Reducing the Overhead of Authenticated Memory Encryption Using Delta Encoding and ECC Memory

Salessawi Ferede Yitbarek and Todd Austin
University of Michigan
{salessaf,austin}@umich.edu

ABSTRACT

Data stored in an off-chip memory, such as DRAM or non-volatile main memory, can potentially be extracted or tampered by an attacker with physical access to a device. Protecting such attacks requires storing message authentication codes and counters – which incur a 22% storage overhead. In this work, we propose techniques for reducing these overheads.

We first present a scheme that leverages ECC DRAMs to reduce MAC verification & storage overheads. We replace the parity bits in standard ECC by a combination of MAC and parity bits to provide *both authentication and error correction*. This eliminates the extra MAC storage and minimizes the verification overhead as MACs can be read in parallel with data through the ECC bus. Next, we use efficient integer encodings to reduce counter storage overhead by 6× while enhancing application performance.

ACM Reference Format:
Salessawi Ferede Yitbarek and Todd Austin. 2018. Reducing the Overhead of Authenticated Memory Encryption Using Delta Encoding and ECC Memory. In *Proceedings of DAC '18: The 55th Annual Design Automation Conference 2018 (DAC '18).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3195970.3196102

1 INTRODUCTION

Data in a CPU's registers and caches is extremely hard to physically probe or modify. On the other hand, an attacker can easily probe the data bus, or dump memory contents of off-chip memory through a cold boot attack [4].

Challenges of Secure Off-Chip Storage. Secure data storage requires that the attacker cannot a) know the contents of the data (confidentiality) and b) modify data without detection (integrity). Confidentiality can be ensured by employing encryption, while integrity can be ensured by storing a message authentication code (MAC) along with the encrypted data.

These cryptographic primitives, however, result in high storage overheads (Figure 1a). Strong encryption requires having a unique, one-time (non-reusable) value referred to as a nonce. Counter-mode encryption, the most widely used mode of memory encryption, uses a growing counter value as a nonce. Counters used to encrypt each memory block need to be stored as they are necessary to decrypt the data when it is read back. SGX (Intel's implementation), for example, stores a 56-bit counter for each encrypted 64-byte block, resulting in an ~ 11% overhead [3]. Furthermore, MACs also need to be stored for each memory block. Again, SGX uses 56-bit MACs - incurring an additional ~ 11% storage overhead. Memory read latency is also impacted as reading a protected block requires fetching counter and MAC values, decryption, and integrity checking.

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196102

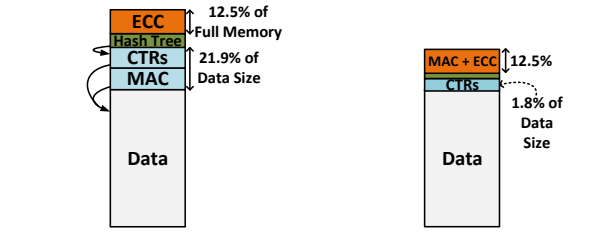


Figure 1: In this work we i) reduce counter storage overhead and ii) merge error correction with integrity checking without compromising security or forgoing error correction.

Even if attackers cannot create MAC values for arbitrary data, they can reset data, MACs, and counters to an older value – a process commonly known as a **replay attack**. To prevent such attacks, the counters and MACs need to be protected using an integrity tree – creating additional overhead. Overall, strong memory encryption incurs more than 22% storage overhead (from the 21.9% counter and MAC overhead, plus the hash tree as shown in Figure 1).

Overview of Contributions. We present the following optimizations for reducing the overheads discussed above while still improving performance:

- We show how the extra memory chips and buses available in ECC DRAM can be exploited to eliminate the encrypted memory's MAC storage overheads – without forgoing error correction or integrity checking capabilities. We describe how MAC values (augmented with only 7-bit Hamming codes) can be used for both integrity checking and error correction. Our results show that this approach has the added benefit of improving the performance of authenticated memory encryption by up to 15% as MACs are read in parallel with the data through the ECC bus.
- We also show how delta encoding can be used to represent counters with fewer bits (Section 4). Delta encoding stores the difference (deltas) between two values, instead of storing the values themselves. Since the deltas will be smaller than the actual values, they can be represented with fewer bits – resulting in lower storage overhead.

Combined together, these techniques reduce the encryption metadata storage overhead from ~ 22% to just ~ 2% without sacrificing performance.

2 BACKGROUND AND MOTIVATION

Authenticated memory encryption aims to protect a system from an attacker that has physical access to a device. The attacker can monitor memory buses, or extract data directly out of memory modules [4]. In this section, we provide an overview of the state-of-the-art in authenticated memory encryption and the associated overheads.

2.1 Authenticated Memory Encryption: An Overview

Protecting off-chip data from bus-snooping and cold boot attacks requires mechanisms that ensure **confidentiality** and **integrity** of

data. Confidentiality is guaranteed by encrypting memory blocks. Such encryption is typically performed by XOR'ing each block with a unique and random bitstream (known as a keystream) – which is generated by using a strong cipher.

Counter Mode Encryption. State-of-the-art memory encryption implementations use a block cipher (mainly AES) in counter mode. In counter mode memory encryption, a distinct counter value is associated with each 64-byte memory block. A counter value associated with a memory block needs to be incremented whenever the memory block is updated. These counters themselves are stored in the off-chip memory in plaintext.

To generate a keystream for a memory block, we encrypt the memory block's counter by using a block cipher such as AES. To make the keystream unique across different memory blocks, the the counter is concatenated with the physical address of the memory block being encrypting before being fed to the block cipher.

Counter Size. The counters need to be large enough to ensure that they do not overflow and wrap around to 0. An overflow would result in keystream reuse – which weakens the encryption. To prevent overflow, counters that are 64-bit or 56-bit wide are typically employed [3]. Using large counters, however, results in significant storage overhead. With a 56-bit counter per 64-byte memory block, the counter storage overhead will be ∼ 11%.

Data Integrity. Encryption cannot prevent an attacker from modifying or resetting data. Ensuring data integrity requires us to compute and store message authentication codes (MACs) for each memory block. A MAC is a one-way function computed over the protected data using a secret key that is securely stored on-chip.

MACs are stored for each 64-byte memory block, resulting in significant storage overheads. For example, Intel SGX computes 56-bit MACs for each memory block, incurring an ∼ 11% storage overhead (in addition to the 11% counter storage overhead).

Integrity Trees. When protecting a large chunks of memory, the MACs will require more storage than what is normally feasible on-chip. As a result, MACs need to stored in an off-chip memory.

However, we need to ensure the attacker cannot tamper the MACs. Otherwise, an attacker can "replay" old values by concurrently resetting the counter, MAC, and data to an older value.

MAC values are stored off-chip in a tamper-proof manner by using **integrity trees** [2, 3, 10]. The main aim of integrity trees is to protect the integrity of a large off-chip memory using a small amount of on-chip metadata storage.

2.2 Previously Proposed Optimizations

Execute-Only Memory (XOM) [7] and AEGIS [12] were the earliest efforts to design an architecture for tamper-proof execution. In this section, we highlight subsequent proposed optimizations that are relevant to our work [2, 10, 13].

Counter and MAC Caches. Verifying integrity by recursively reading nodes from an integrity tree requires extra memory reads. Gassend et al. integrated a dedicated cache for the integrity tree[2] to reduce the latency for reading MACs and counters. Intel's SGX implementation has a dedicated cache for MACs and counters [3].

Bonsai Merkel Trees (BMTs). The work by Rogers et al. [10] made the observation that it is possible to ensure data integrity by only protecting the integrity of counters. Since the size of counters is significantly smaller than the size of data blocks, protecting the counters (instead of the data) results in a significantly smaller tree – which the authors call a Bonsai Merkle tree. Their technique requires the counters used for encryption to also be used as an additional input when computing MAC tags. Intel SGX uses this optimized tree structure [3]. We also use Bonsai Merkel trees as the baseline and apply our proposed optimization over them.

Split Counters. Split counters [13] reduce the size of counters by storing small (typically 7 bits) minor counters (m) per memory block. However, a 7-bit counter would easily overflow, resulting in nonce re-use after just hundreds of writes to a memory block.

The authors address this issue by coupling minor counters with a 64-bit major counter (M). A single major counter is shared by multiple consecutive blocks, incurring significantly less storage overhead compared to storing a 64-bit counter for every block. The consecutive blocks that share a major counter form a **block-group** – which is typically a few kilobytes. When a block is accessed, it's minor counter is concatenated with its major counter to obtain the full counter value. When a minor counter overflows, the entire block-group is re-encrypted using a new major counter. This enables the technique to avoid re-encrypting the entire memory.

Split counters can reduce the storage overhead by 8× compared to storing a 64-bit counter for each memory block. However, this counter compaction scheme requires frequent re-encryption of block groups on memory intensive applications.

In Section 4, we present a counter encoding and storage scheme that results in significantly lower rate of re-encryption while maintaining the compactness of split counters.

Non-Volatile Main Memory Encryption. Encrypting data in an NVMM can result in faster storage media wear out [14]. Frequent re-encryption of memory blocks that result from overflowing counters will exacerbate this problem. The delta encoding scheme we present in this work will reduce potential storage media wear out that can result from more frequent re-encryptions induced by other compact counter storage schemes [13].

3 USING ECC DIMMS FOR INTEGRITY CHECKING AND ERROR CORRECTION

All servers deployed by major cloud providers today are fitted with DRAM modules that support error detection and correction (ECC). In addition, high-end desktops and laptops with ECC DRAM can be purchased on the market today. In this section, we discuss how standard ECC DIMMs can be leveraged to reduce the overhead of secure integrity checking without forgoing the error detection and correction capabilities ECC DIMMs are meant to support.

3.1 Overview

ECC-capable DRAM stores Hamming error correction codes along with data words. Current mainstream ECC implementations store 8 extra bits for every 8-byte word, resulting in a 12.5% storage overhead. These implementations enable single-bit error correction and double-bit error detection (SEC-DED) within an 8-byte word.

While regular DRAM channels have 64-bit wide data buses, DRAM modules and channels that support ECC have 72-bit wide data buses. This enables ECC bits to be read in parallel with the information bits. As a result, the memory controller is able to perform independent error checks for each 64-bit bus transaction.

We propose using the extra storage and bus reserved for ECC bits to store MACs – **without forgoing DRAM error detection and correction**. Merging integrity checking and ECC in this manner provides significant storage savings. SEC-DED ECC incurs a 12.5% storage overhead, while 56-bit MAC tags incur an additional 11% storage overhead. When both error correction and tamper-proof DRAM storage are employed, these storage overheads add up to around $1/4^{th}$ of the protected DRAM space (note that the MAC bits themselves need to be protected using ECC bits). Merging ECC and integrity checking reduces this overhead to a total of 12.5% – the storage overhead of employing ECC only.

Merging ECC and integrity checking also enables us to avoid the extra DRAM transaction required for accessing MACs, as these MAC bits will be accessed in parallel with the data block. Furthermore, as MACs stored as ECC bits will immediately be available

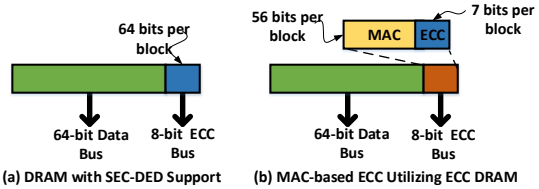


Figure 2: We enable efficient error correction and authentication by storing 56-bit MACs and 7-bit parity in the space dedicated for 64-bit parity in ECC DRAMs. The MACs are used for authentication, error detection, and correction. The extra 7-bit parity provides ECC for MACs. The extra ECC bus lines are exploited for transferring MAC + parity bits in parallel with the data.

when we read data blocks from DRAM, we do not need to separately cache the MACs – thereby freeing up on-chip tree cache space.

While implementing this scheme, however, we do *not* want to lose the error detection and correction capabilities afforded by ECC DRAM. As we will detail below, MACs can be used for powerful error detection and limited error correction. Previous work [5] has shown how hashes/MACs and full integrity trees can be used for error detection and correction. While the brute-force error correction algorithm we present below bears some similarity to [5], our main focus here is to reduce the memory footprint of authenticated memory encryption while enhancing the performance of integrity checking – while still providing error correction.

3.2 On the Security of SGX's 56-bit MAC Tags

The ECC scheme we propose here relies on the 56-bit Carter-Wegman MAC tags as introduced by Intel SGX [3]. While 56-bit MACs are typically considered short for security purposes, it has been shown that they provide sufficient security guarantees in the context of memory encryption. The analysis in [3] shows that, other serious practical obstacles aside, the rate of MAC forgery is bounded by the throughput of the hardware under attack. They estimate a successful MAC forgery would take 2 million years.

3.3 MACs for Error Detection

MACs can easily be used for *error detection* as bit flips caused by a hardware fault will cause MAC tag checks to fail. This check is part of the standard integrity checking mechanism.

On standard ECC DRAM, 64 extra bits are reserved for ECC per 64-byte memory block. Out of the 64 bits normally reserved for ECC, we use 56 bits for storing MACs. These 56 bits provide robust error detection for the data blocks – provided there are no bit-flips in the MACs themselves.

Corrupted MACs. However, we need to address one major issue before effectively using MACs for error detection. Hamming codes protect both information bits and the additional ECC bits themselves. In contrast, if a MAC check fails, we cannot determine whether it's the MAC or data bits that are corrupted.

To detect and correct bit-flips in the MACs themselves, we generate parity bits over the MAC tags using hamming codes. We will only need 7 parity bits to detect double-bit errors and correct single-bit errors in the MAC itself. Augmenting the MACs with these parity bits enables us to detect and correct errors in the MACs themselves without having to scan multiple layers of the integrity tree. This results in an efficient and simpler implementation.

With these additional bits, we have 56 MAC bits and 7 parity bits reserved, amounting to a total of 63 bits. These bits fit in the space reserved for storing 64 ECC bits for every block (Figure 2).

Enabling Efficient Scrubbing. The MAC and parity bits occupy 63 out of the 64 bits that are available for storing ECC bits. We use the remaining 1 bit for storing a single parity bit computed over the cipher text. This bit can be used by a DRAM scrubbing hardware/firmware (which typically rely on parity bits) to quickly and efficiently scan for single-bit errors without re-computing MACs. The hamming coded MACs can also be scrubbed as hamming codes contain a parity bit.

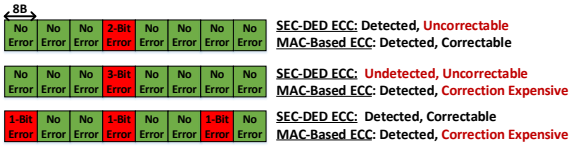


Figure 3: The error correction capability of the two schemes is highly dependent on the number and position of bit-flips.

Comparison with Standard Error Detection. Standard ECC can detect up to 2 bit-flips per 8 byte word. On a 64-byte memory block, we can detect up to 16-bit errors – provided we do not have more than 2 bit-flips per 8-byte word.

With our proposed approach, we will have 2-bit error detection on the MAC bits themselves (as we use standard SEC-DED to protect them). On the data bits, however, we have full error detection, i.e., any number of bit-flips can be detected. Figure 3 compares the two schemes under different fault types.

3.4 MACs for Error Correction

MACs, unlike Hamming codes, only detect bit flips – but not which bit(s) flipped. As a result, error correction becomes challenging.

The most straightforward way to achieve MAC-based error correction without compromising security is performing a **brute-force flip-and-check** on each of the bits. When an integrity check fails, we attempt to correct the bit error(s) by flipping each bit in the memory block one by one and re-checking the MAC value.

Cost of Error Correction. A simple flip-and-check algorithm over 64-byte (512-bit) memory blocks will require a maximum of 512 flip-and-checks to correct single-bit errors, whereas correcting double-bit errors will require a maximum of 130,816 flip-and-checks (512 combination 2). Since state-of-the-art MAC algorithms, which are essentially composed Galois field multiplications, can be computed within a single cycle in hardware [3, 13], performing double error correction within 100s of nanoseconds would be feasible.

Performance Implications. The brute-force ECC scheme will have minimal impact on performance as DRAM errors are rare occurrences. Fault analysis on Facebook's entire fleet of servers reveals that the majority of the servers affected by DRAM errors have at most 9 correctable errors per month. [8]. Note that, no additional steps on top of the existing integrity checking is required for error *detection*.

Comparison with Standard Error Correction. Standard ECC memory is able to correct single-bit errors per 8-byte words. On the other hand, the level of error correction provided by the brute-force approach we propose has the following characteristics:

(1) The level of error correction provided depends on the worst-case latency we are willing to tolerate. Correcting anything beyond double bit-flips inside a 64-byte block will require millions of cycles in the worst case.
(2) Unlike Hamming codes, we cannot provide error correction at the granularity of 8-byte words. As a bit flip in one word will affect all the bits in the MAC tag, we can only perform checks over the entire 64-byte memory.
(3) Standard error correction outperforms the flip-and-check scheme in the event where we have multiple single bit-flips spread across multiple 8-byte words. However, the flip-and-check approach can correct double bit errors with reasonable overhead even when they occur within a single 8-byte word.

In summary, the effectiveness of MAC-based error correction, as compared to traditional ECC depends on the location of the bit-flips.

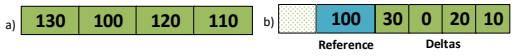


Figure 4: The integers in (a) are delta-encoded using 100 as a reference in (b). The deltas shown can be represented using 5 bits per value, whereas storing the full integers would have required 8 bits.

Figure 3 gives examples of different bit-flips and how SEC-DED and MAC-based ECC perform under those conditions.

4 REDUCING STORAGE OVERHEAD AND RE-ENCRYPTION RATE USING DELTA ENCODING

In counter mode memory encryption, we need to track a counter for each block– resulting in significant storage overheads. In this section, we propose techniques for storing these per-block counters in a more compact manner.

4.1 Overview: Delta Encoding

Delta encoding is a data representation scheme that stores the differences (deltas) between two values, instead of storing the full values themselves. This results in a more compact representation when the range of values in the data is relatively small.

Figure 4 illustrates a flavor of delta encoding, known as *frame-of-reference delta encoding*, that we will employ for compacting counters We store a **reference value** and each counter is stored as a delta to the reference value. The array [130, 100, 120,110] is stored as deltas with respect to the reference integer 100: [130-100, 100-100, 120-100, 110-100] = [30, 0, 20, 10].

Note that, in the example, we can represent the deltas using just 5 bits per value, whereas storing the full integers would have required 8 bits per value. The storage savings will be much bigger when we encode large 56-bit counter values, as discussed below.

Delta Encoding of Counters. To delta-encode counters, we group multiple memory blocks into a **block-group**. Memory blocks that are part of the same block-group are encoded using a common reference value. For example, if we group 64 blocks to form a 4KB block-group, we will store a reference value and 64 deltas.

Delta encoding significantly reduces counter storage requirements. In our design, the reference value is 56-bits, similar to the size of a counter in Intel SGX, and would never overflow during the lifetime of a machine. On the other hand, our experimental results over the PARSEC benchmark show that even a 7-bit delta values are practical for most workloads (Section 5.3).

The encoding scheme we are proposing here provides the same amount of storage savings as split counters [13]. However, as we will detail in the following sub-sections, delta encoding enables certain optimizations that could not be applied on split counters. The optimizations we present below reduce the rate of re-encryption, which in turn limits non-volatile main memory aging resulting from repeated writes, and also results in better energy efficiency.

4.2 Counter Update and Re-Encryptions

When encrypted DRAM blocks are initialized, all counters are initialized to zero. These values are represented with reference = 0 and delta = 0 (Figure 5a). When a memory block is updated, the delta value for that specific block is also incremented.

Delta Overflow & Re-encryption. When a delta overflows, we need to re-encrypt the blocks in a block group with a new counter (Figure 5a). As the counter that just overflowed is the largest counter in the group, we will use it for re-encrypting the block group. In addition, we increment the reference value to the new counter and set all the deltas to zero.

The re-encryption operation involves sequentially reading memory blocks in a block group and encrypting them using an identical counter value, i.e., the largest counter value in the block group.

Block Group and Delta Sizes. The decryption pipeline will perform better if both the reference value and the associated deltas are stored in the same memory block, as both of these values can be loaded with a single read operation. There are multiple block group and delta size combinations that would satisfy this criteria.

To test the effectiveness of our algorithms under low storage overheads, we evaluate our system using 7-bit deltas (in line with the 7-bit minor counters evaluated in [13]). With 7-bit deltas, we can fit a 56-bit reference counter and 64 delta values. Hence, we picked a 4KB (64 blocks) block group. This results in a 6x smaller storage requirement compared to storing the full 56-bit counters.

4.3 Minimizing Overflow

The major limitation with delta encoding (as presented so far) is that small deltas can overflow frequently – similar to minor counters in the split counters scheme. Delta encoding, however, provides us with opportunities for reducing the overflow rate. We present three techniques below for reducing the rate of delta overflow. In the next sub-section, we will detail how these optimizations are triggered and handled by the hardware.

Dual-Length Encoding. The delta of more frequently updated blocks will overflow faster than the less frequently updated ones. Hence, instead of assigning 7-bit deltas to all blocks in a block groups, it would be beneficial to assign more bits to deltas that are growing fast, and less bits to deltas that growing slow. Variable-length encoding is a mechanism that is used to store integers efficiently by assigning fewer bits to smaller integers.

However, decoding an array of arbitrary length integers would require numerous cycles – seriously impacting memory access latency. To mitigate this issue, we designed a constrained form of variable-length encoding – which we call dual-length encoding.

Figure 6 illustrates our dual-length encoding scheme. We group deltas in a block-group into 4 logical delta-groups (each containing 16 deltas). Each delta in this case is just 6 bits (instead of 7 bits as described above). With these slightly shorter deltas, we will have 72 unused bits in each block-group. These extra, unused bits are later used to expand the delta-group that contains an overflowing value that cannot be represented using 6 bits. Each delta is expanded with an additional 4 bits upon overflow.

In the example in Figure 6, at least one delta in group 2 is overflowing. To avoid re-encryption, we assign the reserved overflow bits to group 2. We also set the group index bits to indicate the overflow bits are assigned to group 2. If group 2 or any other group overflows after this point, we will re-encrypt the block group.

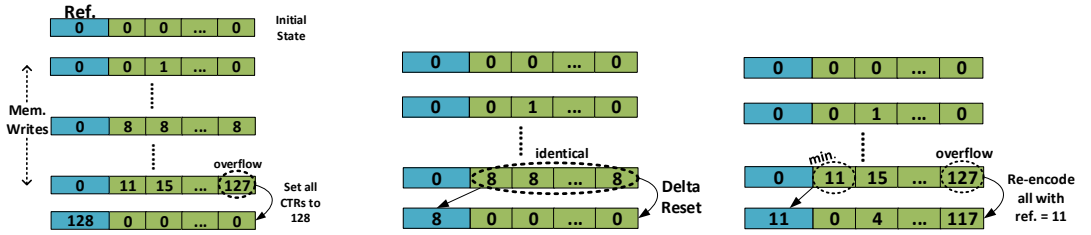
This constrained form of variable length encoding does not provide optimal storage savings, but significantly minimizes the decoding latency. The decoding operation also requires minimal extra hardware (Section 5.3).

Since this encoding cannot eliminate the possibility of overflows, we try to minimize overflows and re-encryption by employing the techniques discussed below.

Resetting Deltas. When memory writes have spatial locality, we can expect delta values of contiguous memory blocks to grow at a comparable rate. This phenomenon can be exploited to reset delta values when they converge to an identical value.

Consider the example shown in Figure 5b. Over subsequent updates, all the deltas converge to an identical value, i.e., 8. When this happens, we can reset the deltas to zero and update the reference value. The final state in Figure 5b shows how the reference and delta are reset without having to re-encrypt storage because the counter values have not changed, only their representation. The reference value is incremented by 8, and all the deltas are set to 0.

This mechanism is especially effective in reducing re-encryption rate for workloads that have writes with frequent sequential writes.



(a) When a delta overflows, we re-encrypt the entire block group using the largest counter value in the group (128 in this example). We then reset the deltas, and store the counter as the reference.

(b) Resetting Deltas: We reset deltas to zero when they all converge to the same identical value, and also increment the reference value.

(c) On an overflow, we re-encode using smaller deltas. We subtract the minimum delta value (Δ_{min}) from all the deltas and increment the reference by Δ_{min}. If $\Delta_{min} == 0$, re-encoding is not possible.

Figure 5: Delta Encoding: Delta overflows are handled by re-encrypting the block group using the largest counter the group (shown in a). We apply two optimizations to reduce the rate of re-encryption (shown in b, and c).

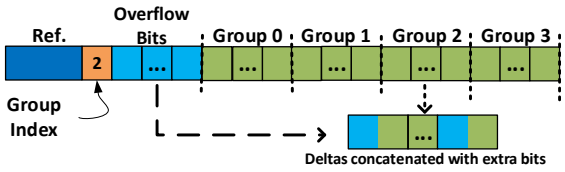


Figure 6: We encode deltas with using a dual-length encoding scheme, where a delta is supplied with extra "overflow bits" upon overflow. This scheme sacrifices optimality for low-overhead decoding .

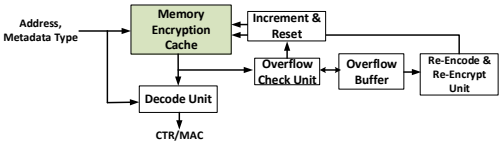


Figure 7: Implementation of Delta Encoding.

The results we present in Section 5.3 show that this technique significantly reduces the re-encryption rate on realistic workloads.

Re-Encoding Counters. If a delta cannot be reset and overflows, we will attempt to avoid (or at least defer) re-encryption by re-encoding the counters using a larger reference value.

In Figure 5c, the last delta would overflow if we increment it on the next write (assuming 7-bit deltas). Instead, we re-encode counters with a larger reference, using the following algorithm:

(1) Find the minimum delta, Δ_{min}, in the block group ($\Delta_{min} = 11$ in Figure 5c)
(2) Subtract Δ_{min} from all the deltas in the block-group.
(3) Add Δ_{min} to the reference value.

In short, we attempt to re-encode block group's counters using smaller deltas to avoid re-encryption. This algorithm is effective if all the deltas in the block group are greater zero , i.e. $\Delta_{min} \neq 0$.

4.4 Putting it All Together: Delta Encoding Implementation

Figure 7 shows the main components for implementing delta encoding/decoding and handling overflows.

The following extra hardware are introduced:

- **Decode Unit**: A small piece of hardware for extracting a delta value and adding it to the reference value is added. The decoding logic is very lightweight, involving a bit extraction and an add operation. These operations can be completed in just 2 cycles at high clock frequencies (Section 5.3).
- **Increment and Reset Unit:** On a write operation, this unit increments the deltas. Before incrementing the delta, an overflow is checked. After a successful increment operation, the reset logic checks if all the deltas are identical. The extra cycles required for these operations will not impact application performance as they are performed on a write.
- **Re-encoding and Re-encryption Unit:** When an overflow is detected, the address and counters of the block-group that need to be re-encrypted is enqueued to the overflow buffer for processing by the re-encryption engine. Before attempting an expensive re-encryption operation, the re-encryption engine will attempt to re-encode the counters using the algorithms described earlier.

Current industrial memory encryption implementations already contain hardware that can be leveraged for re-encoding and re-encryption. Intel SGX has logic for swapping out secure pages to an operating system accessible region. This process involves a re-encryption operation akin to the one we need to perform on overflows. Similarly, AMD's memory encryption hardware is fitted with a microcontroller that is responsible for re-encrypting pages during data migration [6]. Both of these existing hardware components can be enhanced to implement re-encoding and re-encryption without introducing major hardware.

CPU	3.2GHz, OoO, 4 cores, L1: 32KB, 8-way, L2: 256KB, 8-way, L3: 10MB, 16-way, shared
DRAM	4 channels, DDR3-1600
Memory Encryption	32KB, 8-way counter/MAC cache, 5-level Off-Chip Integrity Tree (protecting 512MB region)
Benchmark	PARSEC 2.1, sim-med input, 4-thread parallelism

Table 1: CPU was simulated using MARSSx86, integrated with DRAMSim2 DRAM simulator.

5 EVALUATION

5.1 Experimental Setup

We simulated the system described in Table 1 using the MARSSx86 cycle-accurate CPU simulator [9], integrated with DRAMSim2 [11]. We modified DRAMSim2 to implement the memory encryption.

Benchmarks. We run the PARSEC 2.1 benchmark [1] to completion with the *sim-med* input. We run 4-threads to better stress the memory system. We were able to run 11 of the 13 applications in benchmark suite, while the other two failed as they used instructions that are not fully supported by the CPU simulator.

Memory Integrity Tree. We allocated 512MBs of memory for secure data storage (protected by an integrity tree). We assume 3KB of on-chip SRAM is available [3] to store the top-level nodes of the tree. With this setup, the off-chip *baseline* integrity tree will have 5 levels. We configured the memory encryption engine with a 32KB, 8-way counter/MAC cache in all the experiments.

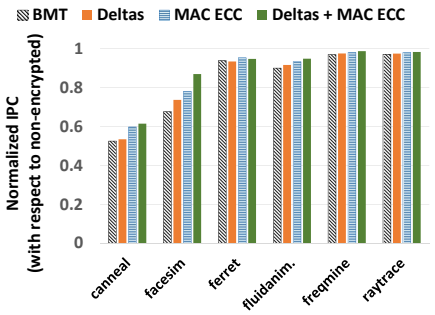


Figure 8: In addition to reducing storage overheads by ~ 10× our optimizations also improve the performance of memory encryption.

5.2 Performance Impact

Figure 8 presents the performance impact of authenticated memory encryption when our storage optimizations are enabled. It can be seen that our storage-optimized tree also reduces the performance impact of memory encryption by an average of 5% over the PARSEC benchmark suite compared to Bonsai Merkel Trees (for the reasons discussed below). Authenticated encryption has no measurable impact on some of the applications (bodytrack, vips, blackscholes, swaptions). These applications do not benefit from any further optimizations and are not shown in the figure. On the other hand, our optimizations improve the IPC of the the other applications by 1%-28% (compared to BMTs).

MAC-Based ECC. Our MAC-based ECC scheme enables us to read the MAC in parallel with the data, reducing the amount of memory transactions required to verify the integrity of blocks (Section 3). Avoiding these extra verification overheads improves IPC over the PARSEC benchmarks by an average of ~3% (with up to ~15% improvements) compared to Bonsai Merkel Trees (BMTs).

Delta Encoding. Delta encoding is able to improve application performance as compact counter storage reduces the depth of the Bonsai Merkel trees – resulting in fewer extra memory reads. For the system we evaluate, the depth of the tree is reduced from 5 to 4 levels when counters are delta encoded. Furthermore, the counter cache hit-rate also improves as we are fitting more counters into a single memory block. Our simulation models do not include the separate re-encryption logic, but its performance impact will be minimal as re-encryption can be performed without completely suspending the rest of the system.

5.3 Delta Encoding

This section analyzes the re-encryption rate, and the performance impact of counter decompression when employing delta encoding.

Counter Decoding Overhead. Counters need to be decoded by concatenating the deltas with extra overflow bits (or zeros), and then summing the base and delta. To measure the decoding overhead, we synthesized the decoder logic to IBM's 45nm silicon-on-insulator (SOI) technology library using Synopsis Design Compiler. With this setup, the decoding logic is able to complete within 2 cycles for frequencies up to 4GHz, and has negligible area overhead ($\sim 0.002mm^2$). Our simulations account for these 2 extra cycles. The other operations associated with delta encoding (counter reset, re-encoding, and re-encryption) are triggered during a write, and hence will not directly impact read latency. Furthermore, as discussed in Section 4.4, current industrial memory encryption implementations already contain hardware that can be leveraged for re-encoding and re-encryption.

Re-Encryption. If a delta overflows, re-encryption is triggered on the block-group (Section 4.2). Table 2 compares the re-encryption rate for different counter representations. It can easily be seen that, for memory intensive workloads, delta encoding (combined with our delta reset algorithms) significantly reduces the number of re-encryptions compared to split counters [13]. Dual-length deltas perform better than 7-bit deltas overall. On facesim, however, dual-length deltas have increased re-encryption rate as multiple delta-groups overflow the default, shorter 6-bit delta storages concurrently, and cannot be all extended using the reserved overflow bits. As discussed in Section 2.2, this makes delta encoding more efficient and non-volatile memory friendly.

Program	7-bit Minor Split CTR [13]	7-bit Delta	Dual Length Delta
facesim	880	113	176
dedup	725	51	14
canneal	167	167	128
vips	77	77	24
ferret	33	23	5
fluidanimate	4	4	0
freqmine	3	0	0
raytrace	2	2	0
swaptions	0	0	0
blackscholes	0	0	0
bodytrack	0	0	0

Table 2: Average Number of Re-Encryptions per *1 billion cycles*. Average across three full executions to account for variations in multi-threaded execution.

6 CONCLUSION

In this work, we presented two optimizations that can be combined to reduce the overhead of counter and MAC storage, while simultaneously lowering the performance impact of authenticated memory encryption by up to 28%.

We showed how the MAC storage overhead can essentially be eliminated on a system that already has ECC memory by substituting the 64 parity bits with a 56-bit MAC plus 7 parity bits. This combination of MAC and parity can be used for error detection and correction, as well as memory integrity checking. Furthermore, we show how the counter storage overhead can be significantly reduced by using delta encoding.

7 ACKNOWLEDGMENT

This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA.

REFERENCES

[1] C. Bienia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. PACT*, 2008.
[2] B. Gassend et al. Caches and Hash Trees for Efficient Memory Integrity Verification. In *Proc. HPCA*, 2003.
[3] S. Gueron. Memory Encryption for General-Purpose Processors. *IEEE S&P*, 2016.
[4] J. A. Halderman et al. Lest We Remember: Cold-Boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5), 2009.
[5] R. Huang and G. E. Suh. IVEC: Off-Chip Memory Integrity Protection for Both Security and Reliability. *ACM SIGARCH Computer Architecture News*, 2010.
[6] D. Kaplan, J. Powell, and T. Woller. AMD Memory Encryption Whitepaper.
[7] D. Lie et al. Architectural Support for Copy and Tamper Resistant Software. *ACM SIGPLAN Notices*, 2000.
[8] J. Meza et al. Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field. In *DSN*. IEEE, 2015.
[9] A. Patel et al. MARSS: A Full System Simulator for Multicore x86 CPUs. In *Proc. DAC*, 2011.
[10] B. Rogers et al. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS and Performance Friendly. In *Proc. MICRO*, 2007.
[11] P. Rosenfeld et al. DRAMSim2: A Cycle Accurate Memory System Simulator. *CAL*, 10(1), 2011.
[12] G. E. Suh et al. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proc. ISC*, 2003.
[13] C. Yan et al. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *Proc. ISCA*, 2006.
[14] V. Young et al. DEUCE: Write-Efficient Encryption for Non-Volatile Memories. In *Proc. ASPLOS*, 2015.