with and without caching counters in LLC. We model 1MB of L2 and 2MB of LLC per thread in Pintool.

We examine all workloads used by recent prior works [11][26] on improving address translation that we can run in Pintool; when evaluating graphBIG [28], we use Facebook-like dataset (i.e., $8\_5 - fb$ [29]) and four threads. We measure total DRAM traffic overhead, including counter accesses to DRAM and Morphable-specific traffic (e.g., overflow traffic), normalized to the total number of normal data accesses across the whole lifetime of each workload.

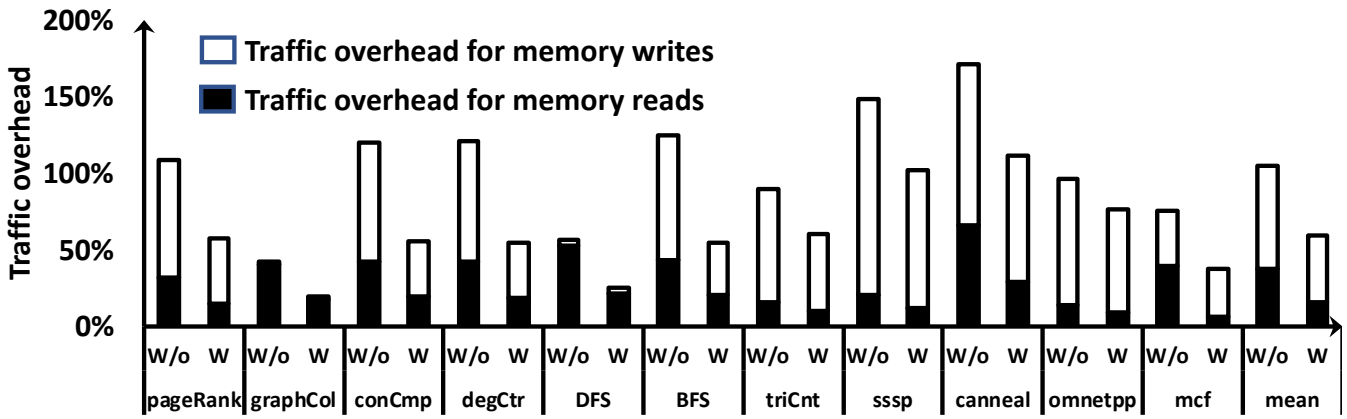


Figure 2: DRAM traffic overhead normalized to normal data traffic.

The results show that caching counters in LLC reduces total DRAM traffic overhead from 105% down to 59% (see Figure 2). As such, caching counters can effectively reduce memory bandwidth overhead.

The problem with caching counters in LLC, however, is that that modern server CPUs have long LLC access latency. This long LLC latency not only can significantly diminish the benefit of caching counters in LLC, but can actually sometimes significantly increase counter access latency compared to not caching counters in LLC.

A. Quantifying LLC Latency in Server CPUs

We conduct real-system experiments to measure average LLC latency for an Intel Xeon W-3175X CPU with 28 cores.

Real-system Methodology: We write a read-intensive microbenchmark with 16MB workload size to generate 100% LLC hit rate on a server with 20MB LLC. We fix the CPU frequency to the CPU's base frequency 3.1GHz. To minimize measurement noise, we ensure only one cache or memory access at a time by turning off all prefetching and using pointer-chasing in the main access loop. We use RDTSC to measure the latency for each LLC access. We repeat the microbenchmark 28 times, each time pinned to a different core.

Measurement Results: Figure 3 shows our measured LLC hit latency. It is 23ns, on average. This is close to the 26ns reported in public references [30][31].

The LLC hit latency shows obvious variations; some LLC hits can take significantly (e.g., $> 10ns$) longer than others. Modern server CPUs have a distributed architecture that splits LLC into multiple tiles. Each L2 communicates with different LLC tiles via multiple hops through a network-on-chip (NoC) that connects multiple LLC tiles. The multiple hops incur long latency. The number of hops depends on the distance between the requesting L2 and the destination LLC tile and, hence, the variation in LLC hit latency. Figure 4 shows an example of how L2 cache communicates with LLC tiles via the NoC after a L2 miss. The NoC topology shown in Figure 4 is the NoC topology of the CPU used in our real-system measurements.

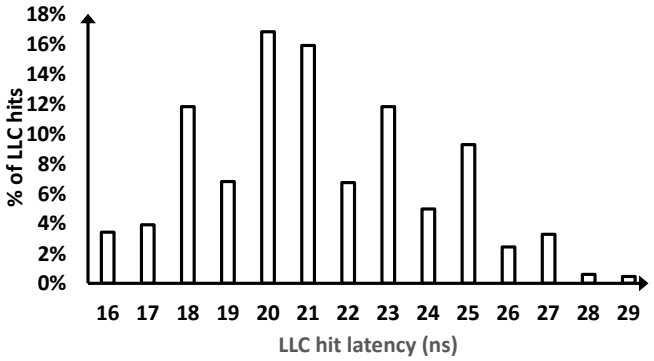


Figure 3: Distribution of LLC hit latency.

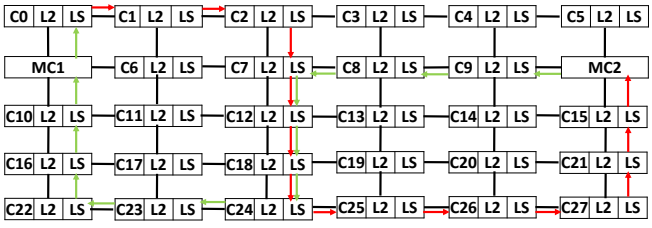


Figure 4: NoC traffic for an L2 cache miss. "C" stands for core; "LS" stands for LLC slice. Core0 executes a load to data block X. X misses in Core0's L2. The L2 uses X's address and a static mapping function to determine the LLC slice (i.e., slice 24 in the example) that can cache X. When X also misses in LLC slice 24, the slice requests X from MC. Red arrows show requests; green arrows show response.

B. Why Long LLC Latency is a Problem for Counters

Due to the symmetry of NoC topology, fetching counters from LLC to MC takes the same long latency as fetching data from LLC to L2. We call the latency of fetching data

from LLC to L2 *Direct LLC Latency* (i.e., the latency of directly accessing LLC without the latency of accessing L2). As such, in a secure memory system that caches counters in LLC, Direct LLC Latency would be the latency for MC to access the counter in LLC. Based on our measurements, we estimate Direct LLC Latency to be 19ns.[1] This long Direct LLC Latency increases the total latency of a memory access in secure memory systems, which we refer to as *Secure Memory Access Latency*; Secure Memory Access Latency starts from MC receiving a data request and ends at MC replying decrypted and verified data back to LLC.

When counter misses in LLC, Secure Memory Access Latency increases by Direct LLC Latency compared to without caching counters in LLC. Without caching counters in LLC, MC directly fetches the counter from DRAM when counter misses in MC's cache (see Figure 5). With caching counter in LLC, when a counter misses in MC's cache, MC requests LLC for the counter, suffers from LLC miss, and then fetches the counter from DRAM. Without caching counters in LLC, counter accesses in MC and DRAM are on the critical path of Secure Memory Access Latency (see Figure 5). Caching counters in LLC adds another LLC access (i.e., counter access in LLC) to the critical path (see Figure 5); as such, caching counters in LLC increases Secure Memory Access Latency by 19ns Direct LLC Latency.

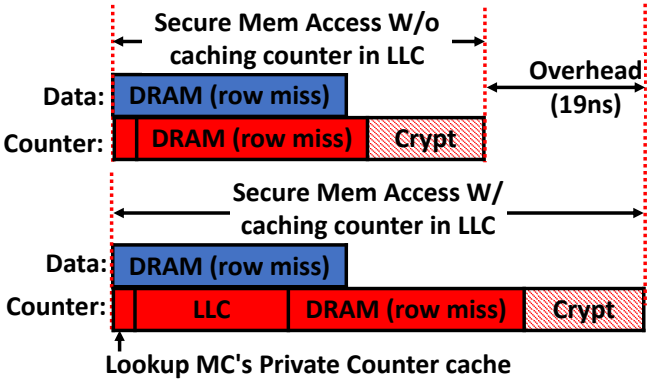


Figure 5: Timeline of Secure Memory Access Latency under counter miss in caches. Each bar is drawn proportionally assuming the following: accessing counters in LLC takes 19ns; counter-dependent cryptography computation (i.e., counter-mode AES) takes 14ns[2]; MC's counter cache takes 3ns, which is between L1 hit latency and L2 hit latency. We assume MAC is embedded in data [2][4]; as such, MAC access to DRAM is not shown.

[1]We measure L2 access time to be 6ns, like the L2 hit latency reported in public references [30] [31]. As L2 cache is big (i.e., 1MB), we assume data read only occurs sequential after tag hit. As such, under LLC hit, which requires L2 miss, we assume L2 access only consists of tag miss without data read. Assuming L2 data read takes 2ns, L2 access takes 6ns - 2ns = 4ns under L2 miss. Therefore, Direct LLC Latency = LLC hit latency - 4ns = 23ns - 4ns = 19ns.

Increasing Secure Memory Access Latency for counter misses in LLC is a problem for workloads with significant counter miss rate in LLC. Many irregular workloads suffer from frequent counter misses when caching counters in LLC; on average, 19% of normal block misses in LLC also suffer from counter misses in LLC (see Figure 6). This is despite taking careful evaluation measures to minimize counter miss rate. For example, all of our Pintool experiments run workloads under 2MB huge pages to maximize counter hit rate for Morphable Counters. Morphable Counters is sub-optimal under 4KB pages; while each counter block in Morphable covers two adjacent 4KB physical pages (i.e., 128 adjacent physical memory blocks), OS may map two nearby 4KB virtual pages to two far-apart physical pages when using 4KB pages. Two far-apart physical pages require two different Morphable counter blocks, instead of one; this increases counter misses.

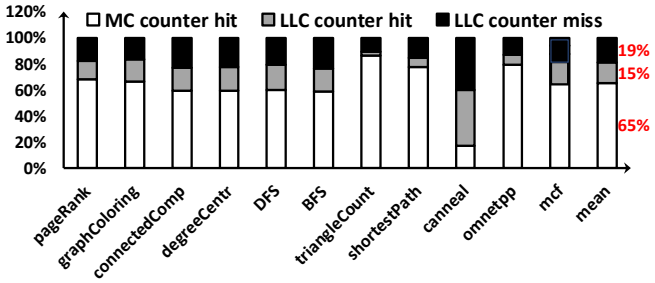


Figure 6: Counter hits and misses in MC and LLC for read requests to normal data under an LLC with 2MB/core and a shared counter cache with 32KB/core. The number of counter hits and misses are normalized to the number of normal memory reads.

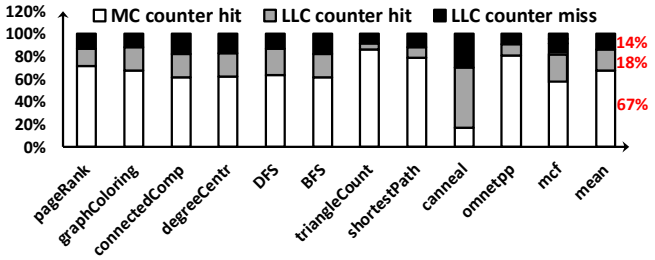


Figure 7: Counter hits and misses in MC and LLC for read requests for normal data under an LLC with 12MB/core and a shared counter cache with 32KB/core. The number of counter hits and misses are normalized to the number of normal memory reads.

Counter miss rates in LLC remain high even under bigger LLCs. Figure 7 shows that the average counter miss rate

[2]Cryptography computation typically uses AES-128. The 14ns latency we assume is faster than the latency of AES-128 under 7nm technology [32] considering possible future improvements.

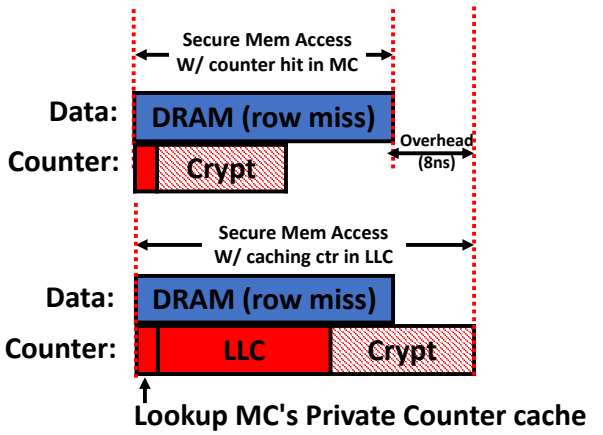


Figure 8: Timeline of Secure Memory Access Latency under counter hit. It assumes the same latencies as Figure 5.

in LLC only reduces from 19% down to 14% when LLC increases from 2MB per core to 12MB per core.

Even if counter hits in LLC, Secure Memory Access latency also increases compared to counter hits in MC's cache (see Figure 8). When counter hits in MC's counter cache, counter-mode AES completely overlaps with DRAM access so that counter access is not on the critical path of Secure Memory Access Latency (see Figure 8). However, with counter access in LLC, counter-mode AES only partially overlaps with DRAM access (see Figure 8) and, therefore, adds counter access to the critical path and increases Secure Memory Access latency.

We expect the problem of long counter access latency to LLC to worsen going forward. First, as the number of cores increases, the number of hops required to traverse the NOC also increases; this increases the latency of accessing LLC. Second, emerging multi-chiplet architectures, which move MC and subgroups of cores to different chiplets, will further increase the latency for MCs to access LLC.

IV. Eager Memory Cryptography in Caches

To hide the long latency of accessing counters in LLC, we note the root problem lies with MC sitting behind LLC; as such, MC can only see LLC misses and, therefore, can only fetch counters from LLC serially after data miss in LLC.

A promising solution is to parallelize counter and data access in LLC. Perfectly parallelizing counter and data accesses in LLC can achieve the same performance as idealized zero-latency counter accesses from LLC. This enables MC to simultaneously receive from LLC a request for data and either the data's counter or miss request for the data's counter. If MC simultaneously receives a request for data and the data's counter, MC can start counter-mode AES when starting DRAM access for data, just like a hit in MC's counter cache. If MC simultaneously receives a request for data and a miss request for the data's counter, MC can fetch data and counter from DRAM in parallel, just like without caching counters in LLC.

A. Challenges of Parallel Counter & Data Accesses in LLC

Parallezing counter and data Accesses in LLC requires addressing two key challenges:

Challenge 1): Parallelizing counter and data access in LLC can significantly increase LLC accesses and NoC traffic. Parallelly accessing counters in LLC for *every* data access in LLC will double or even quadruple LLC accesses and NoC traffic; note that verifying a data block not only requires the data's counter, but sometimes also the counter's counters. Such high on-chip overheads are due to two reasons. First, whether a counter access is necessary is not known until data access in LLC completes; the parallel counter access to LLC issued at the start of the data access in LLC becomes useless if the data access hits in LLC. Second, whether a counter is already in MC's counter cache is also not known at the start of data access in LLC; the parallel counter access to LLC issued at the start of the data access in LLC becomes useless if the needed counter block is already in the counter cache at MC. As such, parallel counter and data accesses in LLC can cause many useless counter accesses in LLC and useless NoC traffic.

Challenge 2): Truly parallel lookup is impossible. Due to the highly non-uniform NoC latency (see Figure 3), the NoC latency for accessing counters from LLC can sometimes (e.g., 50% of the time) be longer than accessing data from LLC. When counter access hits but data access misses in LLC, accessing counter is also slower than data due to A) incurring tag + data lookup latency in LLC, instead of just tag latency, and B) NoC taking longer to transmit actual counter payload than the data miss request.

B. Key Idea: Moving decryption/verification from MC to L2

In this paper, we address the above two challenges to extract the full performance benefit of truly parallel lookup at the cost of minimal on-chip bandwidth overhead.

We observe that in modern CPUs, L2 is typically the first place that caches new data from DRAM. Unlike Intel's older Haswell architecture, where L3 is inclusive of L2, newer Skylake architecture uses a non-inclusive cache hierarchy where L3 serves as a victim cache that caches evictions from L2. Because L2 is the first place that caches data from DRAM, data arriving from DRAM need not be decrypted and verified until they reach L2. So it is possible to move selected memory cryptography tasks from MC to L2. Doing so provides the following benefits:

1) Because L2 sits before LLC, L2 can issue counter and data requests to LLC in parallel, just like how MC can issue counter and data requests to DRAM in parallel.

2) After L2 receives counters from LLC, L2 can naturally cache the counters. Caching counters in L2 filters out most of the useless counter accesses to LLC, like how MC's counter

cache can eliminate all useless counter accesses to DRAM in prior designs that do not cache counters in LLC.

3) Calculating counter-mode AES in L2 can increase the overlap between computation and data movement; calculating in L2 can overlap computation with data traveling from DRAM back to L2, instead of just overlapping computation with data traveling from DRAM to MC like prior designs.

As such, we propose moving common-case decryption and verification tasks from MC to L2. Specifically, we propose a) caching counters and b) using counters directly in L2 and refer to this idea as Eager Memory Cryptography in Caches (EMCC). Figure 9 gives an overview of EMCC.

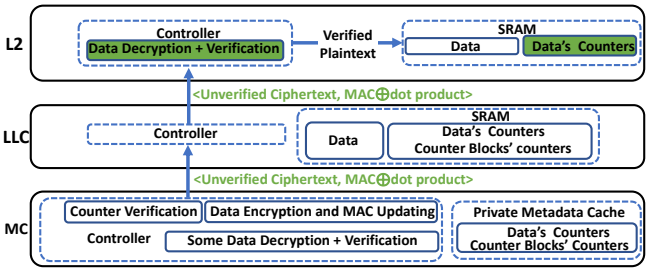


Figure 9: Architectural overview of EMCC. New/modified actions are in green.

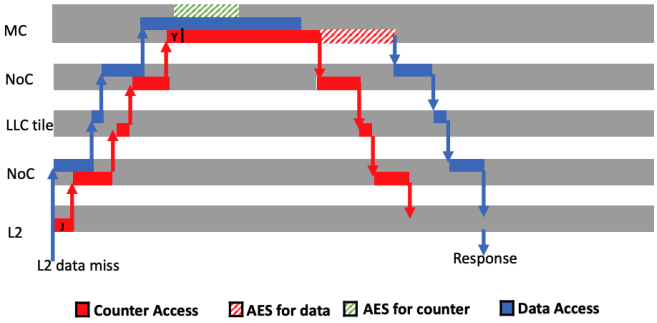
The rest of Section IV is organized as follows. Section IV-C describes how EMCC efficiently caches counters in L2 to filter out useless counter accesses to LLC. Section IV-D describes how EMCC uses the counters cached in L2 to decrypt and verify counters directly at L2.

C. Caching counters in L2

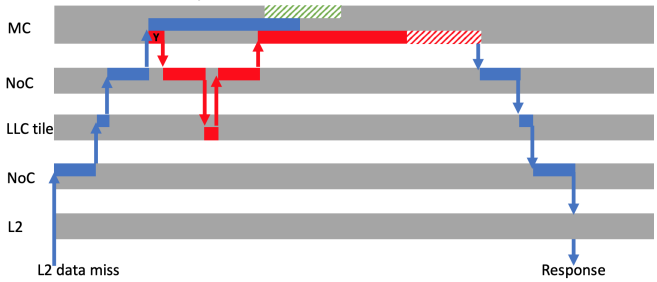
EMCC caches counter blocks in L2, like how current CPUs cache page table blocks in L2. Under EMCC, after each data miss in an L2, L2 serially looks up the data's counter in the L2. If the counter block also misses in L2, L2 issues a request for the counter block and caches the counter block when it arrives from LLC. As such, EMCC reuses L2 as a small level-one counter cache to filter out most of the counter accesses to LLC, just like how the counter cache in MC eliminates unnecessary counter accesses to LLC in a basic design that serially accesses LLC for counter after data miss in LLC.

Serially looking up a counter in L2 only after a data block misses in L2 reduces the degree of overlap between data access and counter access in LLC. However, the resultant counter access to LLC can still proceed mostly in parallel with the data access in LLC (see Figure 10a); as such, EMCC can still respond to L1 much sooner than the baseline (see Figure 10b).

When caching counters in L2, one issue is that DRAM accesses (both reads and writes) require accessing multiple counters (i.e., not just the counter protecting the data, but also counters protecting data's counters). Caching so many



(a) EMCC: memory access timeline under counter miss in LLC.



(b) Baseline: memory access timeline under counter miss in LLC.

Figure 10: Timeline of memory accesses under counter miss in LLC and row buffer miss in DRAM. 'J' is the delay of counter access in L2 due to sequential access of data and counter. 'Y' is the latency of counter access in MC's counter cache. In this scenario, EMCC can respond decrypted and verified data back to L1 16ns earlier than the baseline.

counters per memory access can pollute the small L2 cache. Fortunately, only data's counters are frequently used. As such, EMCC only caches data's counters in L2. We also note that reads are more critical to performance than writes; as such, L2 only accesses and caches counters for data misses in L2, but not for writebacks from L2. As such, EMCC continues to verify counters and cache tree nodes in MC (see Figure 9). Like all prior works, MC is still fully responsible for verifying counters when they miss in caches (i.e., not found in L2, LLC, and MC's counter cache). MC only replies to LLC and L2 with a counter block after verifying the counter block.

Another issue is that at the time of L2 data miss, L2 does not know whether the data will also miss in LLC. However, to parallelize counter and data accesses in LLC, L2 must speculatively request LLC for the data's counter, if the counter also misses in L2. The parallel counter access to LLC may be useless if the data access hits in LLC. We note that even in this scenario, the counter access to LLC may still be useful if the copy of the counter inserted into L2 is used for a later data miss in LLC. As such, a parallel counter access to LLC from L2 is only useless if it is never used for any data miss in LLC between the time the counter

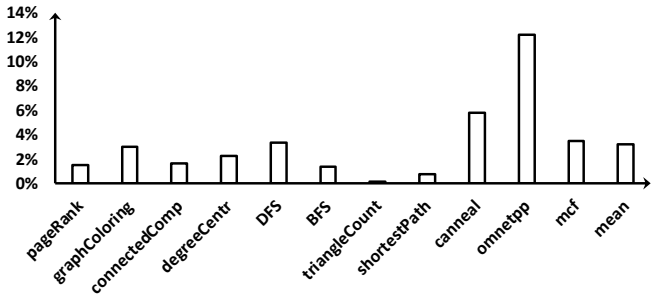


Figure 11: Useless counter accesses to LLC under EMCC, normalized to total number of L2 data misses. It is only 3.2% on average, thanks to caching counters in L2. Morphable Counters is used as the underlying counter design.

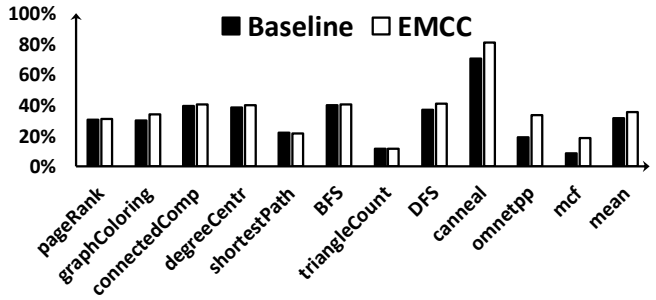


Figure 12: Total counter accesses to LLC under EMCC and baseline, normalized to total number of L2 data misses. Here, Morphable Counters is used as the underlying counter.

is inserted into L2 and is evicted from L2.

We model EMCC in Pintool. Figure 11 shows the number of counter accesses that L2 issues to LLC normalized to total data accesses to LLC. It is only 3.2%, on average. As such, caching counters in L2 practically eliminates the bandwidth overhead due to speculatively accessing counter in parallel with accessing data in LLC. Figure 12 shows the total counter accesses to LLC normalized to total data accesses to LLC under EMCC; it is 35.6%, on average. As comparison, Figure 12 also shows the total counter accesses to LLC normalized to total data accesses to LLC under a baseline design that uses LLC as a conventional second-level cache for counters; this baseline always serially accesses a data block's counter in LLC after the data block misses in LLC to minimize counter accesses to LLC. EMCC incurs only 4.2% more counter accesses to LLC than the baseline.

Hardware Overheads: Caching counters in L2 incurs no SRAM area overhead; EMCC reuses the same L2 to cache counter blocks, just like how L2 is used to cache data blocks, instruction blocks, and page table blocks. Caching counter blocks in L2 requires no changes to existing L2 circuits; it only requires adding a new standalone circuit to issue counter block requests, like the L2 prefetcher. Another hardware change is adding a standalone circuit to MC to issue invalidation requests when MC updates counters while serving writebacks to DRAM. This is an implementable change. For example, Intel's previous Haswell CPUs' MCs contain Home Agents to send invalidation requests to caches to support multi-socket systems [33]; consider a two-socket system, before Socket A writes to a block owned by Socket B, A sends an upgrade request to B's MC, which invalidates all cached copies in B.

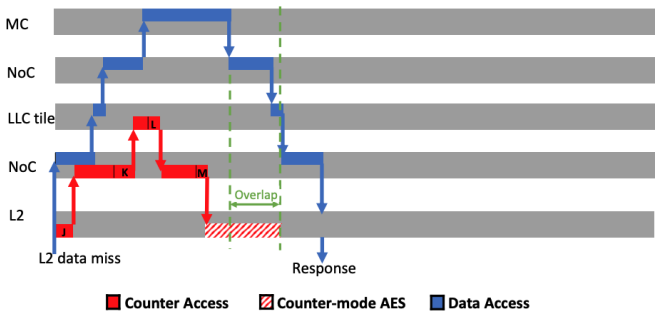
D. Using Counters in L2 for Cryptography Computation

As described in Section IV-A, another challenge with parallel counter and data access in LLC is that counter accesses are often slower than data access, even if the two requests are issued at the same time. Our approach of serially accessing counters in L2 only after data misses in L2 to filter out useless counter accesses to LLC (see Section IV-C) further worsens this problem; it delays the corresponding counter request to LLC by L2 latency after a data request to LLC. Contentions in L2 may further delay the serial counter access in L2; after a data miss in L2, EMCC only looks up the corresponding counter in L2 during spare L2 cycles.

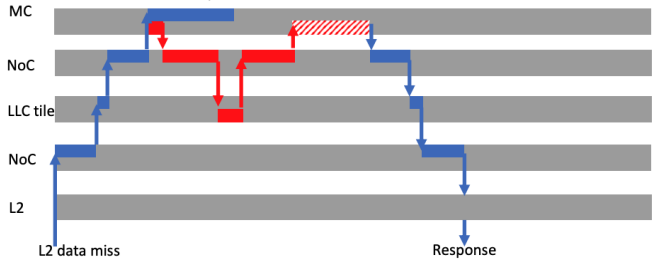
To address the challenge that counter accesses to LLC are slower than data accesses to LLC, we note that under prior designs, MC can overlap counter-mode AES computation with data traveling from DRAM to MC [34]. However, the overlap ends when the data block arrives at MC.

We observe that most of the LLC access latencies in modern server CPUs are due to NoC latencies; using our real-system measurements in Section III, we calculate average one-way latency between two NoC nodes on-chip to be 7.5ns; see 'One-way latency calculation' in Appendix. As such, data response takes a long time to travel from MC to L2; see 'MC-to-L2 route' in Appendix for how we calculate MC-to-L2 NoC latency. Therefore, we propose computing counter-mode AES for data directly in L2. Computing in L2 enables computation to overlap with data traveling from DRAM all the way to L2; this provides significantly more overlap than overlapping computation with simply data traveling from DRAM to MC. The increased overlap between computation and data movement can effectively hide the delayed start of computation due to slower counter accesses to LLC than data requests to LLC.

Figure 13a "Overlap" shows how EMCC overlaps computation with data traveling from DRAM to L2, assuming counter hit in LLC and row buffer hit in DRAM. After a counter miss in L2 following a data miss in L2, L2 issues a request to LLC to fetch the data's counter. After counter arrives from LLC, L2 uses it to start counter-mode AES, in parallel with waiting for data response from MC. With the long latency of data traveling from MC to L2, when data arrive at L2 from LLC, L2 would have finished calculating AES. L2 uses the locally computed results to decrypt and verify the data. In the common case, the data is correct; L2 caches the decrypted data and forwards it to L1. In the

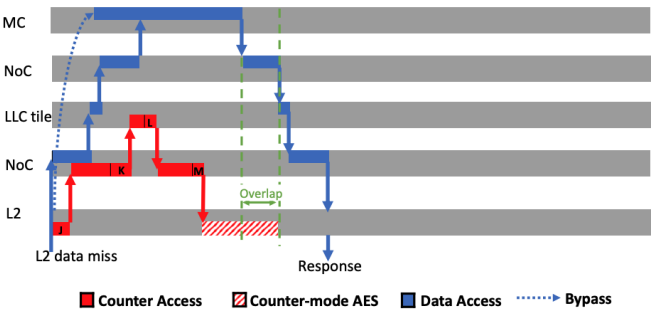


(a) EMCC: memory access timeline under counter hit in LLC.

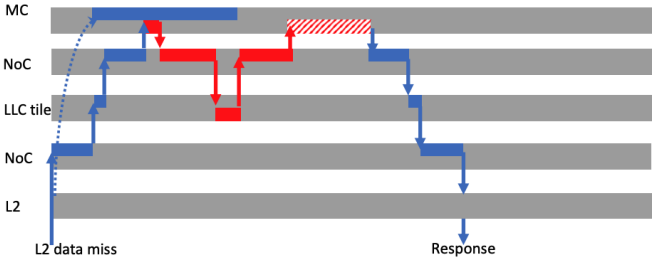


(b) Baseline: memory access timeline under counter hit in LLC.

Figure 13: Timeline of memory accesses under counter hit in LLC under EMCC and under the baseline as comparison. Red indicates counter access path; Blue indicates data access path. 'K' represents non-uniform NoC latencies that sometimes make counter requests travel longer than data request; 'L' indicates the latency due to serially looking up data array after tag match when a counter hits in LLC; 'M' is the latency of transmitting actual counters (as opposed to just counter requests) across NoC.



(a) EMCC: memory access timeline under counter hit in LLC.



(b) Baseline: memory access timeline under counter hit in LLC.

Figure 14: Timeline of memory accesses under row buffer miss in DRAM, counter hit in LLC, and LLC miss prediction. In this scenario, EMCC can respond decrypted and verified data back to L1 22ns earlier than the baseline.

uncommon case, L2 detects data tampering; L2 incurs a hardware interrupt to signal the system, just like how L2 incurs ECC hardware interrupt for ECC error.

Even with row buffer miss, EMCC can also overlap computation with data traveling form DRAM to L2. Figure 14 gives an example of overlap under row buffer miss and LLC miss prediction. LLC miss prediction like Intel XPT [35][36][37][38] predictively forwards a miss request from L2 directly to MC in parallel with accessing LLC. Because irregular workloads suffer from high LLC miss rate (i.e., 91% on average across workloads in Section III), we expect XPT to be used for common-case memory accesses for irregular workloads.

EMCC only moves data decryption and verification to L2. EMCC continues to encrypt data and calculate new MAC for data in MC, just like existing designs because these tasks are only required for writing back to DRAM; note that L2 only observes reads from DRAM, but not writebacks to DRAM. Because AES only takes counter as input but not data (see Figure 1), L2 uses the counters it caches to calculate AES for both data decryption and verification. To help L2 to verify data arriving from MC, MC embeds in the data response to LLC the bitwise XOR of data's MAC from DRAM and the dot product of data (see Figure 9); as shown in Figure 1b, MAC is the bitwise XOR of this dot product with AES result. As such, when L2 receives data and the embedded XOR result, L2 verifies the data by checking if the XOR result matches the AES result calculated locally at L2.

One issue with calculating the dot product at MC is that MC only has ciphertext, while the dot product is typically calculated from plaintext (see Figure 1b). This issue can be addressed by simply changing the MAC calculation to use each block's ciphertext, instead of plaintext.

L2 only verifies data arriving from DRAM; counter blocks arriving from DRAM are always verified in MC (see Figure 9), like prior designs. MC only sends counters arriving from DRAM back to LLC after verifying them; this ensures L2 only uses verified counters to verify data. Verifying only data, but not counters, at L2 minimizes the control logic/area to add to L2; counter verification is a complex operation that accesses up to multiple counter blocks, starting from the lowest integrity tree level that is still cached on-chip.

To minimize area overhead, instead of adding more AES units to L2, EMCC moves some AES units from MC to L2s so that the total AES units in the processor remains unchanged. EMCC distributes half of the AES units in MC across the L2's; MC retains the other half to perform encryption for writebacks and to verify counters accessed from DRAM. Moving AES units from MC to L2s is feasible

because AES units are small (e.g., $3894um^2$ per unit under 7nm technology node [32]).

One issue is that each L2 can only get relatively low AES calculation bandwidth (e.g., see calculations in Section V). As such, spikes in L2 misses can lead to long queuing delay to wait for AES calculations.

To address this challenge, EMCC adaptively offloads data decryption and verification back to MC. When EMCC determines that the AES queuing time for a new L2 miss exceeds the latency that can be saved by performing cryptography computation in L2, L2 gives up cryptography computation for this L2 miss. Instead, L2 embeds this binary decision in the miss request that L2 sends to LLC, which propagates the bit to MC if the request also misses in LLC. When receiving a LLC miss request with this embedded decision bit, MC knows to decrypt and verify the requested data after it arrives from DRAM before responding to LLC, just like existing MCs.

Another issue is that when a data block hits in LLC, calculating AES for the data access is useless and wasteful. As such, L2 only starts calculating AES for a data miss at L2 after waiting LLC hit latency. At this time, if L2 still has not received the response to the L2 data miss, L2 knows the data block likely also misses in LLC and, therefore, can safely start calculating AES for the data miss without risking wasting any AES bandwidth at L2.

To further conserve AES bandwidth at L2, EMCC only decrypts and verifies a data block at L2 when the data block's counter hits in LLC or L2. When MC receives a miss request for a counter block, EMCC decrypts and verifies the corresponding data access to DRAM in MC (see Figure 10a), instead of L2, and tags the data block as decrypted and verified while sending the data block back to the cache hierarchy; this informs L2 to not redundantly decrypt and verify the data block when receiving the block.

E. Security Analysis

Prior works cache counters in LLC. The only thing that MC does differently from prior works, as far as security may be concerned, is also caching counters in L2.

Caching counters in L2 should not open up any new useful side channels because cached counters are less useful to attackers than cached data blocks; counter blocks cannot be explicitly flushed. In the vulnerable scenario of finely (i.e., at small time windows) sharing cores across multiple users/processes, attackers can simply use existing/traditional attacks targeting data blocks in L2, without any motivation to explore new attacks targeting counters in L2. Caching counter blocks in L2 makes side-channel attacks more difficult by adding more uncertainty to cache replacement and making L2 less controllable. In the converse scenario, when different processes run on their own cores, caching counters in L2 has no impact on side-channel vulnerabilities; security-sensitive users (e.g., ones wanting memory encryption) typically do not finely share their cores with others as fine sharing opens up broad side-channel vulnerabilities.

F. Discussion

Inclusive Cache Hierarchy: While EMCC naturally suits non-inclusive cache hierarchies, EMCC can also be extended to an inclusive cache hierarchy; however, some changes to LLC and L2 are required.

For LLC, when an encrypted and unverified block arrives from DRAM, LLC caches it as usual to ensure inclusivity but marks it as *encrypted_&_unverified_in_LLC* (e.g., via a new bit per LLC cacheline). When an L2 miss hits in LLC, if the LLC copy is currently marked as such, LLC satisfies the request by fetching a copy from an L2 that owns (i.e., under MOESI) or shares (i.e., under MSI) the block. Conversely, LLC resets the new bit to false after receiving a copy from L2 for any reason; L2 copies are always decrypted and verified.

For L2, when LLC replies to it a memory block that is *encrypted_&_unverified_in_LLC*, L2 decrypts and verifies the block as described in Section IV-D. By adding a new bit per L2 cacheline, L2 can remember to perform a clean writeback if evicting the block later in clean state, like the clean writebacks in non-inclusive caches. L2 also resets an L2 cacheline's new bit to false after LLC fetches the copy in the L2 cacheline for any reason.

Non-memory-intensive Workloads: While speeding up memory-intensive applications, EMCC should ideally waste neither precious L2 space nor energy for non-memory-intensive workloads. For applications with high L2 hit rate, EMCC wastes little to no L2 space or energy because EMCC only accesses counters in L2 after regular memory blocks miss in L2. However, for applications with high L2 miss rate but high LLC hit rate, accessing and then caching counters in L2 after L2 miss still wastes L2 space and energy. As memory has little impact on the performance of non-memory-intensive applications (e.g., ones with fewer than one memory access per thousand instructions), L2 may dynamically turn off EMCC (i.e., dynamically offload all decryption/verification back to MC) when detecting non-memory-intensive applications; to estimate the memory intensiveness of the application accessing it, an L2 may compare how many of its misses are satisfied by DRAM to how many requests it receives (i.e., periodically sample memory accesses per thousand L2 accesses).

V. Methodology

We evaluate the performance of EMCC using Gem5 [39]. We simulate the benchmarks in Section III. In Gem5 full-system mode, We run graphBIG [28] under multi-threading. We run SPEC and Parsec as multi-programmed workloads; each multi-programmed workload has four instances of the same benchmark. Same as our Pintool experiments in

CPU	X86, 4 cores, 3.2 GHz, 4-wide OoO, 192 entry ROB
D-TLB, I-TLB	1536 entries each
Page Walker Cache	2KB
Degree of constant stride prefetcher	L1: 1 L2: 2
L1 ICache/DCache	32/64KB,4/8-way,2ns
L2 Cache	1 MB 8-way, 4ns
L3 Cache	8 MB 16-way, 17ns
Counter Cache in MC	128KB 32-way, 3ns
Decoding of Morphable Counters	3ns
AES-128 latency	14ns
NoC Lat Between LLC and MC	17ns
NoC Lat Between L2 and MC	34ns
Memory	128 GB DDR4
Memory Data Rate	3.2 GT/s
tCL, tRCD, tRP	13.75ns
tRFC	350ns
Row buffer policy	500ns timeout
Read/Write queue	256 entries
Channels, Ranks	1, 8
Mapping Function	XOR-based like Skylake [41]
Bank-level scheduling policy	FR-FCFS-Capped

Table I: Primary microarchitecture parameters. Listed cache latencies are additive (e.g., total L2 hit lantecy is 2+4 = 6ns.)

Section III, our Gem5 experiments simulate running the benchmarks under 2MB huge pages. We fast forward each benchmark deep into the region of interest using Gem5's KVM mode in native execution speed. We then use Gem5' atomic mode to warm up the counter values for 25 billion instructions, like [2]. Next, we warm up the branch predictor by running the benchmark for 10ms in detailed mode. Finally, we evaluate performance during the next simulated 20ms in detailed mode; performance modelling for counter accesses are turned on during this last stage of simulation.

Table I lists the simulated microarchitecture parameters. We use 128KB counter cache in MC, like [2]. Similar to prior works [4][5][6][3][7][8], LLC also caches counter blocks and integrity tree nodes. In our simulation, EMCC only caches 32KB worth of counters in L2; limiting the amount of counters in L2 helps to ensure that our evaluated performance benefits over the baseline do not simply come from caching more counters. We use Ramulator [40] to model 128GB DRAM in Gem5. We use Gem5's classic model to simulate the cache hierarchy. We modify the classic model to simulate non-uniform NoC latency for L3 hit and L3 miss by adding a non-uniform latency component to Gem5's L3 Hit Latency and Gem5's L3 Miss Response Latency for individual cache accesses. The modelled non-uniform NoC latency follows the real-system distribution we measured in Figure 3.

Simulation Details for Counter-mode AES: To decide on the AES bandwidth for each L2 cache under EMCC, we calculate the peak AES bandwidth requirement under Morphable Counters. Since the DRAM transfer rate we model is 3200MT/seconds, the maximum number of memory accesses (include reads and writes) per second is 3.2*8B/(64B) = 400,000,000. Assuming 1:1 memory write to read ratio, the number of memory reads and writes are 200,000,000 both for reads and writes per second. Each memory read calls for five AES calculations (i.e., one for verifying data, four for decrypting data's four words); each memory write-backs calls for eight AES calculations (i.e., four for updating four MACs, four for encrypting data's four words). As such, the peak AES bandwidth under Morphable Counters is 200,000,000*5 + 200,000,000*8 = 2,600,000,000 AES calculations/second. Providing this AES bandwidth requires multiple AES units, similar to requiring multiple floating units to achieve high GFlops. EMCC moves half of the AES units from MC to four L2s so that the four L2s can collectively provide half of the peak AES bandwidth. As such, AES bandwidth in each L2 is (2600000000/2)/4 = 325,000,000 AES calculations/second.

MC fetches MAC for each memory access to verify the accessed data (see Section II). Like [2], we co-locate data, its MAC, and error correction code in the same memory block; this enables data, its MAC, and ECC to be accessed together in one DRAM access without any traffic overhead.

Baselines: We compare against Morphable Counters [2] as our primary baseline. Since we choose Morphable Counters as the primary baseline, EMCC is always applied on top of Morphable across all evaluations. We also simulate an older work - SC-64 [3].

Under Morphable Counters, extracting a block's counter value from a counter block requires decoding the counter block. Counter decode first requires extracting the corresponding minor counter from Morphable Counter block; this can take several cycles because counter blocks contain a variable and non-power-of-2 (e.g., 36, 42, 51) number of non-zero minor counters. Second, calculating the end counter value from a minor counter requires adding two major counters and the minor counter. We simulate 3ns counter decoding latency both for the baseline and EMCC.

Morphable Counters and SC-64 are split counter designs. Split counters require reading and writing entire memory page(s) to re-encrypt them when a writeback causes a counter overflow. We simulate at most two outstanding overflows at a time (i.e., MC rejects all incoming LLC requests after detecting a writeback that would incur a third overflow). In the background, MC continuously generates for outstanding overflow(s) a limited number of 64B requests at a time to prevent them from occupying more than eight slots in the read/write queue at any time; this prevents the 64B requests due to overflow(s) from seizing up the entire read/write queue.

Figure 15 shows the breakdown of bandwidth utilization for different types of memory accesses under Morphable Counters during the window of performance evaluation.

VI. Results

Figure 16 shows the performance of EMCC, Morphable Counters, and SC-64, normalized to a non-secure memory

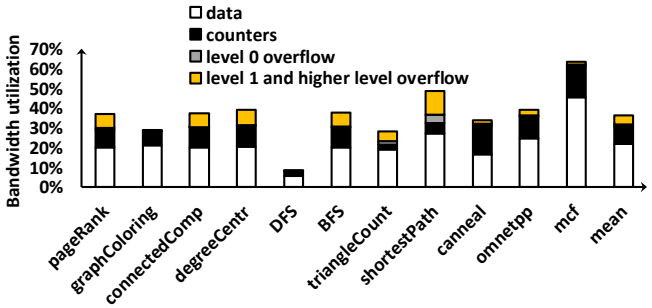


Figure 15: Memory bandwidth utilization due to DRAM data accesses, counter accesses, and overflows for each benchmark under Morphable Counters. Bandwidth is normalized to the memory channel's peak physical bandwidth.

system without encryption and verification. EMCC improves performance by 7%, on average, compared to Morphable. Canneal gets the most benefit - 12.5%; this is because canneal has the highest miss rate in MC's counter cache (see Figure 6). Since EMCC hides the long latency of accessing counters in LLC, the benchmark with the most frequent counter accesses to LLC benefits the most from EMCC.

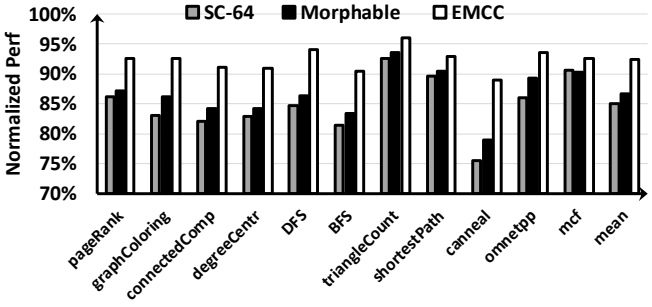


Figure 16: Performance of EMCC, Morphable and SC-64 normalized to a non-secure memory system.

Figure 17 shows L2 miss latency for both baseline and EMCC. EMCC saves, on average, 5ns on L2 data miss latency over Morphable Counters. Benchmarks with the highest savings on L2 miss latency (e.g., canneal, graphColoring) also benefits from the most performance improvement.

A. Sensitivity to AES Latency

While our primary evaluation assumes 14ns AES latency for AES-128 (see Section III), AES takes longer when stronger security is required. Providing stronger security requires AES to perform more sequential rounds of calculations. For example, AES-256 has four more rounds and takes 6ns longer than AES-128 [32].

We perform a sensitivity analysis on AES latency by evaluating the performance benefit of EMCC under longer AES latencies. The performance benefit increases with AES latency. The average improvement increases to 9% when AES latency increases to 25ns (see Figure 18).

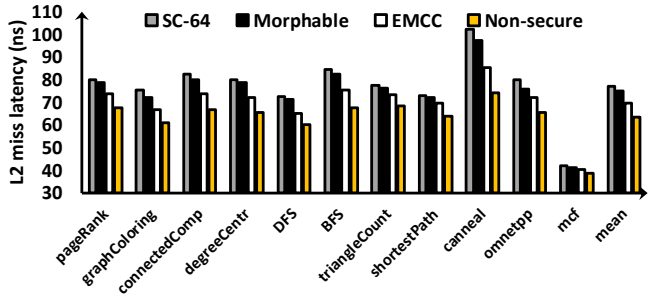


Figure 17: Average L2 miss latency of EMCC, Morphable, SC-64 and non-secure systems.

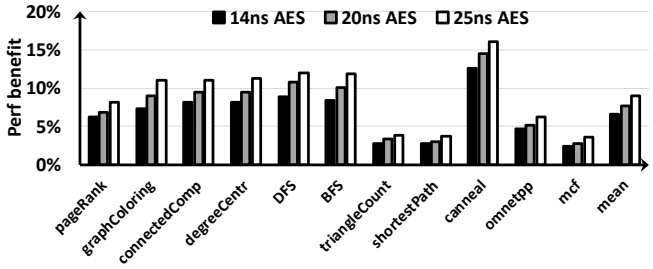


Figure 18: Performance improvement over Morphable Counters under different AES latencies: 14ns, 20ns, 25ns.

The increasing benefit of EMCC can be explained by different critical paths of Secure Memory Access for baseline and EMCC. In baseline designs, counter-mode AES is on the critical path of Secure Memory Access (see Figure 8). As such, increasing AES latency increases Secure Memory Access latency. Under EMCC, however, AES for a data block finishes much earlier than when the data block arrives at L2 (see Figure 13a); as such, data accesses are not delayed by AES even when AES takes longer.

B. Sensitivity Analysis on AES Bandwidth at L2

To determine how much AES bandwidth to move from MC to L2s, we evaluate moving different fractions of AES bandwidth from MC to L2s. Figure 19 shows what fraction of DRAM data accesses are decrypted and verified at L2s when moving different fractions of AES bandwidth to L2s. When distributing half of the AES bandwidth from MC to L2s, EMCC decrypts and verifies 76.3% of DRAM data accesses at L2, on average.

Figure 19 shows that for mcf, EMCC only decrypts and verifies 50% of DRAM accesses in L2s. This is due to the high memory access rate under mcf (see Figure 15). mcf's high memory access rate causes spikes that exceed the AES bandwidth that EMCC moves from MC to L2; as such, L2 offloads decryption and verification back to MC for many memory accesses.

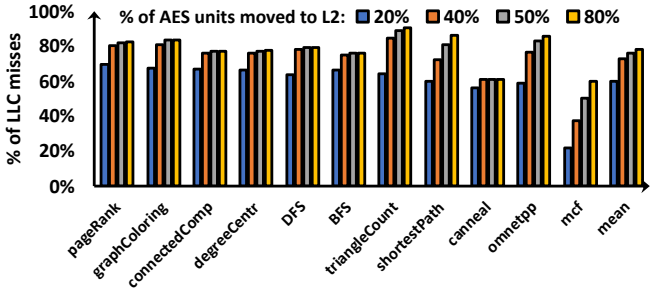


Figure 19: The number of DRAM data accesses decrypted and verified by L2s under EMCC. The number is normalized to the total number of DRAM data accesses.

C. Sensitivity Analysis on Counter Cache Size

We also perform a sensitivity analysis on counter cache size. We increase MC's counter cache to 256KB and 512KB. Figure 20 shows the performance benefit of EMCC when MC uses bigger counter caches. With bigger counter cache, the benefit of EMCC decreases because bigger counter caches reduce the number of counter accesses to LLC. Because the benefit of EMCC comes from hiding the latency of counter accesses in LLC, fewer counter accesses to LLC decreases the benefit of EMCC. However, the decrease in benefit is less than 1%; we find counter cache miss rates decrease little as counter cache grows in size. Counter cache miss rate reduces from 35%, on average, down to 31%, on average, when increasing from 128KB to 512KB.

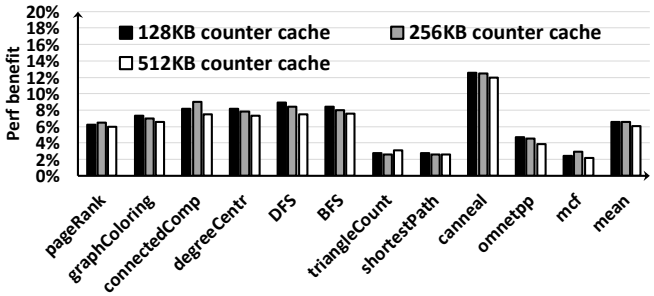


Figure 20: Performance benefit over Morphable Counters under different sizes of counter caches.

D. Sensitivity Analysis on DRAM Bandwidth

To evaluate the benefit of EMCC under higher DRAM bandwidth, we also evaluate EMCC under eight channels. As shown in Figure 21, the performance benefit of EMCC over Morphable Counters increases under eight channels. When modeling eight channels, we use bits 8 to 10 of the memory address of each incoming memory request as the 3-bit channel ID to access the corresponding memory channel.

The increasing benefit is due to faster data access under higher memory bandwidth. As can be seen in Figure 8, reducing data access latency (e.g., shrinking the two blue data access bars in Figure 8) would worsen the overhead of the baseline (e.g., would lengthen the "Overhead" arrow in Figure 8). When the baseline incurs more overhead, EMCC can provide more improvement over the baseline since EMCC is designed to hide the latency overhead.

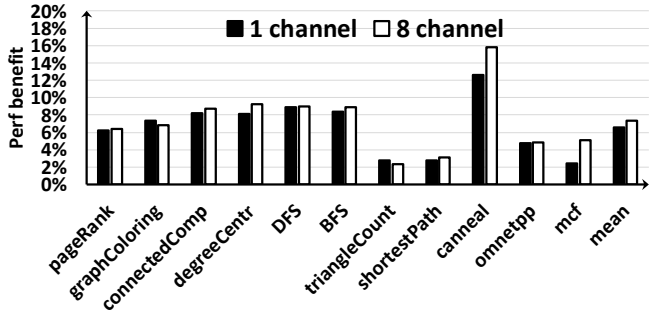


Figure 21: Performance benefit over Morphable Counters under single memory channel and eight memory channels.

The shorter DRAM latency under high memory bandwidth is due to reduced queuing delay. We measure the queuing delay of an individual memory request as the amount of time between when the request enters MC's read/write queue and when MC issues the first DRAM command (e.g., ACT or READ) for the request. Figure 22 shows the queuing delay of data and counter accesses.

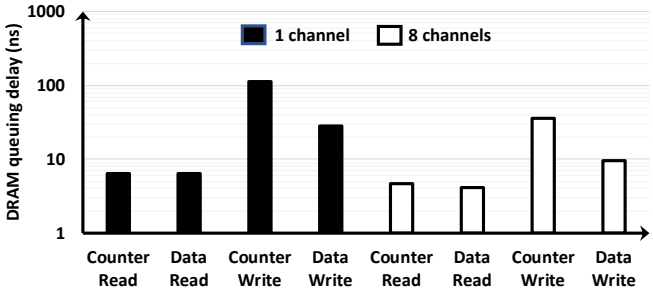


Figure 22: Queuing delay of different types of memory accesses under EMCC, on geometric mean across the evaluated set of benchmarks. Queuing delay reduces with more channels. In general, writebacks to memory experiences higher queuing delay than reads as they are deprioritized relative to reads (latency of writebacks do not affect program performance, unlike memory reads).

E. Effect of Cache Coherence on Caching Counters in L2

When writing back a block to DRAM, MC must update the block's counter. If an L2 caches a copy of a counter block, the L2 copy must be invalidated when MC updates the block; in general, writing to a cacheline in a multi-core CPU requires invalidating other copies of the cacheline. Invalidating counter blocks in L2 may reduce counter hit rate in L2. We measure the number of counter blocks invalidated in L2 under EMCC. On average, only 1.7% of counter blocks inserted into L2 are invalidated (see Figure 23).

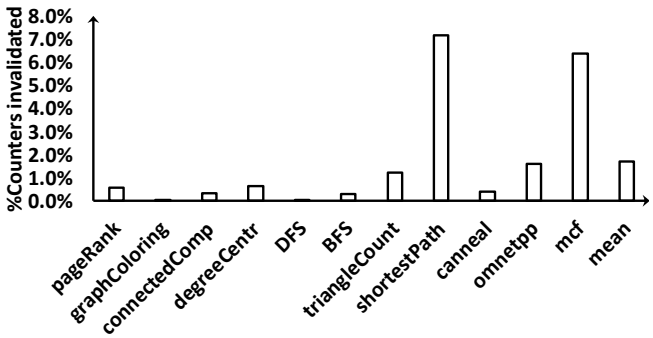


Figure 23: The number of counter block invalidation in L2, normalized to counter block insertions into L2.

F. Impact on NoC Traffic

EMCC can incur some useless accesses to LLC; they are few (see Figure 11), however. For completeness, we also measure the number of useless counter accesses in LLC for more benchmarks. Figure 24 shows the number of useless counter accesses in LLC for the other benchmarks in SPEC 2017 and Parsec 3.0, normalized to the number of data accesses in LLC. EMCC suffers from only 1% useless counter accesses in LLC, on average.

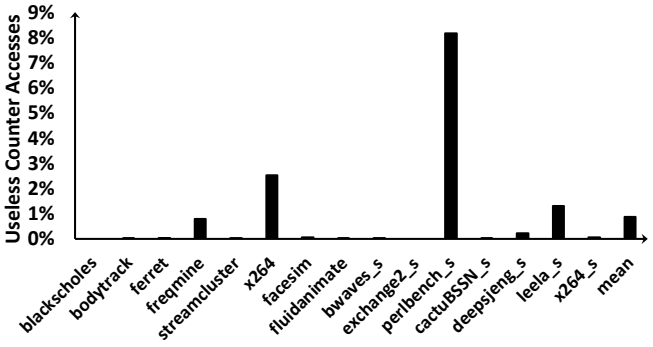


Figure 24: The number of useless counter accesses in LLC for other benchmarks in SPEC and PARSEC. All numbers are normalized to the total number of L2 data misses.

VII. Related Work

OTP Pre-computation: To hide the latency overhead of decryption for memory accesses whose counters miss in caches, OTP Prediction [42] proposes pre-computing (i.e., predictively computing) AES results using predicted counter values, similar to prefetching (i.e., predictively fetching) using predicted memory addresses. OTP Prediction uses the counter block of some data blocks in a page to predict the counter values for the other data blocks in the same page. However, split counters, like Morphable, eliminate the need to predict the counter values of other blocks in a page because a single counter block covers an entire page.

Pre-calculation for Persistent Memory: Many prior works [43][44][45][46] hide *encryption* latency in the context of persistent memory systems. In persistent memory programs, writes to persistent memory are on the critical path of program execution. To hide the latency of encryption for writes to persistent memory, Janus [43] provides a new software interface for programmers to explicitly initiate hardware encryption for persistent memory writes sooner. Our paper, however, focuses on hiding the latency of memory decryption and verification for reads from DRAM. Also, we propose purely microarchitecture-level optimizations, without any changes to software.

Lightweight Ciphers: Emerging lightweight ciphers, like QARMA [32], are faster than AES. However, trust is important to trusted computing; newer ciphers are not as trusted as AES, which has withstood longer and more scrutiny. Newer ciphers are typically be used where AES cannot. For example, ARM uses QARMA-64 to authenticate pointers, which are 64-bits; AES is undefined for 64-bit inputs. However, AES remains the standard for most systems.

VIII. Conclusion

Due to the large memory memory footprint and irregular access patterns of many real-world applications, counter caches are too small to provide high counter hit rate. Caching counters in LLC is a promising solution to reduce counter miss rate. However, the long LLC access latency in modern server CPUs not only can significantly diminish the benefit of caching counters in LLC, but can actually significantly increase counter access latency compared to not caching counters in LLC.

We note that the root problem lies with MC sitting behind LLC; MC can only see LLC misses and, thus, can only fetch and use counters serially after data miss in LLC. As such, we propose Eager Memory Cryptography in Caches to offload decryption and verification tasks from MC to L2 to enable parallel fetch and use of counters with data accesses. EMCC improves performance by 7% when applied to the state-of-the-art prior work. The maximum benefit is 12.5%.

Acknowledgment

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies. Authors would like to thank Thomas Woller and Mike Ignatowski for the feedback on the earlier versions of this paper. This work was supported by National Science Foundation (NSF) under grant 1850025. We thank Advanced Research Computing (ARC) at Virginia Tech for providing computational resources. We also thank the reviewers for providing helpful comments to improve the paper.

Appendix

One-way-trip latency calculation: Direct LLC Latency consists the two-way NoC latency between L2 and LLC, in addition to SRAM latency in LLC. We consider SRAM

latency to be at most 4ns for two reasons. First, SRAM latency in LLC tile is close to SRAM latency in L2 because the LLC tile size (i.e., 1.3MB) is close to L2 size (i.e., 1MB). Second, we use Cacti [47] to estimate the access latency of an 1MB 32nm SRAM under the slowest setting (i.e., sequential tag and data lookup, slow LSTP transistor, conservative interconnection, and optimized for area); Cacti also reports 4ns. As such, one-way NoC latency = (19ns - 4ns)/2 = 7.5ns.

MC-to-L2 route: As described in Figure 2.6.b in [48], when an access misses in L2 and then misses in LLC, the response for L2 miss first travels from MC to an LLC tile, which then forwards the response to L2. To verify the route for L2 miss response, we also measured in a real system the end-to-end per-access latency under DRAM row buffer miss. To achieve row buffer miss, we write a microbenchmark with 8KB stride and dependent accesses. To avoid row conflicts, we just run one process of the microbenchmark, with 16 ranks in DRAM. The measured latency is 77ns. Excluding the 30ns row buffer miss latency, the remaining 47ns = 77ns - 30ns is nearly twice of 23ns LLC hit latency. Therefore, we consider this remaining latency to primarily come from L2-to-LLC access latency and LLC-to-MC access latency. We reach the same conclusion when using a small-stride microbenchmark to generate row buffer hits in DRAM.

REFERENCES

[1] S. Gueron, "A memory encryption engine suitable for general purpose processors," Cryptology ePrint Archive, Paper 2016/204, 2016, https://eprint.iacr.org/2016/204.

[2] G. Saileshwar, P. Nair, P. Ramrakhyani, W. Elsasser, J. Joao, and M. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 416–427.

[3] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 179–190, 2006.

[4] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 454–465.

[5] J. Lee, T. Kim, and J. Huh, "Reducing the memory bandwidth overheads of hardware security support for multi-core processors," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3384–3397, 2016.

[6] 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)*. IEEE, 2007, pp. 183–196.

[7] 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.* IEEE, 2003, pp. 351–360.

[8] B. Gassend, G. E. Suh, D. Clarke, M. V. Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2003.

[9] "The nexus difference," Last accessed on April 13, 2022. [Online]. Available: https://www.nexustechnology.com/technologies/the-nexus-difference/

[10] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 665–678.

[11] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched address translation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 1023–1036.

[12] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing tlb reach by exploiting clustering in page translations," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 558–567.

[13] C. H. Park, T. Heo, J. Jeong, and J. Huh, "Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 444–456.

[14] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 469–480, 2017.

[15] C. H. Park, S. Cha, B. Kim, Y. Kwon, D. Black-Schaffer, and J. Huh, "Perforated page: Supporting fragmented memory allocation for large pages," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 913–925.

[16] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory hierarchy for web search," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 643–656.

[17] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," *Acm sigplan notices*, vol. 47, no. 4, pp. 37–48, 2012.

[18] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 158–169.

[19] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "Bolt: a practical binary optimizer for data centers and beyond," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 2–14.

[20] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for tlb prefetching: An application-driven study," in *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 2002, pp. 195–206.

[21] D. Lustig, A. Bhattacharjee, and M. Martonosi, "Tlb improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level tlbs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 1, pp. 1–38, 2013.

[22] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 48–59, 2010.

[23] J. Navarro, S. Iyer, and A. Cox, "Practical, transparent operating system support for superpages," in *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*. Boston, MA: USENIX Association, Dec. 2002. [Online]. Available: https://www.usenix.org/conference/osdi-02/practical-transparent-operating-system-support-superpages

[24] A. Bhattacharjee, "Translation-triggered prefetching," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 63–76.

[25] J. Ahn, S. Jin, and J. Huh, "Revisiting hardware-assisted page walks for virtualized systems," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 476–487.

[26] C. H. Park, I. Vougioukas, A. Sandberg, and D. Black-Schaffer, *Every Walk's a Hit: Making Page Walks Single-Access Cache Hits*. New York, NY, USA: Association for Computing Machinery, 2022, p. 128–141. [Online]. Available: https://doi.org/10.1145/3503222.3507718

[27] O. Levi, "Pin - a dynamic binary instrumentation tool," 2012. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

[28] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: understanding graph computing in the context of industrial solutions," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[29] LDBC Graphalytics, "Datasets," Last accessed on April 13, 2022. [Online]. Available: https://graphalytics.org/datasets

[30] "Memory and cache latency comparisons." [Online]. Available: https://pics.computerbase.de/7/9/1/0/2/13-1080.348625475.png

[31] "Skylake (server) - microarchitectures - intel." [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)

[32] R. Avanzi, "The qarma block cipher family. almost mds matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes," *IACR Transactions on Symmetric Cryptology*, pp. 4–44, 2017.

[33] "Skylake mesh architecture." [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html

[34] T. S. Lehman, A. D. Hilton, and B. C. Lee, "Poisonivy: Safe speculation for secure memory," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[35] "Spec cpu2017 platform settings for supermicro systems." [Online]. Available: https://www.spec.org/cpu2017/flags/Supermicro-Platform-Settings-V1.2-CLX-revB.html

[36] Intel, "Hpc cluster tuning on 3rd generation intel xeon scalable processors," 2021, https://www.intel.com/content/dam/develop/external/us/en/documents/HPC-Cluster-Tuning-Guide-on-3rd-Generation-Intel-Xeon-Scalable-Processors.pdf.

[37] "Dell emc poweredge r550 bios and uefi reference guide," Last accessed on April 13, 2022. [Online]. Available: https://www.dell.com/support/manuals/ensg/poweredger550/per550_bios_ism_pub/processor-settings?guid=guid-45a5b59a-2907-44f6-9f14-8d83a15b6969&lang=en-us

[38] "Tuning uefi settings for performance and energy efficiency on intel xeon scalable processor-based thinksystem servers," [Online] Last accessed on April 13, 2022. https://lenovopress.lenovo.com/lp1477.pdf.

[39] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[40] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, vol. 15, no. 1, pp. 45–49, 2015.

[41] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Drama: Exploiting dram addressing for cross-cpu attacks," in *Proceedings of the 25th USENIX Conference on Security Symposium*, ser. SEC'16. USA: USENIX Association, 2016, p. 565–581.

[42] W. Shi, H.-h. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, "High efficiency counter mode security architecture via prediction and precomputation," in *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 2005, pp. 14–24.

[43] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing memory and storage support for non-volatile memory systems," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 143–156.

[44] A. Freij, S. Yuan, H. Zhou, and Y. Solihin, "Persist level parallelism: Streamlining integrity tree updates for secure persistent memory," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 14–27.

[45] Z. Zhang, J. Yue, X. Liao, and H. Jin, "Efficient hardware-assisted crash consistency in encrypted persistent memory," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 750–755.

[46] M. Alwadi, A. Mohaisen, and A. Awad, "Promt: optimizing integrity tree updates for write-intensive pages in secure nvms," in *Proceedings of the ACM International Conference on Supercomputing*, 2021, pp. 479–490.

[47] HP, "Cacti. an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model." [Online]. Available: https://www.hpl.hp.com/research/cacti/

[48] N. E. Jerger, T. Krishna, and L.-S. Peh, "On-chip networks," *Synthesis Lectures on Computer Architecture*, vol. 12, no. 3, pp. 1–210, 2017.