Counter-light Memory Encryption

Xin Wang*, Jagadish Kotra†, Alex Jones‡, Wenjie Xiong*, Xun Jian*
*Virginia Tech †AMD ‡ University of Pittsburgh
xinw@vt.edu, jagadish.kotra@amd.com, akjones@pitt.edu, wenjiex@vt.edu, xunj@vt.edu

***Abstract*—Unlike the well-known counter mode memory encryption (e.g., SGX1), more recent memory encryption (e.g., SGX2, SEV) has no counters. Without accessing any counters, such *counterless* memory encryption improves performance over counter mode encryption and gains wide adoption as a result.**

Counterless encryption, however, still incurs a costly performance overhead. Under counterless encryption, the cipher calculations take data as their direct inputs. As such, the ciphers for decrypting data can only be calculated sequentially after the missing data arrive from memory; this requires *every* last-level cache miss to stall on the cipher calculations after the needed data arrive from memory. Our real-system measurements find counterless encryption can slow down irregular workloads by 9%, on average.

We observe while *counter mode* encryption incurs costly memory access overhead, its cipher calculations can often complete before data arrive because they take *counters* as input, instead of data, and counters can fit on-chip much better than data. As such, we explore how to combine both modes of encryption to achieve the best of both worlds – the efficient memory accesses of counterless encryption and fast cipher calculations of counter mode encryption. For irregular workloads, our proposed memory encryption – *Counter-light Encryption* – achieves 98% the average performance of no memory encryption. When memory bandwidth is starved, Counter-light Encryption is slower than counterless encryption by only 1.4% in the worst case.

I. INTRODUCTION

While many companies are moving to cloud to reduce computing cost, cloud computing raises new security and trust concerns as companies lose control over physical accesses to the servers running their applications. To protect the confidentiality of sensitive data, the CPU-side memory controller (MC) encrypts memory values to protect data from attackers.

Older memory encryption (e.g., SGX1 [28]) provides not only confidentiality, but also protection against physical replay attacks (i.e., reverting memory blocks to older values via physical access to memory). Physical replay protection is achieved by encrypting each block with a counter that increases whenever writing the block to memory and protecting each counter with a tree of counters called the integrity tree.

The counters, however, incur both costly memory bandwidth and capacity overheads. Prior Split Counters designs [62] [70] [67] effectively reduce the memory ***capacity*** overhead down to just 1%; however, the costly ***bandwidth*** overhead to access the counters still remains. While caching counters can practically eliminate the bandwidth overhead for ***regular*** workloads, ***irregular*** workloads suffer from high counter cache miss rates and, thus, costly memory bandwidth overheads.

Mainstream memory encryption today eliminates all counter accesses by eliminating counters all together. We refer to this as *counterless encryption*. Examples include SGX2 [43], TME [36], MKTME [35], SME [46] [4], SEV [6], SEV-SNP [7]. Without counters, they only provide encryption and integrity check, but do not protect against ***physical replay*** attack; they only protect against ***software replay*** attacks [7] [35]. Cloud data centers like Microsoft Azure, AWS, and Google cloud are deploying counterless encryption widely in their cloud servers. For example, on AWS, users can launch an Amazon EC2 instance with SEV turned on [3].

Problem: Counterless encryption, however, still incurs a costly performance overhead due to its slow cipher calculation (i.e., AES [20]) for decrypting data; the cipher calculation slows down *every* LLC (last-level cache) read miss due to starting after the arrival of data, which are the direct inputs to the cipher calculations. While prefetching can effectively hide this latency overhead for ***regular*** workloads, it cannot for ***irregular*** workloads – the important workloads that motivated the move from counter mode encryption to counterless encryption. On an Intel Silver 4314 CPU, we find turning on TME, which uses counterless encryption, slows down irregular workloads by 9%, on average (see Section III).

Observation: We observe while counter mode encryption ***accesses memory slower*** than counterless encryption, it ***calculates cipher faster***. Its cipher calculations can often complete before data arrive because they take ***counters*** as input, instead of data, and counters fit on-chip much better than data. While counter ***blocks*** suffer frequent cache misses under irregular workloads, the AES results of counter ***values*** can still be effectively memoized [74]; a ***single*** counter ***value*** can be simultaneously used by many (e.g., millions of) data blocks, allowing the counter values needed by many (e.g., $> 90\%$ of) LLC read misses to hit in a small memoization table [74]. When a counter arrives from DRAM, ***fetching*** the memoized AES result of the counter's value bypasses the long latency of ***recalculating*** the AES result from scratch.

Key Idea: To improve performance over counterless encryption, we propose combining aspects of counter mode encryption with counterless encryption to achieve the best of both world in performance – the efficient memory accesses of counterless encryption and fast cipher calculations of counter mode encryption. Some current systems (e.g., the one we evaluated in Section III) already contain both modes of memory encryption to give users multiple memory security options; under the counterless encryption security option, utilizing parts of the unused counter mode encryption to boost performance also improves the utilization of valuable on-chip resources.

We refer to our proposed memory encryption combining

	LLC Read Miss	LLC Writeback	Protects Against	Memory Overhead
Counterless	No overhead accesses; Always calculate cipher after data arrives.	No overhead accesses.	Physical (or software) probing and non-replay-based tampering.	No memory overhead.
Counter-light (Our work)	No overhead accesses; Calculate cipher after data arrives only if counter misses in AES memoization table.	Overhead accesses only in epochs with spare bandwidth.	Same as counterless; also faces the same consequences under replay attacks.	Same as counter mode.
Counter mode	Always access counters; Calculate cipher after data arrives only if counter misses in AES memoization table.	Always access counters.	Physical (or software) probing and all tampering including replay.	1.6% of memory, assuming split counters design.

Fig. 1: Comparing counterless, Counter-light, and counter mode encryption. Counter-light Encryption achieves the best of both worlds in performance.

both modes as *Counter-light Encryption*. It encodes each data block's current encryption mode and counter value (if any) into the block's chipkill-correct ECC. During each LLC writeback, encoding the data block's encryption mode in the block's ECC allows Counter-light Encryption to dynamically decide and record the data block's new encryption mode without needing any overhead access to record the encryption mode somewhere else in memory; as such, the blocks being written can seamlessly switch to counterless encryption ***for free*** (i.e., no bandwidth overhead involved) to turn off ***all*** bandwidth overheads during fine-grained (e.g., 100us) epochs with high memory bandwidth utilization (i.e., when bandwidth overhead is the most harmful). For each LLC read miss, Counter-light Encryption decodes the arriving data block's encryption mode and counter value (if any) from the block's ECC to decrypt the arriving data mostly using the fast ***counter mode*** decryption without any overhead access to memory to fetch counters like ***counterless encryption***.

This paper makes the following contributions:

- We explore and address the performance overhead of counterless memory encryption.
- We explore how to combine aspects of counter mode encryption with counterless encryption to improve performance over counterless encryption alone. Our proposed Counter-light Encryption achieves the best of both worlds - the fast memory accesses of counterless encryption and fast cipher calculations of counter mode (see Figure 1).
- Our evaluation shows Counter-light Encryption allows irregular workloads to retain 98% of their average performance over no encryption. Under high memory bandwidth utilization, Counter-light Encryption is only 1.4% slower than counterless encryption in the worst case.

II. Background and Related Work

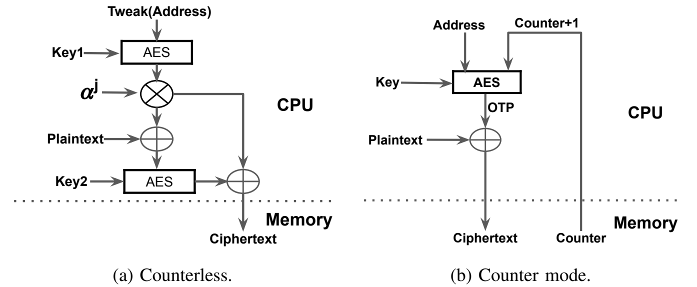
To enforce the confidentiality of memory in cloud, server CPUs encrypt memory blocks before writing them to memory and decrypt them after reading them from memory. The most time-consuming part of encryption and decryption is cryptography calculation.

The Advance Encryption Standard [20] (AES) is the most common cryptography algorithm used by memory encryption. AES has a fixed size of 128 bits for both input plaintext and output ciphertext. AES encryption has different encryption modes [66] (e.g., CTR, XTS, GCM, CFB, ECB, CBC), which take different inputs. Encryption modes like AES-XTS uses data values and addresses as the inputs to the AES block cipher calculation. AES-CTR uses counters and addresses (but not data) as inputs.

A. Counterless Memory Encryption

Many flavors of counterless memory encryption are being used today. Total Memory Encryption (TME) [36] and Secure Memory Encryption (SME) [4] [46] perform system-level encryption to encrypt the entire memory system using one key. Memory Encryption-Multi-Key (TME-MK) [35] and Secure Encrypted Virtualization (SEV) [46] [45] [7] perform per-VM encryption to encrypt different VMs using different keys.

Intel SGX2 [23], Intel TME [36] and Intel TME-MK [35] use AES-XTS [56]. AES-XTS is also used [5] by AMD SME [4] and AMD SEV (+ES/SNP) [53] [46] [22]. AES-XTS uses data values and address to calculate AES (see Figure 2a).



(a) Counterless.

(b) Counter mode.

Fig. 2: (a) Counterless encryption. For each 16B word, it performs two AES calculations – an address-only AES and a data-dependent AES. 'Tweak(Address)' is a nonnegative tweak value assigned to each 64B block; while is a constant across different blocks, j is the sequence number of the 16B word in the data block. (b) Counter mode encryption. For each 16B word, it performs a single AES calculation using both counter and data. The 'Address' is for a 16B word, while the 'Counter' is for a 64B memory block.

Threat Model and Protection: Counterless encryption is designed for a threat model where attackers have physical access to victim systems and, thus, can snoop and tamper with the data sent over the CPU-memory bus. Malicious personnel with physical access to Cloud servers (e.g., disgruntled Cloud employees) can tamper with the memory values of applications migrated to Cloud. An attacker can use one of the many existing commercial off-the-shelf memory bus probes, intended for system-level integration test and debugging, to probe or even modify values transmitted over the memory bus.

Counterless encryption protects against malicious probing of memory values on the memory bus because it encrypts

memory blocks before writing them to DRAM. To protect memory from tampering, some counterless encryption designs [35] maintain message authentication codes (MACs) per data block. The MAC is calculated from data through a hash function (e.g., SHA-3 under Intel MKTME [35]) and stored in DRAM during the LLC writeback. During each LLC miss, the memory controller (MC) recomputes the MAC and compares it to the stored MAC to check for MAC mismatch and detect tampering. MKTME [35] protects each block with a 28-bit MAC and takes away some of the block's ECC bits to store the MAC [17].

Counterless encryption, however, provides no protection against replay attack over the entire memory block [43] [36] [35] [46] [45] [7]. An attacker can revert a memory block to its earlier state by overwriting the block and its MAC with its older value and older MAC. Because the older MAC is consistent with the older block value, the MAC check will pass when the replayed block is accessed later.

B. Counter Mode Memory Encryption

Older memory encryption like Intel SGX1 [28] use counter mode (e.g., AES-CTR [54]), which encrypts and decrypts data blocks through One Time Pads (OTPs).

The ciphertext for data is the result of bitwise xoring the OTP with data (see Figure 2b). The AES calculation for generating OTP takes the block's write counter and address as inputs. Because the XOR of two ciphertexts using the same OTP is identical to XOR of two plaintexts, re-using counter values will leak the plaintext information. As such, each write counter is a nonce (number only used once) that increases each time the block is written to memory.

On an LLC read miss, after the data block arrives from DRAM, the MC uses the block's counter to recompute the same OTP previously used to encrypt the block and then bitwise XOR the recomputed OTP with ciphertext to recover the block's original plaintext. Unlike data, counters are stored in memory directly as plaintext.

Threat Model and Protection: Like counterless encryption, counter-mode encryption also protects against malicious probing of memory values; it also protects against tampering by protecting each block with a MAC. However, it requires stronger MACs than counterless encryption due to being more vulnerable to malicious tampering of data; because $Plaintext = Ciphertext \oplus OTP$, an attacker can intentionally flip a ***specific*** bit in the plaintext by flipping the same bit in the ciphertext. Counterless encryption is secure against such intentional attacks on specific bit(s) because flipping one bit in ciphertext will randomly flip half of the bits in the plaintext. As such, each MAC is 56 [28] or 64 bits [63] and is calculated from OTP instead of SHA-3. The MAC is the bitwise XOR between a truncated OTP and a truncated Galois Field dot product on the plaintext and secret keys [28].

With a MAC alone, however, each data block is still vulnerable to replay attack like under counterless encryption. When an attacker replays an old tuple of {Data, MAC, counter} to the CPU, the MAC cannot detect replay because the replayed data and counter are consistent with the replayed MAC. Detecting such replayed tuples requires detecting counter replay.

To detect counter replay and, thus, detect full replay of {Data, MAC, counter}, counter-mode memory encryption has a tree of counters called the *integrity tree* [28]. Each leaf counter in the integrity tree protects multiple data's counters (e.g., eight under Intel SGX1 [28]); each leaf and the data's counters the leaf protects collectively calculate a MAC that is stored in memory. Similarly, for each parent counter and its child counters, the integrity tree maintains a MAC in memory. When incrementing a counter for an LLC writeback, the MC fetches the integrity tree nodes protecting the counter to increment a counter in each fetched node; the MC also updates the MACs in the fetched nodes using their new counter values. As such, a successful replay attack requires also replaying multiple counters in the integrity tree. But the root counter of the integrity tree is ***always*** stored in the CPU, where it cannot be replayed; as such, it can guarantee to detect any possible replay attack in DRAM.

Verifying a counter's integrity requires fetching all related counters (e.g., the counter's parent and other counters protected by the parent) to recalculate a MAC to compare against the MAC in memory. To minimize the memory accesses to fetch the related counters for calculating a MAC, all related counters in the same level are colocated in a 64B memory block so that they can be fetched together in one access instead of many (e.g., eight) memory accesses; these related counters are stored in one **integrity tree block**. Similarly, each **counter block** stores multiple data's counters (e.g., the eight counters of eight adjacent data blocks under SGX1).

C. Prior Works on Memory Encryption

Many prior works have optimized memory encryption (e.g., [62] [28] [76] [77] [63] [70] [52] [73] [74] [51]) in the context of counter mode encryption. As our proposed Counter-light Encryption seeks to boost performance over counterless encryption by combining both types of encryption to achieve the best of both worlds in performance, we describe below relevant optimizations for counter mode encryption.

Counter Storage Overhead: To minimize the storage overhead of the counter blocks for data and the integrity tree, a large body of works over several decades have explored split counters [62] [70] [76] [33] to pack more counters per block. With split counters, counter blocks and integrity tree nodes collectively take up $1.6\% \sim 3.2\%$ of memory.

MAC Storage overhead: A prior work, Synergy [63], stores a 64-bit MAC in the ECC bits of each block while still ensuring chipkill-correct, an industry-standard server memory feature that protects against errors in any single memory chip per rank (i.e., **single-chip error**). Explicitly, Synergy uses MACs to replace the error detection bits in chipkill-correct ECC. Using the MAC to replace the detection bits saves space compared to adding MAC on top of the full chipkill-correct ECC. Figure 3 shows the layout of each data block under Synergy [63] in DDR5 server memory, which typically provides 16B of ECC storage per 64B of data. Synergy uses

8B of the 16B ECC storage to store an 8B MAC for **both error detection and integrity check**. The remaining 8B is an 8B parity for error correction; the parity is calculated as $Parity = D1 \quad D2 ... \quad D8 \quad MAC$, where Di is the block's data stored in chip i (see Figure 3).

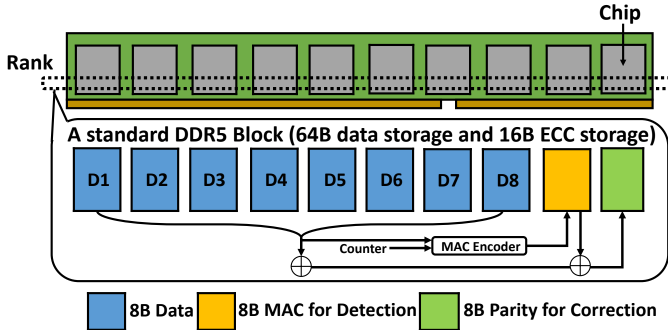


Fig. 3: Memory block layout under Synergy in a ***standard*** DDR5 DIMM, which has 8 data chips + 2 ECC chips = 10 chips per rank. The eight chips store $8 \cdot 8 = 64B$ of data per block and the two ECC chips can store an 8B MAC and an 8B parity per block. Synergy is widely used in many later works on memory encryption [62] [80] [24] [58] [31] [69] [32] [73] [75].

When detecting an error via the MAC, Synergy corrects the error via trial and error. In the first trial, it assumes the chip storing *D1* is faulty and reconstructs D1 as $D1^{\prime} = Parity \quad D2 \quad D3 ... \quad D8 \quad MAC$ and checks whether *MAC(D1', D2, ..., D8, counter)* matches the fetched MAC; *MAC(...)* means to the MAC encoding function. Synergy then repeats seven more trials, each assuming the data chip storing *D2* or $D3$ or ... or $D8$ is faulty. Synergy also repeats a trial assuming the MAC chip is faulty and reconstructs the MAC as $MAC^{\prime} = Parity \quad D1 \quad D2 ... \quad D8$ to check whether $MAC^{\prime} == MAC(D1, D2, ..., D8, counter)$. Synergy reports the error is uncorrectable if either no trial has MAC match or multiple trials have MAC match.

Synergy is applicable to the MAC used under both counter mode encryption and counterless encryption. However, by focusing on the MAC, Synergy only optimizes integrity checks, but not decryption.

Reducing Cipher Calculation Overhead for Irregular Workloads: Irregular workloads have poor low spatial locality; as such, they can suffer from high counter cache miss rate. On a counter miss, fetching the counter from DRAM and ***then*** using it to calculate AES to decrypt data can significantly slow down each LLC read miss; this increases the LLC read miss latency to the sum of DRAM access latency and AES calculation latency, instead of just the DRAM access latency.

Prior work (RMCC [74]) memoizes the AES results calculated from counters to speed up irregular workloads. When a missing counter arrives from memory, RMCC uses the counter's value to look up a memoization table to quickly obtain the counter's cipher output instead of slowly recalculating from scratch (see Figure 4).

To achieve high table hit rate, RMCC enhances the counter update policy for LLC writebacks to increase the counter value to values whose results are memoized in the table. RMCC enables 90% of LLC misses to benefit from memoization table hit even for irregular workloads.

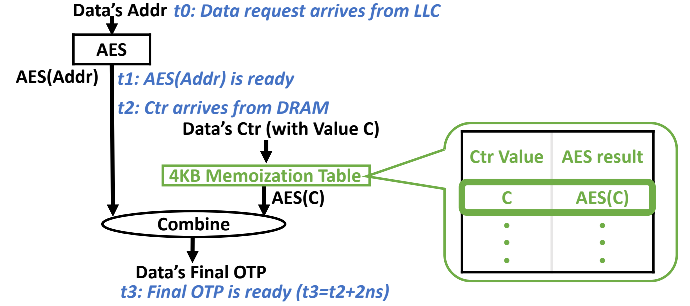


Fig. 4: RMCC [74] memoizes the AES results of counters to quickly use them when a missing counter arrives from memory, instead of slowly using the arriving counter to recalculate AES. The final OTP, which combines an address-only AES and a counter-only AES, considers both counter and address like a regular OTP (see Figure 2.b).

While memoization effectively reduces the latency overhead of cipher calculation for irregular workloads, the high counter cache miss rate continues to incur high memory bandwidth overhead to access counters. Memoization does not help reduce any overhead memory accesses as looking up the memoize table for a data block requires the block's counter.

III. Characterizing The Problem

Without any counters, counterless encryption calculates ciphers using address and data, instead of address and counter. As such, to perform decryption for an LLC read miss, the data-dependent AES calculation (see Figure 2) can only start after the missing data block arrives. Because AES consists of multiple ***sequential*** rounds of linear and non-linear transformations (e.g., 10 rounds under the commonly-used AES-128), waiting for the slow AES calculation after the missing data have already arrived increases the end-to-end LLC miss latency compared to no encryption; without memory encryption, the missing data can be used immediately after arriving from memory, without waiting on the AES calculation latency.

We measure AES latency under counterless encryption in an Intel Silver 4314 CPU [2] with TME. We write a read-intensive microbenchmark with 128MB memory size to cause 100% miss rate in the 24MB LLC. We fixed the CPU frequency to the CPU's base frequency 2.4GHz. To minimize measurement noise, we ensure only one memory access is issued at a time by turning off all prefetching and using pointer-chasing in the main access loop; we run the microbenchmark under 2MB standard huge pages to minimize TLB misses. We use RDTSC [38] to measure each LLC miss's latency and calculate the average across 10 experiments.

With counterless encryption turned on, the per-access memory latency is 10ns longer than when it is turned off. The measured 10ns difference is close to the AES latency reported in prior work [8] that synthesizes AES under 7nm technology.

To evaluate the performance impact on irregular workloads, we run the same benchmarks as previous memory encryption works on improving irregular workloads [73] [74]. Details of these benchmarks are discussed in Section VI.

We measure each workload's total execution time when TME is turned on to when it is turned off. With counterless encryption, the average performance drops to only 91% of without encryption (see Figure 5).

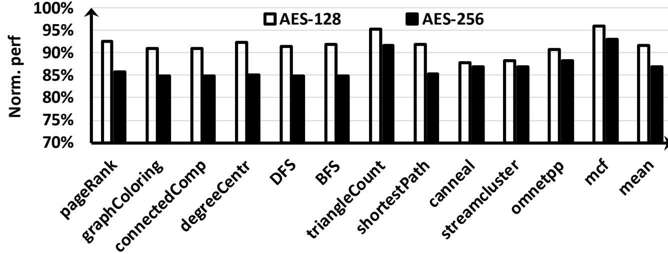


Fig. 5: Application performance with counterless encryption turned on normalized to when encryption is turned off. We evaluate AES-128 on a real-system and evaluate AES-256 via simulation.

The slowdown would increase under the stronger but slower AES-256, which is being adopted for memory [4] to achieve strong post-quantum cryptography; AES-256 has 14 sequential rounds so it is slower. We simulate AES-256 in Gem5 [11] because we do not have root access to a real system with AES-256 at the time of this writing. Since AES-256 has 14 rounds, we simulate the AES-256 latency as $(14/10) \quad 10ns = 14ns$ (see more details in Section V). As Figure 5 shows, the applications slow down by 13%, on average.

One naive approach to reduce the slowdown due to cipher calculations is to replace AES with faster lightweight ciphers proposed by prior work [13] [12] [8]. However, lightweight ciphers are weaker than AES, which contradicts the desire to adopt even stronger ciphers. For example, [39] shows how to exploit the structural linear relations that exist for PRINCE [13] to perform a key recovery attack.

IV. Counter-light Encryption

To improve performance over counterless encryption, we propose *Counter-light Encryption* to combine counterless and counter mode encryption to achieve the best of both worlds in performance – the efficient memory accesses of counterless encryption and fast cipher calculations of counter mode.

For LLC ***writebacks***, we combine both modes of encryption by dynamically switching between them depending on the memory bandwidth utilization at the granularity of short $100us$ epochs. Since bandwidth overhead is the most harmful when the memory bandwidth utilization is high, we propose to dynamically turn off all overhead accesses to counters when writing to memory during times of high bandwidth utilization by dynamically using counterless encryption for writebacks. To gracefully switch between using different encryption modes for writebacks across different epochs, Counter-light Encryption ***individually*** records in memory the mode used for ***each*** data block when writing it back to memory.

For *LLC **read misses*** to data blocks currently using counter mode, we combine both types of encryption by ***lightly accessing counters*** to only meet ***counterless security***. As such, for LLC read miss, Counter-light Encryption uses ***only a single*** counter – just the missing block's own counter.

Design Challenges: For each LLC ***writeback***, recording the written block's encryption mode to memory incurs a new bandwidth overhead that can prevent our goal of dynamically turning off all writeback bandwidth overheads during times of high memory bandwidth utilization. For an LLC ***read miss*** to each block currently using counter mode, the one remaining overhead access to the missing block's individual counter can still incur costly ***latency*** overhead because the counter access can sometimes complete later than the data access. The LLC read miss stalls at least until after the counter arrives.

Observation and Optimization: We observe each block's individual counter value and encryption mode record are small – small enough to fit within each data block itself. As such, we propose encoding them into each data block's chipkill-correct ECC in server memory, without sacrificing the amount of ECC in each block. Encoding a data block's counter into a data block's ECC ensures the counter always arrives at the same time as the data; encoding a data block's encryption mode into the block's ECC means recording the block's latest encryption mode on-the-fly with writing the block to memory without incurring any bandwidth overhead. Our final design strictly speeds up LLC read misses at the cost of incurring memory traffic overhead only for LLC writebacks (but not LLC read misses), and only when memory has spare bandwidth. Figure 1 compares and contrasts Counter-light Encryption with prior memory encryption designs.

A. Serving LLC Read Misses and the Design Challenge

When reading data blocks currently in counter mode, Counter-light Encryption uses only a single counter - the missing block's individual counter (see Figure 6.b). Accessing just a single counter is enough to match the security of counterless mode (i.e., providing encryption and integrity check, without physical replay protection [35] [7]). In comparison, on each LLC read miss, prior counter mode designs use one or more memory block(s) worth of counters (see Figure 6.a); using so many counters is to detect counter replay and, thus, all physical replay attacks (see Section II-B).

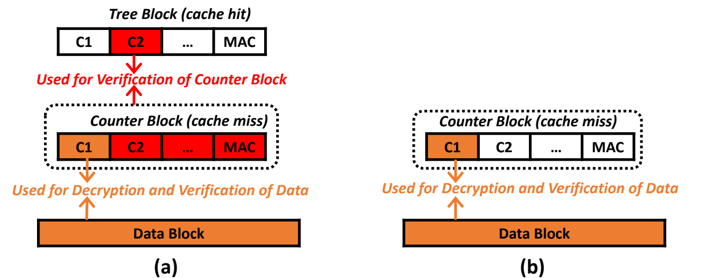


Fig. 6: (a) An example of how prior counter mode designs use counters to verify and decrypt data for LLC read misses. The example shows miss of counter block but hit of the counter block's parent counter. **All** counters in the missing counter block and the counter block's parent counter are used. (b) Using only a single counter per LLC read miss.

Compared to counter mode encryption, only using the data block's individual counter, instead of block(s) worth of

counters, can effectively reduce the bandwidth overhead of counter mode encryption.

Compared to counterless encryption, using the results memoized for the counter value to complete cipher calculations (see Figure 7.b) reduces LLC read miss latency over how counterless encryption calculates cipher from scratch after the missing counter arrives from memory (see Figure 7.a).

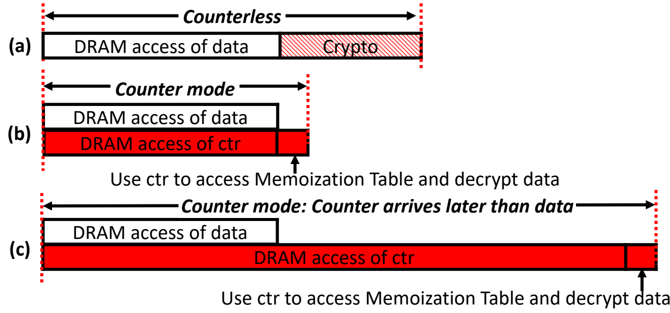


Fig. 7: (a) LLC miss under counterless encryption. (b) LLC miss under counter mode encryption with memoization (see Figure 4) when counter arrives at the same time as data. (c) The design challenge is that the counter can sometimes also arrive later than data, making counter mode decryption slower than counterless encryption.

Design Challenge: Even though Counter-light Encryption only uses one counter per LLC read miss, this one access can still incur a costly ***latency*** overhead as it can sometimes complete later than the data access and slow down the overall LLC read miss (see Figure 7.c). For example, the counter access can sometimes incur a row conflict, which is slow, while the data access experiences a row hit or row miss, which is faster. Furthermore, on an LLC read miss, counter access to DRAM always begins after data access and, therefore, often completes later. This is because each counter access to DRAM must first wait for the counter cache access to complete; speculatively accessing counter in DRAM without waiting for the counter cache to report whether the cache access hits or misses can result in unnecessary counter accesses to DRAM.

To quantify how often counters arrive later than data for LLC read misses, we simulate RMCC (see Section II-C) by using the same benchmarks and simulation methodology as RMCC. Figure 8 shows how often a data block's counter arrives from memory after the data block has arrived and by how much. Counter arrives later than data for 22% of all LLC misses. As such, this counter access ***alone*** reduces performance by 7%, on average (See Figure 9). It is almost as high as the total overhead of counterless encryption.

B. Serving LLC Writebacks and the Design Challenge

Unfortunately, to preserve the same security as counterless encryption in the presence of physical replay attack, LLC writebacks still require updating multiple blocks of counters, including counters in the integrity tree. Under counterless encryption, an attacker can only replay an old data block, revealing no new data. But under counter mode encryption, replaying a counter value prior to an LLC writeback can reveal the newly written data. As Figure 10 shows, through replaying

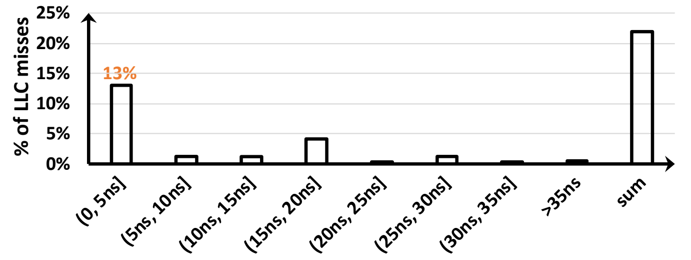


Fig. 8: Distribution of counter arrival time minus data arrival latency across all LLC misses (i.e., across both LLC misses that hit and miss in the counter cache). For example, counter access completes later than data access by $> 0ns$ but $\leq 5ns$ for 13% of all LLC misses.

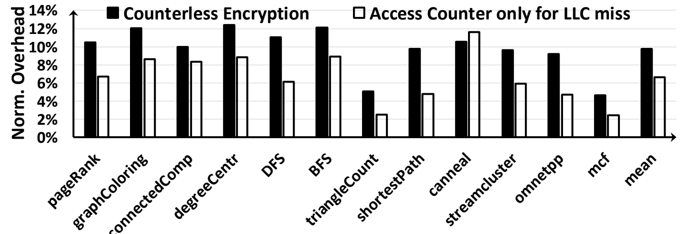


Fig. 9: The performance overhead ***strictly*** due to accessing the missing block's ***one*** counter on each LLC read miss; it is simulated by dropping all counter accesses for LLC writebacks and dropping all integrity node accesses across both LLC misses and writebacks. The performance overhead of counterless encryption in the ***same simulation window*** is shown as a reference.

a counter ③, an attacker can calculate the plaintext of the new data just by xoring one known plaintext ① and two observed ciphertexts ②④.

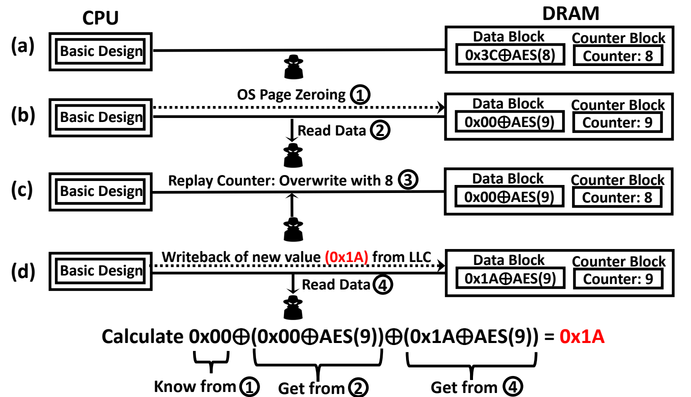


Fig. 10: An example showing how a physical replay attack on counter before LLC writeback can reveal the plaintext of new data (i.e., 0x1A).

Preventing this attack requires that ***for LLC writebacks***, Counter-light Encryption continue to access the integrity tree to verify the counter values are not replayed, in exactly the same way as traditional counter mode encryption.

As such, ***for LLC writebacks***, Counter-light Encryption still reintroduces costly accesses to the counter blocks and the integrity tree, even though it introduces no accesses to the integrity tree for LLC ***read misses***.

Even if we were to compromise on security by removing the integrity tree, a data block's counter must still be updated when writing the block to memory (or there would be no

counter mode encryption at all). The overhead access to ***just update the counter alone*** can incur a costly write overhead.

To minimize the performance overhead due to the bandwidth overhead for LLC writebacks, we note our goal of reintroducing counter accesses is only to improve performance over counterless encryption; as such, when counter accesses harm instead of improving performance, they can be turned off. As such, when memory bandwidth utilization is high, Counter-light Encryption dynamically turns off all writeback bandwidth overheads by switching to using counterless encryption for writebacks.

To measure bandwidth utilization, Counter-light Encryption counts the total number of memory accesses (i.e., LLC misses + writebacks + counter accesses) during each 100us epoch.

If the utilization is $< 60\%$ (i.e., if observed accesses exceeds a threshold number of accesses equal to 60% of the maximum possible accesses in the epoch), Counter-light Encryption uses counterless encryption for LLC writebacks during the ***next*** epoch; otherwise, Counter-light Encryption uses counter mode for writebacks in that new epoch until either the end of the epoch or the number of observed accesses in the epoch exceeds that same threshold number of accesses. Large studies report the median bandwidth utilization in cloud is only 10% [44] [25]; as such, a 60% threshold can allow most LLC writebacks to use counter mode encryption. The orange parts in Figure 11 illustrate how to dynamically switch the encryption mode.

To dynamically decide which encryption mode to use for LLC writebacks in fine-grained epochs, Counter-light Encryption cannot afford to globally re-encrypt all memory blocks during a new epoch. Instead, Counter-light Encryption ***individually records for every block its current encryption mode*** so that blocks receiving LLC writebacks can be switched dynamically to a new encryption mode without forcing unwritten blocks to also switch to the new mode.

The design challenge is how to address the overhead accesses to the per-block records of encryption modes. For every LLC writeback, Counter-light Encryption must also write to memory the encryption mode used for the writeback. An alternative is writing the encryption mode to memory only if it changes; however, this alternative incurs even more overhead because checking whether the mode changes for each writeback requires adding an extra read access to memory to fetch the block's encryption mode for each writeback, even if the written block's encryption mode remains the same.

Having to also write to memory a block's latest encryption mode for every LLC writeback would defeat our goal of dynamically turning off all writeback bandwidth overheads during times of high memory bandwidth utilization.

One potential solution is to store encryption mode bits in memory blocks and cache them like counter blocks. Each 64B block of encryption mode can serve 512 blocks; this is close to how each counter block under Split Counters serves 128 data blocks (see Section II-C). But our evaluations of irregular workloads (see Section VI) find that accesses to Split Counter blocks for LLC writebacks suffers from 98% miss rate in the counter cache. As such, we expect the encryption mode blocks would also suffer from high miss rate for LLC writebacks.

Worse, the encryption mode of a block must also be read from memory for every LLC miss so that MC can use the correct decryption mode for the block; having to ***also*** wait on the encryption mode block, ***in addition to counter block***, will further slow down read LLC misses.

C. Addressing the Unique Challenges

To address the unique design challenges that Counter-light Encryption faces for LLC read misses and writebacks, we note the extra information Counter-light Encryption needs per memory access are small – just the block's encryption mode and ***one*** counter. Being small, they can fit in the data block itself to enable Counter-light Encryption to update and access them ***without any overhead accesses***. 'Fitting' a block's counter in the data block itself also ensures the counter always arrives at the same time as the data and eliminate the latency problem in Figure 7.c. This in turn ensures LLC misses always complete quicker than counterless like in Figure 7.b.

We encode a data block's encryption mode and counter value into the block as one unified word that we call $EncryptionMetadata$. ***In epochs that use counter mode for LLC writebacks***, the $EncryptionMetadata$ to encode into each written block is the block's counter value after it has been verified via the integrity tree; the maximum allowed counter value is 2^n 2 when choosing an *EncryptionMetadata* size of n bits. ***In epochs that use counterless encryption for LLC writebacks***, the $EncryptionMetadata$ to encode into each written block is the maximum possible word value of 2^n 1; this serves as a flag value to indicate the encryption mode is counterless instead of counter mode.

We only encode $EncryptionMetadata$ into data blocks, but not counter blocks and integrity tree blocks. These blocks are only accessed by Counter-light Encryption for writebacks; writebacks are not performance-critical.

To encode an $EncryptionMetadata$ into each data block, we note prior works [64] [29] [68] encode into the ECC of a block some extra information that are not related to reliability. Encoding the extra information into ECC does not increase the size of the block as the information is not physically stored in any dedicated space in the block. On LLC read miss, the extra information can be decoded from the block's ECC without physically fetching it from memory.

However, none of the prior works can be directly reused because the ECC used in server memory today is different from the ECC used in prior works. Prior works are designed for single-bit error correction (a.k.a, SECDED), which is used in GPUs and older memory systems; modern servers, however, commonly use chipkill-correct ECC to protect against single-chip failure per rank [9] [34] [30] [21] [26] [27] [79] [15] [42] [47] [40] [41] [72] [78].

As such, we explore how to encode $EncryptionMetadata$ into chipkill-correct ECC. To the best of knowledge, our paper is the first work to *encode* into chipkill-correct ECC extra

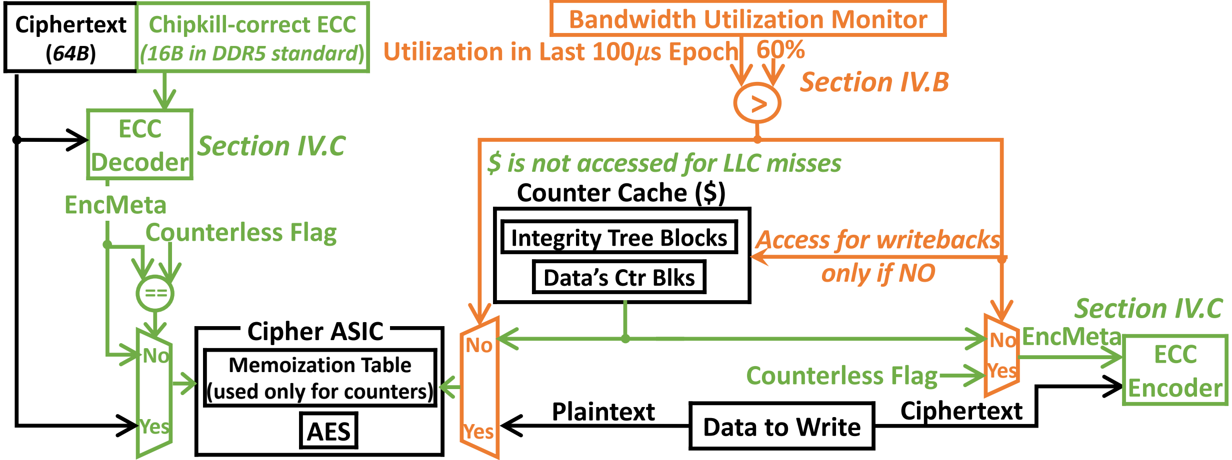


Fig. 11: Architecture overview of the full design of Counter-light Encryption. Green and orange highlight the key differences from prior works. Orange especially highlights the dynamic switching of encryption mode discussed in Section IV-B. 'EncMeta' means 'EncryptionMetadata'.

information that are unrelated to reliability, without needing to spend dedicated physical bits per block to store them.

The $EncryptionMetadata$ should ideally be encoded not just into any chipkill-correct, but one specialized for encrypted memory. We encode $EncryptionMetadata$ into a prior ECC design widely used in many prior works in memory encryption - Synergy [63] (see Figure 3).

Figure 11 provides an architecture overview of the full design of Counter-light Encryption.

LLC Writeback: Counter-light Encryption encodes each block's $EncryptionMetadata$ into the block's parity by calculating $parity = EncryptionMetadata \quad D1 \quad D2... \quad D8 \quad MAC$, where Di is the block's ciphertext data in chip i (see Figure 12). Later, when the block is read, this new parity allows the block's $EncryptionMetadata$ to be quickly decoded as $parity \quad D1 \quad D2... \quad D8 \quad MAC$, incurring the delay of $log_2(9) = 4$ XOR gates.

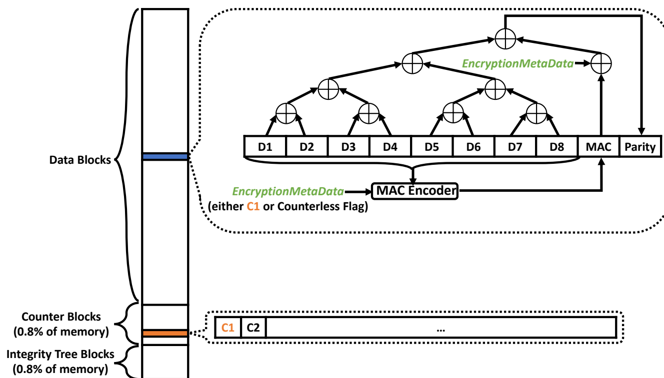


Fig. 12: Full memory layout under Counter-light Encryption. Each counter (i.e., C1, C2...) in the counter block ***logically*** encodes a 4B counter value. ***C1*** protects the expanded data block shown in the figure. Each Di ($1 \leq i \leq 8$), MAC, and Parity in the data block is 8B. Unlike Synergy in Figure 3, Counter-light Encryption ***also*** uses a data block's *EncryptionMetadata* as an input when calculating the block's parity and MAC.

Since each parity is 8B, the $EncryptionMetadata$ can also be up to 8B; but we make it 4B, to leave 4B to encode other extra information (e.g., locks for spatial safety [68]) to avoid sacrificing other extra information proposed by prior works. When a block's counter value increases beyond the maximum word value of $2^{32} \quad 2$ and reaches the counterless flag of $2^{32} \quad 1$, the block naturally switches to counterless encryption permanently until the next system reboot. In a 64GB memory channel that does nothing but continuously write at the full DDR5 transfer rate of 6400MT/s for eight years, at most 1.4% of blocks will permanently switch to counterless mode and cease to benefit from Counter-light Encryption.

The MAC in each block also takes the block's $EncryptionMetadata$ as an input (see Figure 12) so that the MAC can later detect during LLC read misses whether the decoded $EncryptionMetadata$ may be erroneous. Under counter mode, the OTP for calculating the block's MAC takes by default a counter value as an input (see Section II-B); the $EncryptionMetadata$ for counter mode is the same as the counter value. Same as Synergy, the MAC has 64 bits. Under counterless mode, we add the $EncryptionMetadata$ as an input to the SHA-3 used for the counterless MAC (see Section II-A); to keep hardware regular, we keep the MAC 64 bits, instead of using a smaller (e.g., 28-bit) MAC [37] for the counterless mode.

LLC Miss: After a data block arrives, Counter-light Encryption decodes the block's $EncryptionMetadata$ from the block's parity, and uses it to decrypt the block; the correctness of data and parity-decoded $EncryptionMetadata$ are checked by using them to recalculate the block's MAC to compare against the fetched MAC.

The common case is that fetched data block has no hardware error and is not tampered with. In this case, the two MACs always match and the check passes. Figure 13 illustrates the steps for this common case. If the data block uses counter mode, Counter-light Encryption saves the AES latency from the critical path of LLC miss; else, the LLC miss latency is the same as today's counterless encryption.

Error Correction: If the fetched data block is bad or

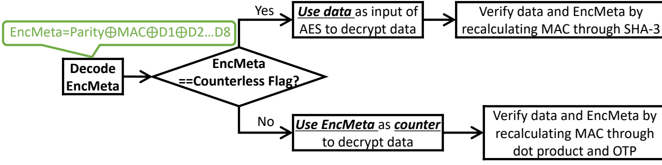


Fig. 13: A fault-free LLC read miss under Counter-light Encryption.

tampered with, the check fails. Reusing the error correction procedure in Section II-C requires the original parity in Figure 3 (i.e., the parity without $EncryptionMetadata$ xored into it). The original parity can be obtained by xoring $EncryptionMetadata$ with the parity fetched with the block; xoring the same value twice into another value cancels it out from the other value.

But when a block is erroneous, so can its $EncryptionMetadata$ decoded from the parity. The correct $EncryptionMetadata$ can be either the flag for counterless encryption or the counter value recorded in the block's counter block. As such, Counter-light Encryption separately assumes the two possible $EncryptionMetadata$ values and attempts correction under each assumption (see Figure 14). The correction under the wrong assumption will use the wrong method to recalculate MACs (i.e., use SHA-3 when OTP should be used or vice versa) and mismatch the fetched MAC or parity-corrected MAC (see Section II-C). But the correction assuming the correct $EncryptionMetadata$ will succeed ①② (i.e., have MAC match) if only one chip has error. When multiple chips have problems, all correction can fail ③ as chipkill-correct is primarily designed to correct errors only in one chip; this leads to a detectable uncorrectable error (DUE) and renders the block's value unusable.

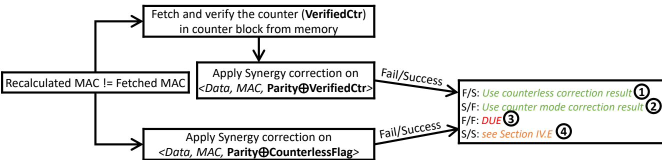


Fig. 14: Error correction under Counter-light Encryption. MAC uses SHA-3 under counterless and uses AES and dot product under counter mode.

D. Implementation Details and Overheads

Latency: Compared to no memory encryption, LLC misses under Counter-light Encryption are only 0.75ns slower in the common case when the missing block uses counter mode and its counter value hits in the memoization table. After ***half*** of the data block arrives (i.e., ***1.25ns before*** the ***entire*** block arrives), the MC has the 4B of parity needed to decode the 4B counter value (i.e., the $EncryptionMetadata$). The total latency to fetch memoized AES result for the counter value and then combine it with the address-only AES to generate the OTP takes 2ns (see Figure 4). So the OTP is available $2ns \quad 1.25ns = 0.75ns$ after the whole data block arrives and, thus, the 0.75ns latency overhead compared to no encryption.

While MAC verification/error detection also requires calculating the dot product using the arrived data and secret keys and xoring it with the OTP (see Section II-B), dot product can start without waiting for the OTP. Also note that in a system ***without*** encryption, using standard ECC instead of the MAC to detect error takes $1ns$. As such, as long as the dot product takes $< 0.75ns + 1ns = 1.75ns$, it will not cause any additional latency overhead over no encryption beyond the $0.75ns$ latency overhead to wait for OTP generation; note that the eight products summed together to produce the dot product can be calculated in parallel.

Memory Overhead: Counter-light Encryption needs an integrity tree to preserve the confidentiality of data blocks with counter mode encryption (see Section II-B). A single tree is enough considering memory encryption today - either total/system memory encryption or per-VM encryption - has no integrity tree. Counter-light Encryption is compatible with recent Split Counters design [62], which only uses 1.6% of memory to store both the counter blocks and tree nodes; each counter value to encode into a data block is a full counter value (i.e., the sum of a major and a minor counter, but the minor counter value by itself).

CPU Area Overhead: Counter-light Encryption inherits from RMCC a 4KB memoization table that records the AES results calculated from a ***single global key***. Only the few blocks using ***counterless*** mode have a need for ***per-VM keys***. If the same key were used for all VMs, ***counterless*** encryption would produce the same ciphertext for the same data stored at the same block, which enables the ciphertext side-channel attack [22]. In this attack, an attacker knowing the plaintext and ciphertext of a block in the attacker's VM can precisely infer the plaintext of a later VM that reuses the block and writes the same ciphertext to the block. ***Counter mode*** encryption, however, can use the same key for all VMs because different VMs writing the same value to the same block always write different ciphertexts as the counter is different each time. Like existing memory encryption [35] [7], all the encryption keys are maintained in hardware and completely hidden from software (e.g., host OS, guest OS, etc).

Summary of Counter Block Accesses: Counter-light Encryption accessing the counters and integrity tree nodes when writing back data. Counter-light Encryption also accesses them in the rare case that a block is erroneous and requires error correction (see Figure 14). To minimize the traffic overhead to access counters and integrity nodes, we use a 64KB counter cache in Section VI. Because Counter-light Encryption does not access any counter blocks on LLC misses, Counter-light Encryption does not cache counters during LLC misses.

E. Reliability

The error correction procedure in Synergy [63] (see detail in Section II-C) has a small $2\ ^{61}$ probability of reporting correction fails (e.g., a detected uncorrectable error or DUE) when ***only one*** chip is bad; in the rare case that two trials experience MAC matches (i.e., one for the right correction assuming the right bad chip and one for a wrong miscorrection

assuming the wrong bad chip), Synergy cannot tell which one is wrong. Counter-light Encryption doubles the total number of trials due to also guessing two possible values of $EncryptionMetadata$; as such, Counter-light Encryption can double the DUE rate to $2\ ^{60}$. But $2\ ^{60}$ is still small.

If needed, Counter-light Encryption can be enhanced to only ***marginally*** increase the original DUE probability of $2\ ^{61}$, instead of ***doubling*** it. We note that wrongly decrypting encrypted data (e.g., due to using the wrong $EncryptionMetadata$) is the same as re-encrypting the already encrypted data; in general, ciphertext is highly random (i.e., has high entropy). As such, we expect wrongly decrypted data to be characterized by high entropy. We evaluate the entropy of all benchmarks used in Section IV-A and find $\ 99.9\%$ of LLC read misses for each benchmark have a entropy of $\quad 5.5$ for wrongly decrypted data out of the theoretical maximum entropy 6, while all original plaintexts have a entropy < 5.5. As long the decrypted data under only one of the two encryption modes have an entropy of $\ 5.5$, the MC can decide this mode is incorrect and that the other encryption mode is correct. In this way, Counter-light Encryption only adds $2\ ^{61}\ (1\quad 99.9\%) = 2\ ^{61}\ \ 0.001$ to the original DUE probability of $2\ ^{61}$.

Because error correction requires reading the counter block, Counter-light Encryption gets no benefit from encoding counter value and encryption mode into bad blocks with permanent hardware faults. Existing server CPUs can detect permanent faults in a bank or rank to activate the rank-level spare bits to remap data out of the faulty chip [57]. Counter-light Encryption may permanently switch back to counterless encryption an entire bank or rank that is diagnosed with permanent fault. For such a faulty bank or rank, Counter-light Encryption always uses only counterless encryption for LLC writebacks, misses, and error correction.

F. Security

If an attacker tampers with a block's parity to cause the parity to decode the wrong $EncryptionMetadata$, the block's MAC can securely detect a wrong $EncryptionMetadata$. The possibility of wrong $EncryptionMetadata$ passing the MAC verification is $2\ ^{64}$ because the MAC has 64 bits. In the rare event the wrong $EncryptionMetadata$ passes the MAC verification, decrypting data using wrong $EncryptionMetadata$ produces garbage plaintext, without harming confidentiality. While an attacker can always replay the whole data block to pass the integrity check, there is no loss on integrity compared to counterless encryption, which also does not detect physical replay attacks.

The prior memoization work [74] combines counter-only AES and address-only AES using carry-less multiplication, which is a linear operation; but linearity is undesirable in security. A naive approach to make the combining logic of counter-only AES and address-only non-linear is to use a lightweight cipher. But this is an overkill because a main design target of ciphers is to protect against known-plaintext attack [20], where an attacker extracts the secret key by selecting plaintexts and examining their ciphertext outputs to analyze correlations; the input to the combining logic – the address-only AES and counter-only AES – are unknown to attackers because they are both AES outputs, not plaintext an attacker can choose or even observe.

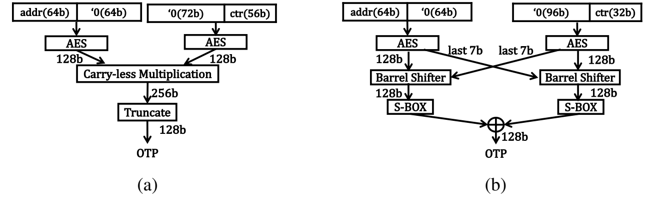


Fig. 15: (a) RMCC combines the address-only AES result and counter-only AES result using carry-less multiplication and truncation to generate the OTP; these are weak linear operations. (b) Counter-light Encryption combines address-only AES result and counter-only AES result using barrel shifting for diffusion and using nonlinear S-Box transformation for confusion.

Although counter-only AES and address-dependent AES are unknown to attackers, a plausible attack is to solve for these unknowns by setting up algebraic equations (i.e., algebraic attack [19] [18] [50]); if an attacker can solve these unknowns, he/she can then combine them to calculate many OTP values and use them to decrypt other data blocks. Explicitly, an attacker can set up a system of bit-level (i.e., boolean algebraic) equations based on observing multiple OTPs for multiple data blocks sharing multiple counters, where each equation is translated from the combinational circuit to calculate ***one*** OTP ***bit***; each equation uses each bit in the counter-only AES and address-only AES as a boolean variable and uses each bit in the observed OTPs as a constant. When α memory blocks share c different counter values, the OTPs of the α blocks are calculated from α address-dependent AES results and c counter-dependent AES results. Because each AES result is 128 bits and each bit in the AES results is an unknown variable, the total number of unknowns n is:

$$n = 128(\alpha + c). \tag{1}$$

Because the α memory blocks sharing c different counter values can have $\alpha\ \ c$ 128-bit OTPs, the total number of boolean quations m is:

$$m = 128\alpha\quad c. \tag{2}$$

A system of equation is solvable when the number of equations is equal to the number of unknowns. The simplest solvable case occurs when $\alpha = 2$ and $c = 2$; here, $m = 512$ and $n = 512$ according to Equation (1) and (2).

A system of equations that is **theoretically**-solvable, however, can still be **unsolvable in practice** if they are ***too complex***, especially when the equations are non-linear. For example, even for a pair of known plaintext and ciphertext under AES, attackers can set up a system of 128 equations with 128 unknowns (i.e., the 128-bit key) but cannot practically solve the equations. In comparison, the minimal solvable system of equations under our proposed combining logic has 512 unknowns; these equations are also non-linear.

To check whether the equations are indeed too complex to solve, we attempt to solve them using a SAT solver (Minisat [65]). Even with the simplest case (i.e., $\alpha = 2$ and $c = 2$), which is intuitively the easiest to solve, the SAT solver fails to find a solution after more than two months. The solver's progress estimate ceases to increase after the first day. After exhausting heuristical algorithms, the SAT solver falls back on the brute-force approach that is $O(2^n)$ with the number of variables; here, $n > 512$ because transforming the initial 512 equations to the CNF form required by the SAT solver introduces many new intermediate variables.

We also examine a more advanced algebraic attack that transforms the equations to Multi-Quadratic (MQ) form [71] [18] [19], instead of CNF; a system of m ***independent*** MQ equations with n variables can be solved in polynomial time if $m \quad n(n \quad 1)/2$ [71]. As such, checking whether the equations transformed into MQ form can be solved in polynomial time requires comparing m and n.

We transform the system of original equations obtained from A * C One-Time Pads (OTPs) to m MQ equations, where

$$m = 760\alpha \quad c + 160(\alpha + c). \tag{3}$$

Transforming the original equations to MQ form adds many intermediate variables in addition to the original $128(\alpha + c)$ bits of AES results. As such, Equation (1) can be extended to:

$$n \quad 128(\alpha + c). \tag{4}$$

By using Equation (3) and Equation (4), we can compare m and n and conclude that $m < n(n \quad 1)/2$ (i.e., the equations are NP hard, not polynomial in complexity).

While our setup of MQ equations may be simplified, doing so will only reduce m, without affecting the conclusion that $m < n(n \quad 1)/2$. Meanwhile, adding more redundant equations to the system (e.g., adding multiple transformations of the same equations) is useless just like how doing so is useless under regular algebra.

V. Methodology

Using Gem5 [11], we evaluate Counter-light Encryption using the same workloads as Section III. These benchmarks include IBM graphBIG [59], which uses IBM's System G framework, a set of industrial graph computing toolkits used by many commercial clients. We run GraphBig as four threads using Facebook-like dataset [1] as input. We simulate graphBig executing under multi-threading. We also select from SPEC2017 [14] and PARSEC [10] another four benchmarks - canneal, streamcluster, omnetpp and mcf, which are used by recent prior works on speeding up irregular workloads [55] [61]. We evaluate the SPEC/PARSEC benchmarks through running multi-programmed workloads; each workload contains four instances of the same benchmark.

Using Gem5's KVM mode, we fast forward each benchmark into its region of interest. Then we use atomic simulation to warm up the integrity tree and memoization table for 25 billion instructions. After warming up in atomic mode, we run the workload for 20ms in Gem5's atomic mode and 20ms in detailed mode to warm up caches and branch predictor. Lastly, we measure performance within a 20ms observation window in Gem5's detailed mode.

CPU	4 OoO cores, 3.2 GHz
Prefetchers	Next-line: L1$, L2$ Stride: L1$(degree 1), L2$(degree 2)
L1d$/L1i$/L2$/L3$	32KB/64KB/1MB/8MB; 2/2/4/17ns
Counter$/Memoization Table	64KB 32-way, 4KB
AES-128/AES-256/sha3	10ns/14ns/1ns
Memory	128 GB, 25.6 GB/s
tCL/tRCD/tRP	13.75ns/13.75ns/13.75ns
Channels/Ranks	1/8
Bandwidth Utilization Threshold	60%

TABLE I: System Configuration.

Table I shows the system setting of performance evaluation in Gem5. We use the 64KB counter cache only for LLC writebacks and an 128-entry memoization table for counter-mode encryption. We use Ramulator [49] and DRAMPower [16] to model 128GB of DRAM in Gem5.

VI. Results

Figure 16 shows the performance of Counter-light Encryption and counterless encryption normalized to without memory encryption. Counter-light Encryption incurs only $\quad 2\%$ slowdown, on average, normalized to without encryption. Only omnetpp suffers up to 5% slowdown; omnetpp incurs high bandwidth overhead (e.g., 96%, see Figure 18).

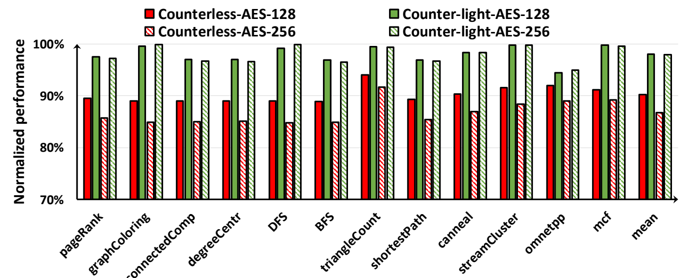


Fig. 16: Performance of Counter-light Encryption and counterless encryption normalized to without encryption, under AES-128 and AES-256.

Compared to counterless encryption, Counter-light Encryption improves performance by 11%, on average across AES-128 and AES-256. The performance benefit increases with higher AES latency. Figure 16 shows the average performance improvement increases from 8.6% to 13.0% when increasing AES latency from 10ns to 14ns (i.e., when going from AES-128 to AES-256). Counter-light Encryption is tolerant to higher AES latency because it encrypts most blocks using counter mode, which can reuse memoized AES results, instead of calculating them from scratch, after the requested data arrives. Counterless encryption, in comparison, always calculate AES from scratch after data arrive for every LLC miss. The improvement comes from reducing LLC miss latency. Figure 17 shows the average LLC miss latency overhead of counterless encryption and Counter-light Encryption, compared to without encryption. Under AES-128, Counter-light Encryption saves, on average, 7.2ns on LLC data miss latency compared to

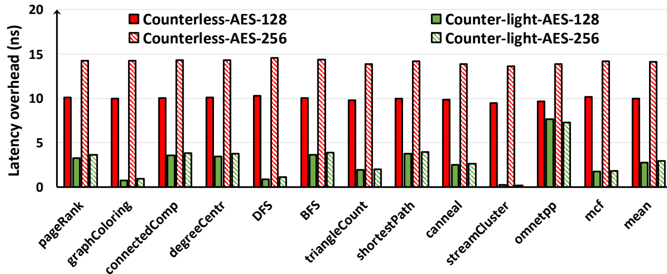


Fig. 17: Average LLC miss latency overhead of counterless encryption and Counter-light Encryption compared to no encryption.

counterless encryption. The saving increases to 11.2ns under AES-256.

Bandwidth Utilization: While Counter-light Encryption incurs a memory write bandwidth overhead over counterless encryption, the impact of this overhead on performance is small because there is often bandwidth to spare. As Figure 18 shows, the average bandwidth utilization is 22% under 25.6GB/s DRAM bandwidth when encryption is turned off; while Counter-light Encryption increases the average bandwidth utilization to 36%, there is still a lot of spare bandwidth.

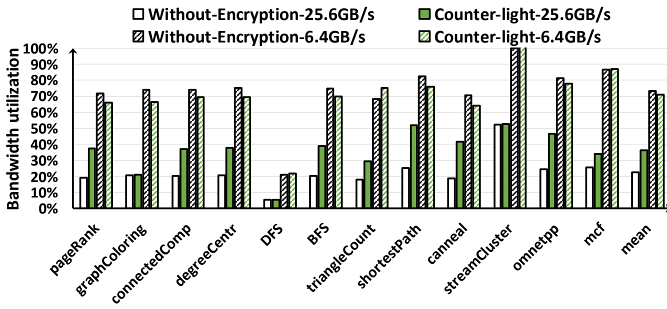


Fig. 18: DRAM bandwidth utilization of Counter-light Encryption and counterless encryption under DRAM with 25.6GB/s and 6.4GB/s bandwidth.

Energy and Power: Counter-light Encryption provides DRAM energy savings of 5.1% per instruction compared to counterless encryption (see Figure 19). The saving comes from improving performance and, thus, reducing idle DRAM energy; idle power dominates in the large memory systems typical in server systems. Only omnetpp has higher energy due to receiving relatively small performance benefit.

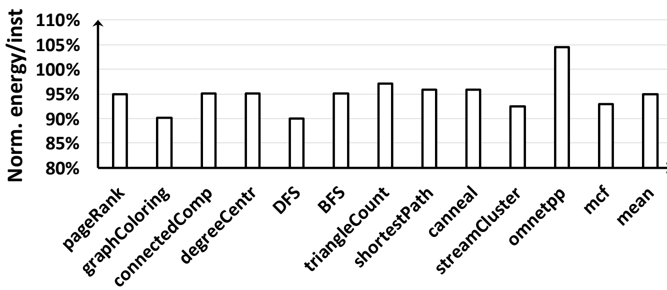


Fig. 19: Energy per instruction of Counter-light Encryption under AES-128, normalized to counterless encryption.

Sensitivity to Bandwidth Utilization: To explore how Counter-light Encryption works under high DRAM bandwidth utilization, we also perform stress test for DRAM with only 6.4GB/s bandwidth, similar with peak bandwidth of DDR2. Under 6.4GB/s bandwidth, the bandwidth utilization increases significantly to 73% (see Figure 18). The performance degradation of Counter-light Encryption in the worst case is 1.4% of counterless encryption (see Figure 20). Counter-light Encryption dynamically reverts the data blocks to counterless encryption when the bandwidth utilization is high and thus has nearly the same performance as counterless encryption for most benchmarks.

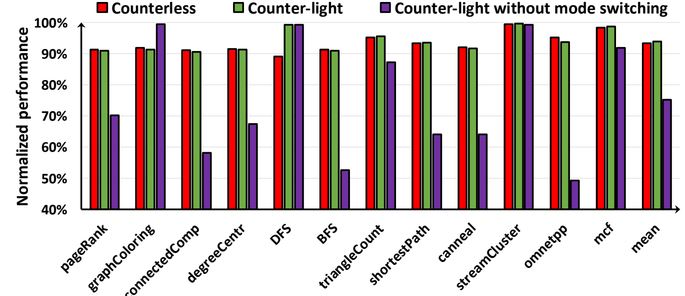


Fig. 20: Performance under a ***low*** 6.4GB/s DRAM bandwidth; everything is normalized to without encryption.

As sensitivity analysis, we also evaluate what would happen if Counter-light Encryption never switches any block back to counterless mode. Without dynamic mode switching, the average performance degradation over counterless encryption is 20%. Omnetpp suffers from the most slowdown (i.e., 51%) because the overhead bandwidth normalized to without encryption under counter mode is high (i.e., 96%, see Figure 18). However, benchmarks with very few LLC writebacks (e.g., the number of LLC writebacks is 1% of the number of LLC misses for streamcluster) only have slight traffic overhead in the absence of dynamic mode switching and suffer little degradation. GraphColoring actually performs better. Its traffic overhead under counter-mode encryption is very low - 3%; thus, the benefit of faster cipher calculation due to ***always*** using counter mode encryption outweighs the small performance loss caused by the small bandwidth overhead.

Sensitivity to Bandwidth Utilization Threshold: We evaluate different bandwidth thresholds – 10%, 60%, and 80% – for switching encryption mode. Figure 21 shows that under the low 6.4GB/s DRAM bandwidth, as the threshold increases from 10% to 80%, the number of LLC writebacks using counterless encryption reduces from 100% down to 70%. Under our default threshold 60%, 91% of all LLC writebacks use counterless encryption; ***but under the regular 25.6GB/s DRAM bandwidth, only 3% of LLC writebacks use counterless encryption***.

Figure 22 shows performance under the different utilization thresholds assuming the 6.4GB/s DRAM bandwidth.

Sensitivity to Workload Type: To explore how well Counter-light Encryption works under different workload types, we also evaluate workloads with regular access patterns

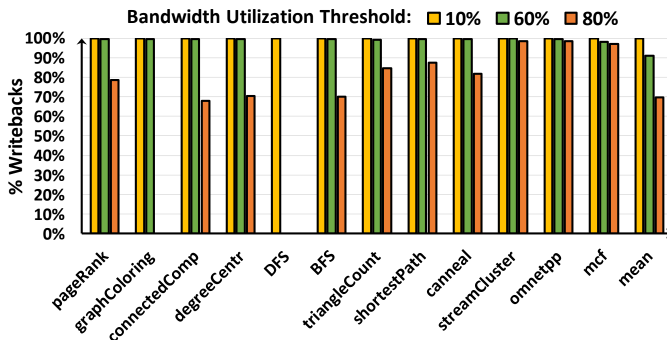


Fig. 21: LLC writebacks using counterless encryption normalized to total LLC writebacks under a ***low*** 6.4GB/s bandwidth.

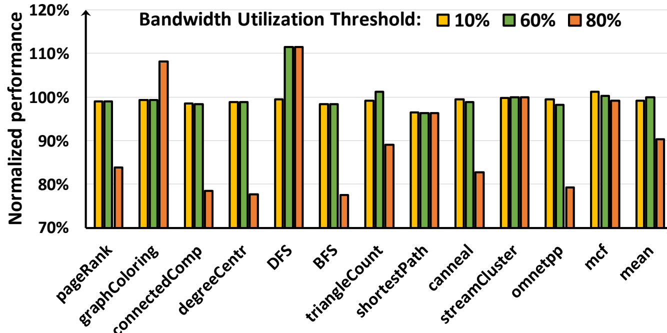


Fig. 22: Performance of using different bandwidth utilization thresholds under the ***low*** 6.4GB/s bandwidth, normalized to counterless encryption.

from SPEC2017 [14]. The average performance of Counter-light normalized to without encryption is 99.5%; for counterless, it reduces to 96.6% (see Figure 23). It is well-known that all memory encryption modes (including counter mode) work well for workloads with regular access patterns. Same as most memory encryption designs, Counter-light Encryption performs well for workloads with regular access patterns. We

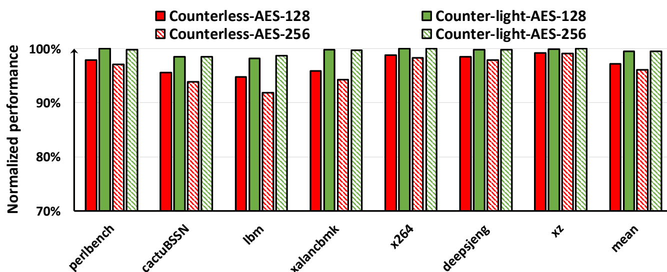


Fig. 23: Performance of Counter-light Encryption and counterless encryption under 25.6GB/s bandwidth, normalized to without encryption. The results are for other workloads with more regular memory access patterns.

also evaluate these workloads using quarter DRAM bandwidth, where bandwidth overheads are costly. Under Counter-light Encryption, these workloads still retain 99.5% of their performance as counterless encryption.

VII. Related Work on Storing Additional Information in Each Data Block

Other prior works have explored how to store additional information in each data block.

Prior works [60] [48] find that $> 90\%$ of data blocks can be compressed slightly to free up 4B to 8B per block; they propose using the compression-freed space to store additional values (e.g., additional ECC). However, this approach is not applicable to encrypted memory. Slightly-compressed blocks with a compressed plaintext size of 60B (i.e., 64B - 4B) or 56B (i.e., 64B - 8B) still produce 64B ciphertext because AES always outputs a multiple of 16B. An alternative is compressing $EncryptionMetadata$ and data together before encrypting the block. However, encrypting $EncryptionMetadata$ by itself will cause cyclic dependency during decryption; thus, this alternative will not work.

Encoding $EncryptionMetadata$ into Chipkill-correct ECC cannot simply reuse prior methods of encoding into SECDED some extra information unrelated to reliability. While an older work NIM6133 [64] extracts the encoded extra information at the end of ***full*** SECDED error correction, full error correction is slower under chipkill-correct ECC, especially Synergy, which requires many trials and errors (see Section II-C). Furthermore, NIM6133 can only encode four extra bits, which is not enough to encode $EncryptionMetadata$. While a recent work [68] can encode 2B of extra information in each 32B GPU memory block, the 'enhanced' code only serves like a checksum to ***detect*** whether an equivalent 2B information from the CPU (e.g., a spatial safety key) is correct; the code cannot recompute on its own the full value of the extra information encoded into it. Counter-light Encryption, however, must precisely reconstruct the full value of $EncryptionMetadata$.

VIII. Conclusion

Mainstream server memory has moved from earlier counter mode encryption to counterless encryption to boost memory performance. But cipher calculations in counterless encryption slow down every LLC read miss. We note counter mode encryption may complete the calculations on or prior to data arrival. As such, we propose Counter-light Encryption to achieve the best of both worlds in performance across counterless and counter mode encryption while meeting the security of counterless encryption. Our evaluation shows irregular workloads achieve 98% of their average performance over no encryption, up from the 88% normalized performance under counterless encryption. When memory bandwidth utilization is high, the performance degradation of Counter-light Encryption over counterless encryption is only 1.4% in the worst case.

Acknowledgment

We thank the National Science Foundation (NSF) for its generous support through grant 1850025. We thank Dr. Atul Mantri for his insightful feedback. 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.

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.

References

[1] "Facebook dataset." [Online]. Available: https://atlarge.ewi.tudelft.nl/graphalytics/zip/datagen-8_5-fb.zip

[2] "Intel® xeon® silver 4314 processor." [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/215269/intel-xeon-silver-4314-processor-24m-cache-2-40-ghz/specifications.html

[3] Amazon, "Amd sev-snp." [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html

[4] AMD, "4th gen amd epyc™ processor architecture." [Online]. Available: https://www.amd.com/system/files/documents/4th-gen-epyc-processor-architecture-white-paper.pdf

[5] AMD, "Sev secure nested paging firmware abi specification." [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56860.pdf

[6] AMD, "Secure encrypted virtualization api version 0.24," 2020. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/55766_SEV-KM_API_Specification.pdf

[7] AMD, "Strengthening vm isolation with integrity protection and more," 2020. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

[8] 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.

[9] M. V. Beigi, Y. Cao, S. Gurumurthi, C. Recchia, A. Walton, and V. Sridharan, "A systematic study of ddr4 dram faults in the field," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 991–1002.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[11] 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.

[12] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems-CHES 2007: 9th International Workshop, Vienna, Austria, September 10-13, 2007. Proceedings 9*. Springer, 2007, pp. 450–466.

[13] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger *et al.*, "Prince–a low-latency block cipher for pervasive computing applications," in *Advances in Cryptology–ASIACRYPT 2012: 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings 18*. Springer, 2012, pp. 208–225.

[14] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[15] S. Cha, O. Seongil, H. Shin, S. Hwang, K. Park, S. J. Jang, J. S. Choi, G. Y. Jin, Y. H. Son, H. Cho *et al.*, "Defect analysis and cost-effective resilience architecture for future dram devices," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 61–72.

[16] K. Chandrasekar, C. Weis, Y. Li, B. Akesson, N. Wehn, and K. Goossens, "Drampower: Open-source dram power & energy estimation tool (2012)," *URL: http://www. drampower. info (visited on 11/14/2017)*, 2017.

[17] P.-C. Cheng, W. Ozga, E. Valdez, S. Ahmed, Z. Gu, H. Jamjoom, H. Franke, and J. Bottomley, "Intel tdx demystified: A top-down approach," *arXiv preprint arXiv:2303.15540*, 2023.

[18] N. Courtois, A. Klimov, J. Patarin, and A. Shamir, "Efficient algorithms for solving overdefined systems of multivariate polynomial equations," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2000, pp. 392–407.

[19] N. T. Courtois and J. Pieprzyk, "Cryptanalysis of block ciphers with overdefined systems of equations," in *Advances in Cryptology—ASIACRYPT 2002: 8th International Conference on the Theory and Application of Cryptology and Information Security Queenstown, New Zealand, December 1–5, 2002 Proceedings 8*. Springer, 2002, pp. 267–287.

[20] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999. [Online]. Available: https://www.cs.miami.edu/home/burt/learning/Csc688.012/rijndael/rijndael_doc_V2.pdf

[21] T. J. Dell, "A white paper on the benefits of chipkill-correct ecc for pc server main memory," *IBM Microelectronics division*, vol. 11, no. 1-23, pp. 5–7, 1997.

[22] Z.-H. Du, Z. Ying, Z. Ma, Y. Mai, P. Wang, J. Liu, and J. Fang, "Secure encrypted virtualization is unsecure," *arXiv preprint arXiv:1712.05090*, 2017.

[23] M. El-Hindi, T. Ziegler, M. Heinrich, A. Lutsch, Z. Zhao, and C. Binnig, "Benchmarking the second generation of intel sgx hardware," in *Proceedings of the 18th International Workshop on Data Management on New Hardware*, 2022, pp. 1–8.

[24] A. Fakhrzadehgan, Y. N. Patt, P. J. Nair, and M. K. Qureshi, "Safeguard: Reducing the security risk from row-hammer via low-cost integrity protection," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 373–386.

[25] 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.

[26] E. Fujiwara and D. K. Pradhan, "Error-control coding in computers," *Computer*, vol. 23, no. 7, pp. 63–72, 1990.

[27] S.-L. Gong, J. Kim, S. Lym, M. Sullivan, H. David, and M. Erez, "Duo: Exposing on-chip redundancy to rank-level ecc for high reliability," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 683–695.

[28] S. Gueron, "A memory encryption engine suitable for general purpose processors." 2016. [Online]. Available: https://eprint.iacr.org/2016/204.pdf

[29] R. H. Gumpertz, "Combining tags with error codes," *ACM SIGARCH Computer Architecture News*, vol. 11, no. 3, pp. 160–165, 1983.

[30] S. Gurumurthi, "Advanced memory device correction (amdc) for servers," *AMD Whitepaper*, 2020.

[31] S. Hong, P. J. Nair, B. Abali, A. Buyuktosunoglu, K.-H. Kim, and M. Healy, "Attaché: Towards ideal memory compression by mitigating metadata bandwidth overheads," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 326–338.

[32] J. Huang and Y. Hua, "A write-friendly and fast-recovery scheme for security metadata in non-volatile memories," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 359–370.

[33] R. Huang and G. E. Suh, "Ivec: off-chip memory integrity protection for both security and reliability," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 395–406, 2010.

[34] IBM, "Introduction to ibm® power® reliability, availability, and serviceability for power10 processor-based systems using ibm powervm™." [Online]. Available: https://www.ibm.com/downloads/cas/2RJYYJML

[35] Intel, "Intel® architecture memory encryption technologies." [Online]. Available: https://cdrdv2-public.intel.com/679154/multi-key-total-memory-encryption-spec-1.4.pdf

[36] Intel, "Intel® total memory encryption white paper." [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/vpro/hardware-shield/total-memory-encrpytion.html

[37] Intel, "Intel® trust domain extensions." [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/690419

[38] Intel, "Using the rdtsc instruction for performance monitoring." [Online]. Available: https://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf

[39] J. Jean, I. Nikolić, T. Peyrin, L. Wang, and S. Wu, "Security analysis of prince," in *International Workshop on Fast Software Encryption*. Springer, 2013, pp. 92–111.

[40] X. Jian, H. Duwe, J. Sartori, V. Sridharan, and R. Kumar, "Low-power, low-storage-overhead chipkill correct via multi-line error correction," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[41] X. Jian and R. Kumar, "Adaptive reliability chipkill correct (arcc)," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013, pp. 270–281.

[42] X. Jian, V. Sridharan, and R. Kumar, "Parity helix: Efficient protection for single-dimensional faults in multi-dimensional memory systems," in

2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2016, pp. 555–567.

[43] S. Johnson, R. Makaram, A. Santoni, and V. Scarlata, "Supporting intel sgx on multi-socket platforms," *Intel Corp*, 2021. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf

[44] 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.

[45] D. Kaplan, "Protecting vm register state with sev-es," *White paper*, p. 13, 2017. [Online]. Available: http://events17.linuxfoundation.org/sites/events/files/slides/AMD%20SEV-ES.pdf

[46] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," *White paper*, 2016. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf

[47] J. Kim, M. Sullivan, and M. Erez, "Bamboo ecc: Strong, safe, and flexible codes for reliable computer memory," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 101–112.

[48] J. Kim, M. Sullivan, S.-L. Gong, and M. Erez, "Frugal ecc: Efficient and versatile memory error protection through fine-grained compression," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[49] 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.

[50] A. Kipnis and A. Shamir, "Cryptanalysis of the hfe public key cryptosystem by relinearization," in *Annual International Cryptology Conference*. Springer, 1999, pp. 19–30.

[51] D. Kline, R. Melhem, and A. K. Jones, "Counter advance for reliable encryption in phase change memory," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 209–212, 2018.

[52] 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.

[53] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, "Cipherleaks: Breaking constant-time cryptography on amd sev via the ciphertext side channel," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 717–732.

[54] H. Lipmaa, P. Rogaway, and D. Wagner, "Ctr-mode encryption," in *First NIST Workshop on Modes of Operation*, vol. 39. Citeseer. MD, 2000.

[55] 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.

[56] L. Martin, "Xts: A mode of aes for encrypting hard disks," *IEEE Security & Privacy*, vol. 8, no. 3, pp. 68–69, 2010. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5470958

[57] S. Micro, "Memory ras configuration." [Online]. Available: https://www.supermicro.com/manuals/other/Memory_RAS_Configuration_User_Guide.pdf

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

[59] 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.

[60] D. J. Palframan, N. S. Kim, and M. H. Lipasti, "Cop: To compress and protect main memory," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 682–693, 2015.

[61] 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

[62] 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.

[63] 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.

[64] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain access control for distributed shared memory," in *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, 1994, pp. 297–306.

[65] N. Sorensson, "minisat." [Online]. Available: https://github.com/niklasso/minisat

[66] W. Stallings, *Cryptography and network security, 4/E*. Pearson Education India, 2006.

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

[68] M. B. Sullivan, M. T. I. Ziad, A. Jaleel, and S. W. Keckler, "Implicit memory tagging: No-overhead memory safety using alias-free tagged ecc," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[69] M. Taassori, R. Balasubramonian, S. Chhabra, A. R. Alameldeen, M. Peddireddy, R. Agarwal, and R. Stutsman, "Compact leakage-free support for integrity and reliability," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 735–748.

[70] 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.

[71] E. Thomae and C. Wolf, "Solving underdetermined systems of multivariate quadratic equations revisited," in *International workshop on public key cryptography*. Springer, 2012, pp. 156–171.

[72] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. P. Jouppi, "Lot-ecc: Localized and tiered reliability mechanisms for commodity memory systems," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 285–296, 2012.

[73] X. Wang, J. B. Kotra, and X. Jian, "Eager memory cryptography in caches," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 693–709.

[74] X. Wang, D. Talapkaliyev, M. Hicks, and X. Jian, "Self-reinforcing memoization for cryptography calculations in secure memory systems," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 678–692.

[75] X. Xin, W. Zhu, and L. Zhao, "Architecting ddr5 dram caches for non-volatile memory systems," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1057–1062.

[76] 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.

[77] 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.

[78] D. H. Yoon and M. Erez, "Virtualized and flexible ecc for main memory," in *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, 2010, pp. 397–408.

[79] D. Zhang, V. Sridharan, and X. Jian, "Exploring and optimizing chipkill-correct for persistent memory based on high-density nvrams," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 710–723.

[80] K. A. Zubair and A. Awad, "Anubis: ultra-low overhead and recovery time for secure non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 157–168.