Self-Reinforcing Memoization for Cryptography Calculations in Secure Memory Systems

Xin Wang*, Daulet Talapkaliyev*, Matthew Hicks*, Xun Jian*
*Virginia Tech
xinw, kosovo, mdhicks2, xunj@vt.edu

***Abstract*—Modern memory systems use encryption and message authentication codes to ensure confidentiality and integrity. Encryption and integrity verification rely on cryptography calculations, which are slow. To hide the latency of cryptography calculations, prior works exploit the fact that many cryptography steps only require a memory block's write counter (i.e., a value that increases whenever the block is written to memory), but not the block itself. As such, memory controller (MC) caches counters so that MC can start calculating before missing blocks arrive from memory.**

Irregular workloads suffer from high counter miss rates, however, just like they suffer from high miss rates of page table entries. Many prior works have looked at the problem of page table entry misses for irregular workloads, but not the problem of counter misses for the irregular workloads.

This paper addresses the memory latency overheads that irregular workloads suffer due to their high counter miss rate.

We observe many (e.g., unlimited number of) counters can have the same value. As such, we propose memoizing cryptography calculations for hot counter values. When a counter arrives from memory, MC can use the counter value to look up a memoization table to quickly obtain the counter's memoized results instead of slowly recalculating them.

To maximize memoization table hit rate, we observe whenever writing a block to memory, increasing its counter to any value higher than the current counter value can satisfy the security requirement of always using different counter values to encrypt the same block. As such, we also propose a memoization-aware counter update: when writing a block to memory, increase its counter to a value whose cryptography calculation is currently memoized.

We refer to memoizing the calculation results of counters and the corresponding memoization-aware counter update collectively as Self-Reinforcing Memoization for Cryptography Calculations (RMCC).

Our evaluations show that RMCC improves average performance by 6% compared to the state-of-the-art. On average across the lifetimes of different workloads, RMCC accelerates decryption and verification for 92% of counter misses.

***Keywords*-memory confidentiality and integrity; counter-mode AES; memory subsystem; memoization**

I. Introduction

Moving computing in Cloud can lower cost for many companies. However, Cloud computing raises new security concerns as companies no longer control physical accesses to the servers running their applications.

To improve security, secure memory systems (e.g., Intel SGX [1]) ensure confidentiality and integrity. Hiding memory values from attackers ensures confidentiality; securely detecting malicious tampering of memory values ensures integrity. Specifically, when writing a memory block to memory, CPU ensures confidentiality by encrypting the block and ensures integrity by protecting the block with a message authentication code (MAC); when reading an encrypted block from memory, CPU decrypts the block and verifies its correctness via the MAC.

Cryptography calculations, such as Advanced Encryption Standard (AES), are at the heart of decryption and verification. Most steps of cryptography calculations require a block's counter (a.k.a, write counter value) as input. Each block has a dedicated counter; a block's counter increases whenever the block is written to memory.

Cryptography calculations are slow (e.g., evaluated as 40ns under recent prior works [2][3]), however. In this paper, we assume and evaluate 15ns - 22ns, according to AES latencies reported under 7nm synthesis [4]. This 15ns overhead equates to doubling DRAM row buffer hit latency.

To hide this long latency, CPUs today leverage the fact that many steps of cryptography calculations only require a block's counter, but not the block itself. As such, CPU's memory controller (MC) caches the counters [1] so that if a block's counter hits in the cache, MC can start calculating for the block before the block arrives from memory.

However, caching counters work poorly for large and/or irregular workloads. Even under space-efficient counter designs, a 64B counter block covers only one [5] or two [6] 4KB physical pages. As such, a counter block provides similar coverage as a normal page table entry. It is well-known that large and/or irregular workloads suffer from high miss rates of 4KB page table entries in the TLB. Many works have looked at high TLB miss rates for large and/or irregular workloads [7][8][9][10][11][12][13][14].

This paper addresses the memory latency overheads that large and/or irregular workloads suffer from due to their high counter miss rates. We observe many (e.g., an unlimited number of) counters can have the same value. For example, after an overflow under split counters [5][6], all blocks in the same page have the same counter value. Based on this observation, we propose memoizing cryptography calculations for hot counter values (see Figure 1). Because a counter value can cover many more memory blocks than

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00055

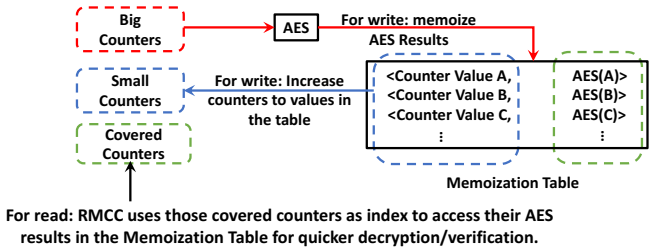


Figure 1: High-level overview of Self-Reinforcing Memoization for Cryptography Calculations (RMCC).

a counter block, memoizing calculations for counter values can be more space-efficient than caching counters. When a missing counter arrives at MC from memory, MC can use the counter value to look up a memoization table to quickly obtain the counter's memoized result instead of slowly recalculating the results.

A challenge with memoization is that although many (e.g., tens of thousands or more) counters can have the same values, it does not mean they actually do at runtime. For memoization to be more effective than counter caching, many counter values must actually be the same at runtime.

To address this challenge, we note counter mode encryption provides strong memory security by encrypting the same block differently every time the block is written to memory; we observe that simply increasing a block's counter value whenever it is written to memory, regardless of which higher value to increase the counter to, can ensure always using different counter values to encrypt the same block. As such, we propose memoization-aware counter update: after writing a block to memory, increase the counter to a value whose cryptography calculation is currently memoized. Over time, more and more counters will conform to values whose calculations are currently memoized. As a result, a few (e.g., 128) counter values will cover nearly all active blocks.

We refer to memoizing the results calculated from counters and the corresponding memoization-aware counter update collectively as Self-Reinforcing Memoization for Cryptography Calculations (RMCC).

To maximize coverage, RMCC only memoizes the arithmetic contribution of counters, instead of the full cryptography calculation, which also depends on address. Because a memory address is unique to a block, memoizing any address-dependent calculation benefits just one block and, thus, is ineffective. To memoize and reuse just the arithmetic contribution of counters, we propose a modified cryptography calculation that independently computes a partial result only from a block's counter and independently computes a partial result only from the block's address; RMCC quickly combines the two to derive the final result for encryption and verification. While only memoizing calculation results for counters does not speed up calculations for addresses, calculations for addresses are already fast because MC always knows addresses, unlike counter values, which must be fetched from memory when they miss in the cache.

This paper makes the following contributions:

- We address the latency overheads faced by large and/or irregular workloads due to frequent counter misses.
- We propose Self-Reinforcing Memoization for Cryptography Calculations (RMCC) to address the latency overheads of cryptography calculations after counter misses. RMCC accelerates decryption and verification for 92% of counter misses.
- RMCC improves average performance by 6% over a state-of-the-art baseline - Morphable Counters [6].

II. Background

For some companies, cloud computing can reduce cost compared to computing onsite. However, moving to Cloud requires companies to surrender control of physical accesses to the computing infrastructure. This opens up the possibility for malicious personnel with physical accesses to Cloud servers (e.g., disgruntled Cloud employees) to snoop and 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 [15], intended for system-level integration test and debugging, to read all values and commands transmitted over the memory bus.

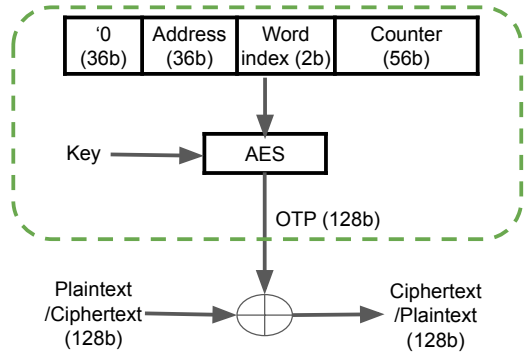
To improve memory security and trust for Cloud computing, Intel SGX [1] enforces memory confidentiality and integrity. Obfuscating memory values from attackers ensures confidentiality; securely detecting malicious memory tampering ensures integrity. When writing a block to memory, a CPU with SGX ensures memory confidentiality by encrypting the block and ensures integrity by protecting the block with a message authentication code (MAC); when reading from memory, CPU decrypts the block and checks its integrity by recomputing the MAC from the block and comparing it against the block's MAC stored in memory.

Memory encryption and verification rely on cryptography calculations. SGX uses Advanced Encryption Standard (AES) as its main cryptography calculation.

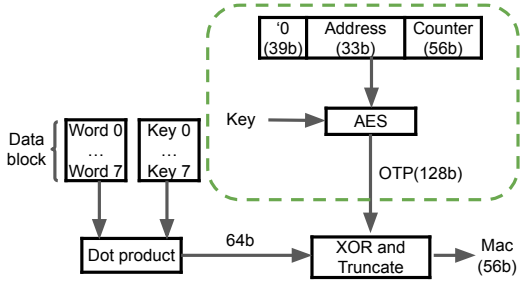
A. Memory Confidentiality

Before MC writes a block to memory, SGX first encrypts the block by calculating an AES result and bitwise XORs the result with the block's value to produce the ciphertext (see Figure 2a); MC only writes ciphertexts to memory.

Encryption: A primary input to AES is the write counter. The write counter input enables AES to always calculate different results whenever MC writes to memory, even for the same memory block. Hence, the AES results are called One-time Pads (OTPs). To calculate different OTPs for the same block, SGX protects every block with its own write counter and increases a block's counter each time MC writes the block to memory.



(a) Encryption and Decryption



(b) MAC Calculation.

Figure 2: Encryption, decryption, and MAC calculation for a block. Dashed steps can complete without the block.

Counters are stored in memory as 64B blocks, which we call counter blocks. Each counter block has eight 56-bit counters. *SGX stores counters in memory as plaintext, unlike normal blocks*.

Another input to AES is a block's memory address. Note that, in general, AES has a fixed input and output size of 128 bits. Because each block is 512 bit (i.e., 64B), encrypting a block requires four AES calculations, one for each of the four 128-bit words in the block. The four calculations use four different memory addresses, each for one of the four 128-bit words in the block.

Decryption: After reading the ciphertext of a block from memory, SGX decrypts the block. SGX first uses the block's addresses and write counter to recompute the block's OTPs and then bitwise XORs them with the ciphertext to recover the block's plaintext.

B. Memory Integrity

To reliably detect malicious tampering of memory values, SGX protects every memory block with a 56-bit MAC. When MC writes a block to memory, SGX calculates the block's MAC as the bitwise XOR between an OTP and a Galois Field (GF) dot product involving the block's values (see Figure 2b); the MACs are stored in memory. After fetching a block from memory, SGX recalculates the block's MAC and compares it with the MAC in memory to reliably detect differences and, thus, malicious tampering.

Like encryption, SGX uses a block's address and counter as inputs to generate the OTP to calculate the block's MAC.

To protect counter blocks themselves from malicious tampering (e.g., to launch replay attacks), SGX also protects counter blocks with MACs. Calculating a MAC for each counter block takes another write counter as input (i.e., each counter block itself is protected by another counter). SGX organizes those counters protecting counters in a tree, called the integrity tree.

C. Counter Cache

AES is the most time-consuming calculation for both decryption and verification. AES requires many serial rounds of computation, where each round consists of four serial transformations. SGX uses AES-128 (i.e., AES with 128-bit keys) [16]; AES-128 requires 10 sequential rounds. The stronger AES-256, which is quantum safe, requires 14 sequential rounds. In comparison, the remaining steps under decryption (i.e., bitwise XOR of ciphertext and OTP) and under verification (i.e., dot product) are much faster; in general, bitwise XOR and dot product are highly parallel.

The AES calculations to generate OTPs for decrypting and verifying a block does not use the block itself as input (see Figure 2); as such, MC can generate OTPs for a block before the block arrives from memory, as long as MC has the block's counter. Therefore, SGX caches the counters in the memory controller (MC) [1]; if the counter block hits in the counter cache, MC can hide most of the latency of AES. Counter blocks are more cache-friendly than normal blocks because each counter block covers eight normal blocks. MC accesses a counter block whenever MC accesses any of the eight normal blocks in memory; as such, each counter block has the combined locality of eight normal blocks.

When a counter block misses in the counter cache, the counter block must be fetched from memory and then verified; verifying a counter block fetched from memory requires using the counter block's counter block to recalculate a MAC for the counter block. To reduce the bandwidth overhead of accessing the integrity tree to verify counter blocks, MC also caches counter blocks' counter blocks (i.e., the integrity tree nodes) in the counter cache.

D. Prior works on Improving Counter Cache Hit Rate

Split counter designs (e.g., SC-64 [5], Morphable Counters [6]) pack many times more counters in each 64B counter block to cover more normal blocks per counter block. Unlike SGX, where each counter in a counter block is 56 bits, each counter in SC-64 [5] is only 7 bits; as such, SC-64 increases the number of normal blocks that each counter block covers from eight to 64. Covering more blocks per counter block exponentially increases the coverage of integrity tree nodes; for example, the number of normal blocks that each level-1 node in the integrity tree increases from 64 to 4096 - a factor of 8^2 increase. A later split counter design, Morphable

Counters [6], increases the number of blocks that each counter block protects even further - from 64 to 128,

One drawback of split counters is incurring bandwidth overhead by causing overflows. An overflow occurs when a counter value increases to a value that cannot be encoded by the smaller counter with fewer bits. SC-64 encodes each normal block's counter value as the sum of a 64-bit major counter shared across all 64 normal blocks in a page and a 7-bit minor counter dedicated to each block in the page. When a minor counter overflows, SC-64 increases all encoded counter values in the counter block to the maximum encoded counter value in the block. SC-64 does so by updating the shared major counter and all of the minor counters in the counter block; correspondingly, SC-64 fetches all normal blocks that the counter block covers, uses their new encoded counter values to compute their new ciphertext and MAC, and then writes the new ciphertexts and MACs to memory.

III. Characterizing the Problem

As dataset and memory system sizes increase over time, the memory size of many real-world applications have increased to hundreds of gigabytes [7][17][18][19]. Big in-memory datasets are general features of many server workloads such as data analytics frameworks and databases [20][21][22][23][24]. Many big memory workloads also have irregular memory access patterns. Large memory footprint and/or irregular memory access patterns causes high TLB miss rates. Many prior works have looked at the performance overhead that large and/or irregular workloads suffer due to their page table entries (PTEs) missing in TLBs [8][9][10][11][12][13].

Although the latest split counter design, Morphable Counters [6], provides a high coverage of 128 blocks or 8KB of memory per counter block, this coverage is still comparable to the coverage of PTEs; a PTE typically performs address translation for 4KB of memory content. Because counter block has comparable coverage as PTEs, which suffer from high miss rate for large and/or irregular workloads, we hypothesize that these workloads also suffer from high counter cache miss rate.

To test our hypothesis, we model Morphable Counters in Pintool [25] to measure counter cache miss rates for several workloads across their lifetimes. We select IBM GraphBig [26], which covers a broad scope of graph computing applications with irregular access patterns. We run GraphBig as four threads using the $8\_5 - fb$ Facebook-like dataset [27] as the input. We also evaluate canneal, omnetpp and mcf, which are used by recent prior works on improving address translation for irregular workloads [7][14]. We simulate 2MB of LLC per thread and 32KB of counter cache per thread. We run all Pintool experiments under 2MB standard huge pages to maximize counter hit rate for Morphable Counters. Morphable Counters is sub-optimal under 4KB pages; while each counter block in Morphable covers two adjacent 4KB physical pages (i.e., 128 continuous physical memory blocks), OS may map two adjacent 4KB virtual pages to two far-apart physical pages when using 4KB pages. Two far-apart physical pages require two different Morphable counter blocks, instead of one; this increases counter misses.

To measure the similarity between TLB miss rate and counter miss rate, we also measure TLB miss rate by using hardware performance counters to observe native executions of workloads on an Intel(R) Xeon(R) Gold 5120 CPU.

Figure 3 shows counter miss rate for LLC misses (i.e., what fraction of LLC misses suffer from counter cache miss) as measured in Pintool. Figure 4 show the number of TLB misses normalized to LLC misses as measured by hardware performance counters. The counter miss rate in Figure 3 correlates well with the TLB miss rate in Figure 4. Counter miss rate is generally high for workloads with high TLB miss rate (e.g., canneal); for workloads with low TLB miss rate (e.g., mcf), counter miss rate is also low.

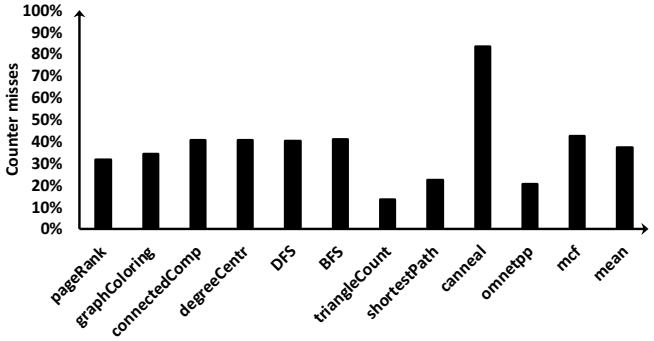


Figure 3: Total counter cache misses due to LLC misses normalized to total LLC misses

Figure 4 also shows TLB miss rate when running the workloads under 2MB huge pages. TLB miss rate is very low under huge pages. This is because each 2MB PTE covers

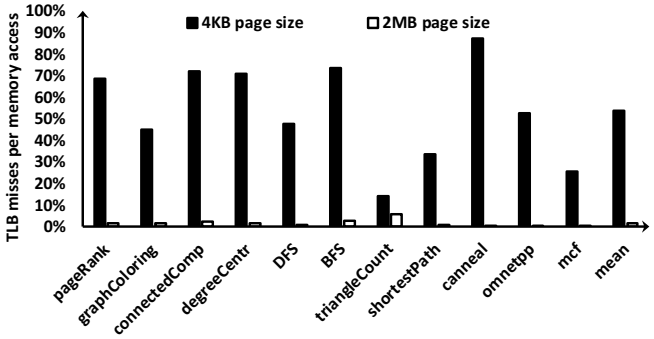


Figure 4: Total TLB misses (i.e., including TLB misses for accessing data that hit in cache) normalized to total LLC misses under normal page and huge page, respectively, during native execution without Pintool.

tens of thousands of memory blocks.

Unfortunately, there is no equivalent of 2MB pages for counters. We seek to achieve similarly high coverage for counters, so that counter values buffered in the MC can each cover over tens of thousands of memory blocks.

IV. Self-Reinforcing Memoization for Cryptography Calculations in Secure Memory

To address the high latency overhead of counter misses, we observe many counters can have the same value. For example, after an overflow, split counter designs increase the counters of all blocks in a page to the same counter value. In fact, because current CPUs and prior works independently update the counter values of unrelated memory blocks, unlimited number of memory blocks can happen to share the same counter value. As such, counter values can have orders of magnitude higher coverage than counter blocks.

Based on our observation above, we propose memoizing the arithmetic contribution of frequently-used counter values to OTPs and reuse these memoized results to accelerate decryption and verification for many normal blocks. We refer to the arithmetic contribution of a counter to an OTP as *counter-only result* and refer to the arithmetic contribution of an address to an OTP as *address-only result*. We propose only memoizing counter-only results because calculating address-only result is fast; address is always known when a memory request arrives at MC. Section IV-C5 describes how to independently calculate counter-only results and address-only results and combine them to produce the final OTPs to encrypt/decrypt/verify data.

Memoizing counter-only AES results can hide up to the full latency of AES calculation for counter misses.

Figure 5 illustrates the memory latency savings due to hiding the latency of calculating counter-only AES results. Without memoization, MC can only start calculating the OTP when the missing counter arrives from memory; serially computing OTP only after counter arrives increases the end-to-end latency of securely accessing memory. With memoization, when the missing counter arrives from memory, MC uses the counter's value to look up the memoization table to get the counter-only result and then combines the memoized counter-only result with the quickly calculated address-only result to produce the final OTP; this eliminates the long latency of slowly using the missing counter value to recalculating AES.

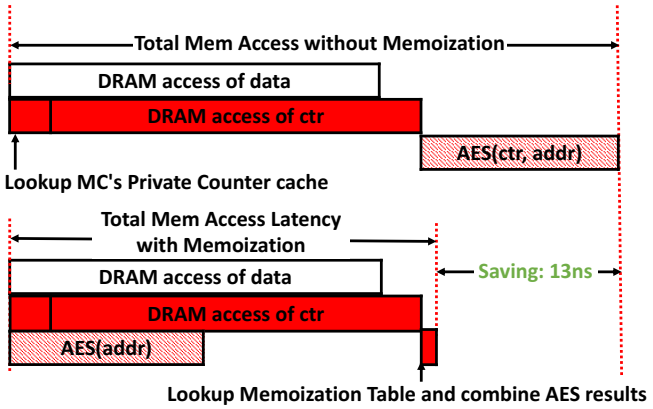


Figure 5: An example showing why memoizing AES results decreases total memory access latency. The example assumes DRAM row buffer miss and 15ns AES.

Memoization also applies to counters in the integrity tree (e.g., L1 or higher counters), instead of just normal blocks' counters (i.e., L0 counters). Under current systems and prior works, after fetching an L0 counter block from memory, MC verifies the L0 block by using the L0's block's counter; this is also called the normal block's L1 counter. Calculations using an L1 counter are stalled when the L1 counter block misses in the counter cache; as such, also memoizing AES results for L1 counters can further improve performance.

A. Challenges for Memoization

Challenge 1: One challenge is that while many (e.g., tens of thousands or more) counters can have the same values, they may not actually do at runtime. For example, different blocks can receive writebacks at different rates. Over a long time, different blocks' counter values can diverge vastly. For memoization to be more effective than just caching, many counters must actually have the same values at runtime.

Challenge 2: Another challenge is how to accurately identify hot (i.e., frequently-used) counter values. Getting frequent benefits from the memoization table requires memoizing AES results for hot counter values. Note that a counter value covering many blocks (i.e., many blocks currently have that same counter value) may not be a hot counter value; if the many blocks that a counter value covers are accessed rarely, memoizing the calculation for the counter value is useless.

Challenge 3: Even if the table currently memoizes a counter's AES result, a little later the table may not when MC updates the counter's value due to a memory write. As such, how to achieve high memoization table hit rate for frequently written data is another challenge.

B. Memoization-aware Counter Update

To address all three challenges above, we observe that when writing to a block, the block's counter can increase to any value. Counter mode encryption provides strong security by ensuring the same counter value will not be reused to encrypt the same data block; when writing a block back to memory, simply increasing the block's counter value, and not necessarily by just one, can ensure not reusing the same counter value to encrypt the same block.

Based on our observation, we propose a memoization-aware counter update policy to effectively address all three

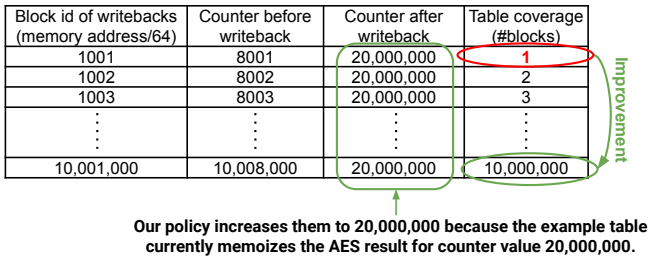


Figure 6: Memoization-aware counter update for an example memoization table that memoizes the counter-only result of just one counter value (i.e., counter value 20,000,000). The example spans 10 million memory writes. The table goes from covering only one block to covering 10 million blocks.

challenges in Section IV-A: *when writing a block to memory, increase the block's counter to the nearest counter value currently in the memoization table.* Over time, this memoization-aware counter update reinforces the counter values in the memoization table to have higher and higher coverage and become increasingly hot. It also enables a block to consecutively hit in memoization table after consecutive update of the block's counter value due to consecutive writebacks of the block to memory.

Specifically, memoization-aware counter update addresses Challenge 1 because, over time, it increases all accessed blocks' counters to counter values in the memoizaton table to maximize the table's coverage (see an example in Figure 6). It also addresses Challenge 2 because blocks that are more frequently read from memory also tend to be more frequently written back to memory; as such, the counter values of more frequently accessed blocks can be more quickly increased to counter values in the table. Memoization-aware counter update also addresses Challenge 3 because, during consecutive writebacks to the data block, it consecutively increases the block's counter value to the next value in the memoization table (see example in Figure 7).

We refer to our ideas of memoizing counter-only results and the corresponding memoization-aware counter update collectively as Self-Reinforcing Memoization for Cryptography Calculations (RMCC). Figure 8 provides an architectural overview of RMCC. We find that a 128-entry table can cover 92% of LLC misses that suffer from counter miss.

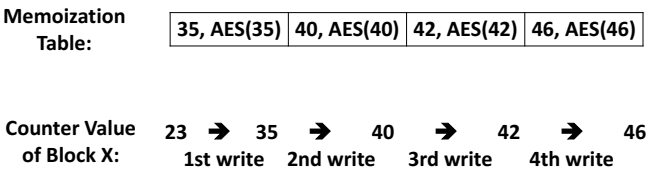


Figure 7: An example of how memoization-aware counter update addresses Challenge 3. The memoization table keeps covering a block's counter value even after the block undergoes many (e.g., N) writebacks to memory, assuming a table with many (e.g., N) entries.

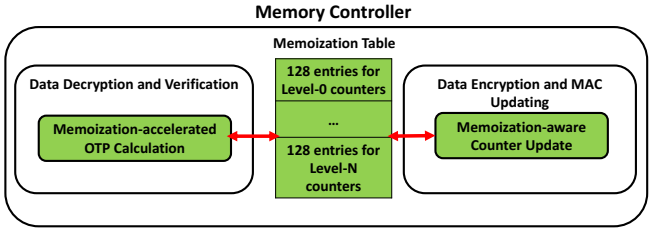


Figure 8: Architecture overview of RMCC.

C. Addressing other Practical Challenges

1) Read-heavy or Read-only Blocks: Another issue is that some blocks may write back rarely; as such, RMCC has little chance to increase their counter values to values in the memoization table. To address this issue, for read requests whose required counters miss in the table, RMCC also applies memoization-aware counter update to the block's counter, even though they are read requests, not writebacks.

Increasing counter values for memory reads, not just for memory writes, incurs a bandwidth overhead, however. To cap this bandwidth overhead, RMCC has a 1% budget of bandwidth overhead. The budget is replenished at the beginning of every epoch of 1,000,000 memory accesses. RMCC tracks the overhead traffic it causes due to memoization-aware counter update for data read requests whose counter values miss in the memoization table. When exceeding its budget, RMCC stops memoization-aware counter update for data read requests for the rest of the epoch. Leftover budget from one epoch is carried over to the next epoch.

2) Bandwidth Overhead when Applied to Split Counters: Split counters [6][5] can suffer from frequent counter overflows due to having few bits per counter; counter overflows incur costly bandwidth overhead (see Section II). When applied to split counters, RMCC can increase the rate of counter flows because memoization-aware counter update can increase the value of a counter by more than one (i.e., to match counter values in the memoization table).

To minimize the bandwidth overhead due to increasing counter overflows, RMCC always inserts a group of contiguous counter values (i.e., values X, X+1, X+2...) into its memoization table. Always inserting a group causes memoization-aware counter update to naturally increase counter values by just one for most memory writes; this reduces the rate of causing extra counter overflows. Specifically, RMCC always memoizes groups of eight consecutive counter values. We refer to a group of consecutive counter values whose counter-only results are recorded in the table as a Memoized Counter Value Group. The 128 memoization table entries in MC are organized as 16 Memoized Counter Value Groups (see Figure 9).

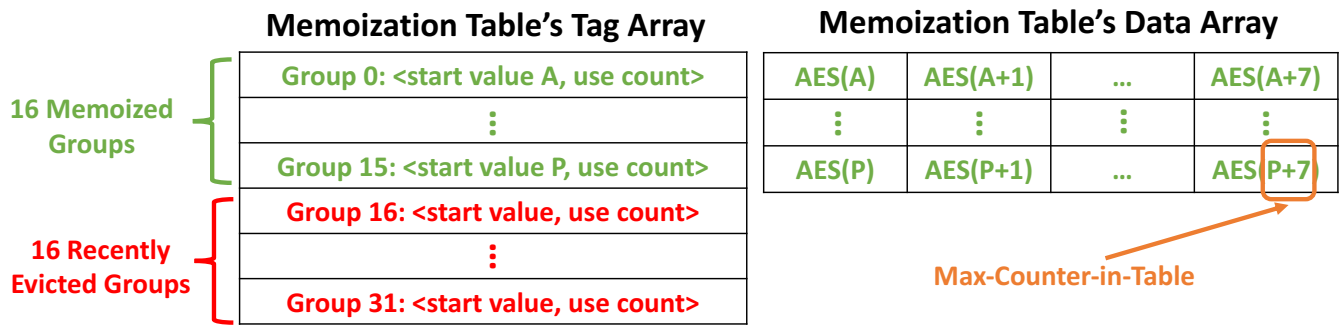


Figure 9: Memoization table organization.

Despite memoizing the counter-only results at the granularity of contiguous counter value groups, memoization-aware counter update still occasionally increases counter values by more than one and, therefore, can still incur some bandwidth overhead due to incurring extra counter overflow. We cap this bandwidth overhead within the per-epoch budget of 1% (see Section VI); 1% is a global budget that is shared across all sources of memory traffic overheads.

When budget runs out during an epoch, RMCC switches back to the baseline counter update policy for the rest of the epoch for most memory writes; RMCC only applies memoization-aware counter update to memory writes where the baseline counter update policy would also incur counter overflows. In this scenario, memoization-aware counter update relevels the counter values of an overflowing page to the nearest higher counter value in the table.

3) Counter Blocks with Very High Counter Values: Another issue is that some counters can have values that are bigger than the maximum counter value (i.e., Max-counter-in-Table) in the table. As such, memoization-aware counter update cannot increase these counters to counter values currently in the memoization table.

To address this issue, if during the epoch, RMCC encounters many (e.g., 2K) read requests whose counter values are greater than Max-counter-in-Table, RMCC inserts a new Memoized Counter Value Group; the values in the new group are higher than the current Max-counter-in-Table. Ideally, the new Memoized Counter Value Group should be higher than the counter values used by most (i.e., 98% of) read requests during the epoch. As such, RMCC monitors the following high counter values: X+1+8*i (i=0..16), $X+129+2^j$ (j=4..17), where X is Max-counter-in-Table (see Figure 9); RMCC tracks during each epoch how many read requests use a counter value that is smaller than each of the monitored values. For the start value of the new Memoized Counter Value Group, RMCC selects the smallest monitored counter value satisfying the 98% requirement.

When inserting a new Memoized Counter Value Group, the memoization table needs to replace an old Memoized Counter Value Group. RMCC replaces the least frequently used Memoized Counter Value Group. As such, RMCC tracks how frequently each Memoized Counter Value Group is used via a frequency counter per group that increments every time a value in a group is used to decrypt/verify a read request. After evicting a Memoized Counter Value Group, RMCC continues to maintain the group's use frequency counter, in a similar way in concept as shadow tags in prior works on cache replacement policies. RMCC maintains the use frequency counter for the 16 most recently evicted groups. We refer to the 16 most recently evicted groups as Evicted Counter Value Groups. At the end of each epoch, RMCC selects the 15 mostly frequently used groups out of all 32 groups and memoizes the counter-only results of their counter values.

4) Harnessing Evicted Counter Value Groups: Counter values within Evicted Counter Value Groups can still cover many blocks; memoizing AES results for these counter values can be beneficial. However, memoizing the AES results for all such counter values can double the size of the memoization table. Instead, RMCC memoizes the calculations for up to 16 most-recently-used counter values that fall under Evicted Counter Value Groups. Figure 10 shows that this additional optimization increases memoization hit rate by 6%, on average (see Section V Lifetime Characterization for methodology details). Because the composition of these 16 most-recently-used counter values can change with every memory access, RMCC's memoization-aware counter update does not seek to increase counters to have these values.

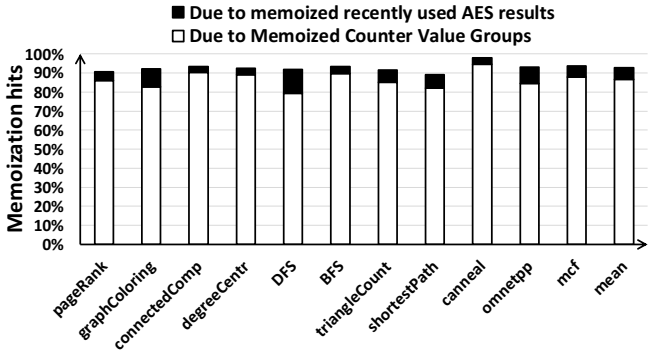


Figure 10: Memoization hit rate for counter misses.

5) Calculating OTP from Independently-Calculated Counter-only Result and Address-only Result: In current systems and prior works, each OTP is obtained from a single AES calculation that simultaneously takes as input both a data block's counter value and address(es).

RMCC calculates OTPs slightly differently; RMCC first independently calculates an address-only AES result and a counter-only AES result and then uses a truncated carry-less multiplication (see Figure 11) to combine the two into an OTP. The new OTP calculation enables MC to independently and quickly calculate address-only AES (e.g., without waiting for the counter to arrive from memory in the event of a counter miss); if the counter value hits in the memoization table, RMCC performs a fast carry-less multiplication on the address-only AES result and the recorded counter-only result to produce the final OTP.

For all OTP calculations (i.e., for encryption, decryption, and MAC), counter-only AES is calculated from scratch only if the counter value misses in the memoization table.

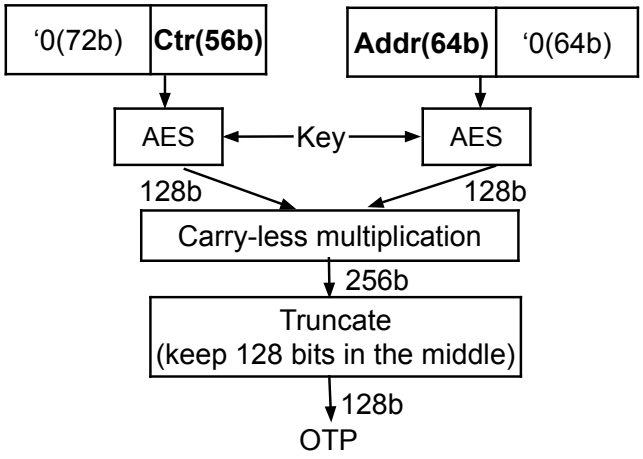


Figure 11: How RMCC calculates OTP. This OTP is a drop-in replacement for the OTPs for encryption and verification in Figures 2. RMCC also uses different keys for address-only AES calculation while calculating OTP for encryption, compared to calculating OTP for MAC, to ensure OTPs for encryption and MAC are different for the same block (like SGX [1]).

D. Security Analysis

Compared to existing CPUs and prior works, RMCC modifies OTP calculation and counter update policy. We discuss the security impact of both of these modifications.

1) RMCC's new OTP calculation: SGX' OTP calculation in Figure 2 guarantees no repeat within an unrealistically long lifetime of 2^{56} writebacks (i.e., until a 56-bit counter reaches an unencodable value of 2^{56}) [1]. When calculating OTPs by multiplying two values together, however, repeats are possible because A) multiplication is commutative (e.g., $6 \cdot 20 = 20 \cdot 6$) and B) multiplying completely different values can produce the same result (i.e., $6 \cdot 20 = 3 \cdot 40$).

RMCC eliminates type A) repeats by *prefixing* a counter value with 72 bits of zeros when calculating AES for counter and *appending* 64 bits of zeros to an address when calculating AES for address (see Figure 11); this ensures that the OTP of the block at address x and with counter value y differs from the OTP for the block at address y and with counter value x.

However, type B) repeats can still occur under RMCC, but with only a very small chance. Because the output of AES are random numbers, the truncated carry-less multiplication of two random numbers (i.e,. multiplication of a counter-only AES result and an address-only AES results) is also a random number. Because each OTP is 128 bits, the chance of any repeating OTP occurring during the targeted system lifetime of 2^{56} writebacks (i.e., the chance of having any repeat after generating 2^{56} OTPs) is the probability that out of 2^{56} numbers randomly chosen out of 2^{128} possible numbers, at least two are equal. Using the Birthday Problem formula, we calculate that only one in one hundred thousand machines will have repeat during their unrealistic lifetimes. For the one unlucky machine, an attacker cannot tell which two writebacks, out of the 2^{56}, have repeating OTP; even if the attacker can tell, he/she only gains the XOR of the two **random/undecidable** writebacks' values - a small loss, given such a miraculous attack.

Another security question that RMCC's OTP calculation may raise is that since OTPs are calculated by multiplying address-only AES and counter-only AES, can an attacker break a subset of OTP values back down to individual address-only AES results and counter-only AES results? If so, attacker can use the derived address-only AES results and counter-only AES results to calculate many more valid OTPs and use them to decrypt many other data blocks. Note that attackers can derive a subset of OTP values via known-text attacks (e.g., the ciphertexts of all-zero pages in memory are simply OTPs, under both baseline designs and RMCC). Also note that an attacker has to derive counter-only AES and address-only AES because RMCC only uses the final OTP to encrypt contents in memory; RMCC never directly use counter-only or address-only AES results to encrypt contents in memory.

Attackers cannot decompose OTPs into individual address-only and counter-only AES results through a better-than-brute-force attack. To understand why, consider the following worst-case scenario: A) all data blocks in memory have the same counter value and an attacker knows this (i.e., baseline systems store counter values in plaintext in memory) and, simultaneously, B) the attacker knows all 4N 16B OTPs in a memory system with N 64B blocks. For each OTP, the attacker can set up a new equation in the form of known OTP = truncate (unknown counter-only AES times unknown address-only AES). Specifically, the attacker

can use the 4N OTPs to set up 4N equations with 4N+1 unknowns (i.e., 4N different unknown address-only AES results and a single unknown counter-only AES result).

In the above worst-case scenario, the system of equations is unsolvable because the number of unknowns exceeds the number of equations.

Note that even if the attacker can set up a system with more equations than unknowns, the equations are still unsolvable. Because RMCC truncates 128 bits of information after multiplying counter-only and address-only results (see Figure 11), RMCC's OTP calculation is a highly lossy and, therefore, irreversible function that does not support any useful symbolic/algebriac manipulations; as such, the system of equations lacks all basic primitives for any analytic solution. The alternative of numerical solution would be prohibitively costly; it would involve creating for every equation a list containing all possible guesses of counter-only AES and address-only AES pairs and then cross examining all guess lists (e.g., find guess pairs where the guessed counter-only AES results match). However, since our OTP calculation cuts out 128 bits of information, the size of each equation's guess list is $\sim 2^{128}$. As such, the computation effort of populating just one equation's guess list can exceed that of a brute force attack on AES-128.

Most of the analysis above hinges on the assumption that our OTP calculations generate random numbers. We check this assumption empirically via NIST randomness tests [28]. Our OTPs pass NIST randomness tests at the same rate as the two streams of AES outputs used to calculate the OTPs.

2) Memoization-aware Counter Update: When applied to split counters, memoization-aware counter update can cause more frequent counter overflows by sometimes increasing split counter values by more than one. An attacker may launch a DoS attack by maliciously manipulating counter values to greatly increase the frequency of split counter overflows. However, DoS, which seeks to slow down applications by many factors or order(s) of magnitude, requires many overflows; such a large number of overflows would be hard to miss. As such, after encountering a large number of overflows in an epoch, RMCC can adaptively pause memoization-aware counter update and revert to baseline counter update policy.

When a block's counter value reaches 2^{56}, the block will start reusing old counter values because each counter has only 56 bits; reusing the same counter values for the *same* block is highly insecure. Preserving security when memory block(s) start reusing old values requires using a new key to encrypt everything in memory (e.g., via a system reboot), a very costly operation. By increasing each counter by only one per writeback, SGX can bound the frequency of reboot to no more than once per 2^{56} writebacks; the guaranteed 2^{56} writebacks without reboot corresponds to the worst case of always writing to just one and the same memory block. Guaranteeing no reboot before 2^{56} writebacks makes rebooting a negligible problem (e.g., less than once per eight years in the worst case [1]).

Memoization-aware counter update, however, increases counter values faster than SGX. Specifically, the choice of new Memoized Counter Value Groups, directly as described in Section IV-C, can cause an unrepresentable counter value before 2^{56} writebacks. Consider the worst-case scenario of always writing to just one and the same block; as the possible new Memoized Counter Value Groups start in counter values with large jumps in-between, memoizing a new Group in the table can cause memoization-aware counter update to increase the one and only written block's counter by more than one and, thus, cause counter reuse before 2^{56} writebacks in this worst-case scenario.

To bound the frequency of reboot to no more than once per 2^{56} writebacks just like SGX, RMCC can track the maximum encountered counter value for any data block (i.e., in an Observed System Max Counter Value Register) and only select new Memoized Counter Value Groups that start below System Max + 1. This ensures that the biggest counter value in the system (e.g., the counter value of the one and only written block in memory under the worst-case scenario) only increments by one at a time.

In practice, the maximum counter value in the system increases much slower than the worst-case of always writing to the same memory block. Our Pintool modeling and measurements across the entire lifetime of our target applications finds that RMCC increases the maximum counter value in the system by 24% over our baseline (i.e., Morphable Counters), on geometric mean across different applications.

E. Area Overhead

The memoization table requires 4KB to store 128 32B entries for Memoized Counter Value Groups; each entry has 16B AES result for decryption and 16B AES result for verification since decryption and verfication use different AES keys. The table needs 1KB to implement 64 16B counters to track the access rate for current groups, recently evicted groups, and new counter candidates. MC may maintain multiple tables (e.g., for different levels of counters/integrity tree.)

RMCC requires a truncated 128X128=128 carry-less multiplier to calculate OTP (see Figure 11). A fast design is to use 12K xor gates to compute and 16K inverters to increase the fanout of inputs. Assuming each XOR is 2X of an SRAM cell [29] and each inverter is half of an SRAM cell, the carry-less multiplier incurs an equivalent area of 4KB SRAM. The maximum gate depth is $log_2(128) = 7$ XOR and $log_4(128) = 3$ inverters.

V. Methodology

We evaluate performance using Gem5 [30], a cycle-accurate simulator. We evaluate the workloads used by recent prior works [7][31] on improving address translation.

Like our Pintool experiments in Section III, our Gem5 simulations run the workloads under 2MB huge pages. We use Ramulator [32] to simulate 128GB DDR4.

Baselines: We compare against Morphable Counters [6] as our primary baseline. As such, our evaluation applies RMCC on top of this baseline. We also evaluate the older Split Counters SC-64 [5].

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

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

Warmup and Observation Window: We fast forward each benchmark using Gem5's KVM mode with native execution speed to let each benchmark run into its region of interest. Then, similar to the Morphable Counters [6], we use atomic simulation to warm up the integrity tree for 25 billion instructions. After warming up the tree, 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.

Simulator Configuration: Table I shows the system setting of performance evaluation in Gem5. We use 128KB counter cache for RMCC, like Morphable Counters. We model the four-level integrity tree and counter overflows in Morphable Counters. Like [6], we co-locate data, its MAC, and error correction code in the same memory block; this enables data, its MAC, and ECC to be accessed together in one DRAM access without any memory traffic overhead. Figure 12 shows a breakdown of bandwidth utilization for different types of accesses under Morphable Counters.

To evaluate RMCC, we use two memoization tables, one for L0 counters and one for L1 counters. Each memoization table consists of 16 groups; each group memoizes AES results for eight consecutive counters.

We evaluate GraphBig [26] workloads under multi-threading. We evaluate each SPEC or PARSEC benchmark as single thread.

CPU	X86, 4 (or 1) core, 3.2 GHz, 4-wide OoO, 192 entry ROB
D-TLB, I-TLB	1536 entries each
Degree of constant stride prefetcher	L1: 1 L2: 2
L1 ICache/DCache	32/64 KB 4/8-way, 2ns
L2 Cache	1 MB 8-way, 4ns
L3 Cache	8 MB 16-way, 17ns
Counter Cache in MC	128KB 32-way
Decoding of Morphable Counters	3ns
AES-128 latency	15ns
Carry-less Multiplication Latency	1ns
Memoization Table in MC	128 entries for L0 counters 128 entries for L1 counters
Memory	128 GB DDR4
Memory Data Rate	3.2 GT/s
tCL, tRCD, tRP	13.75ns
tRFC	350ns
Row buffer policy	500ns timeout
Read/Write queue	256 entries
Channels, Ranks	1, 8
Mapping Function	XOR-based like Skylake [33]
Bank-level scheduling policy	FR-FCFS-Capped

Table I: System Configuration. Listed cache latencies are additive (e.g., end-to-end L2 hit latency is 2+4=6ns.)

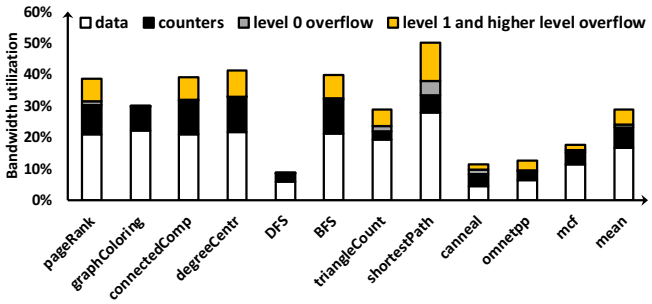


Figure 12: Memory bandwidth utilization due to normal memory accesses, counter accesses, and overflows under Morphable Counters. Bandwidth utilization is normalized to the memory channel's peak physical bandwidth.

Lifetime Characterization: To evaluate the effectiveness and overhead of RMCC, we run Pintool experiments across each benchmark's lifetime to get memoization table hit rate and bandwidth overhead of RMCC. We model multi-threaded GraphBig and single-threaded SPEC/PARSEC benchmarks in Pintool. We model in Pintool 1MB L2 cache, 2MB LLC and 32KB counter cache per core, same as the configuration in Gem5.

Rather than initializing all counter values to zero, we carefully initialize counter values in the beginning of each benchmark. If all counters are zero in the beginning, RMCC will work perfectly: RMCC just memoizes AES result for counter value zero so that all counters hit in the table initially. As such, to show how RMCC actually performs under real scenarios, we initialize all counters randomly. To achieve the randomization of all counters, we run a

write-intensive benchmark to individually increase counters to different big random values; explicitly, the benchmark continuously reads from and writes to randomly chosen data block across the entire memory. MC writes back each memory data block for 100000 times on average. Our Pin-tool simulations updates all states - caches, counter values, memoization table - through the entire experiment, including initialization phases.

VI. Results

Figure 13 shows the performance of RMCC, Morphable Counters [6], and SC-64 [34], normalized to a non-secure memory system that does not provide confidentiality and integrity. RMCC improves performance by 6%, on average, over Morphable Counters. Canneal receives the most performance benefit - 12.8%. Canneal benefits the most from RMCC due to having the highest miss rate in the MC's counter cache (see Figure 3). Since RMCC addresses the latency overheads due to counter misses, the benchmark with the most frequent counter misses naturally benefits the most from RMCC.

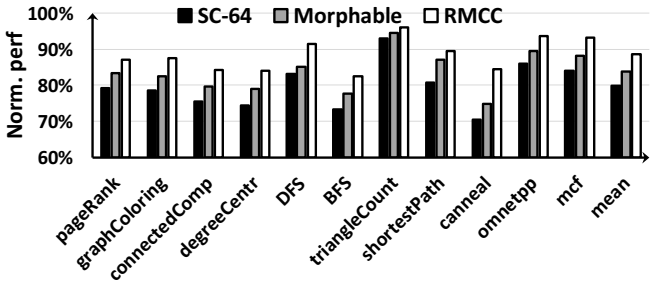


Figure 13: Performance of RMCC, Morphable, and SC-64 normalized to a non-secure memory system.

Figure 14 shows the LLC miss latencies for Morphable Counters, RMCC, SC-64 and non-secure system. RMCC saves, on average, 5.0ns on LLC data miss latency. For benchmarks showing highest performance improvement (e.g., canneal, connectedComp, BFS), they also have the highest savings on LLC miss latency.

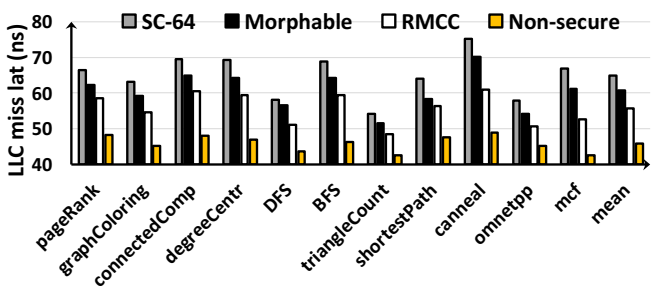


Figure 14: Average LLC miss latency under SC-64, Morphable, and RMCC.

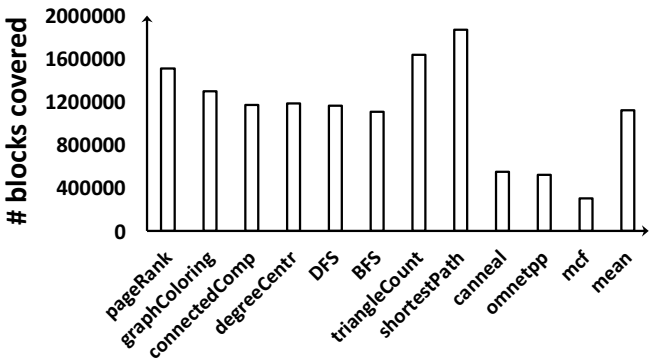


Figure 15: Average coverage of each counter value in the memoization table, across benchmarks' whole lifetimes.

The reduction in LLC miss latency is due to high memoization hit rate. 92.4% of LLC misses that suffer from L0 counter miss benefit from memoization (i.e., the corresponding L0 counter value's calculation is memoized). Similarly, 87% of L0 counter misses that suffer from L1 counter miss benefit from memoization (i.e., the calculation of the corresponding L1 counter value for the missing L0 counter is memoized). The high memoization hit rate is due to the high coverage of memoization table entries. Figure 15 shows the average number of memory blocks covered by each L0 counter value in the memoization table at the end of each workload. Each counter value in the memoization table covers 1.1 million blocks, on average.

All together, RMCC accelerates decryption and verification for 92% of counter misses; the accelerated counter misses encompass L0 counter misses that hit in the memoization table and have their L1 counters hit either in the counter cache or the memoization table.

RMCC incurs a bandwidth overhead, however, due to increasing counter values by more than one. Figure 16 shows the bandwidth overhead compared to Morphable Counters. RMCC is configured with a total of 2% traffic overhead budget. On average, RMCC consumes 4% higher memory bandwidth per instruction than Morphable Counters; this is close to the 2% traffic budget.

Sensitivity to AES Latency: As sensitivity analysis on AES latency, we also evaluate 22ns AES latency reported for the stronger AES-256 [4]. Since RMCC's benefits stem from hiding AES latency, RMCC is expected to provide higher performance benefit over Morphable Counters under the higher AES latency. As shown in Figure 17, the average performance improvement increases from 6%to 11% while AES latency increases from 15ns to 22ns.

Sensitivity to Counter Cache Size: We also evaluate counter cache sizes of 256KB and 512KB. Figure 18 shows the performance of RMCC under bigger counter caches. RMCC improves average performance over Morphable Counters by 5.4% and 5.0% under 256KB counter

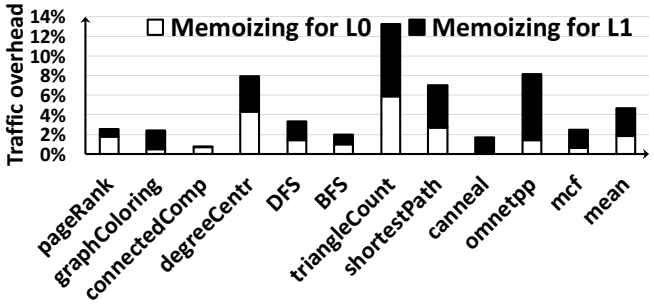


Figure 16: Memory bandwidth overhead of RMCC under 1% traffic overhead budget for memoizing AES for L0 counters and 1% traffic overhead budget for memoizing AES for L1.

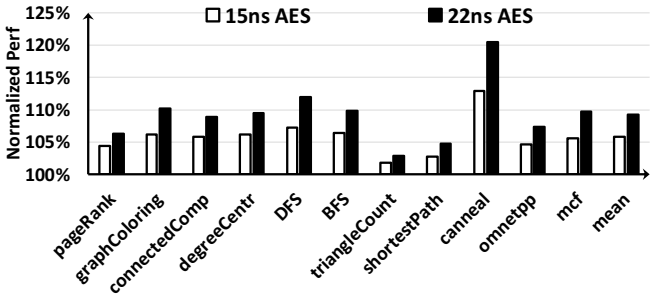


Figure 17: Performance of RMCC normalized to Morphable Counters under different AES latencies.

cache and 512KB counter cache, respectively.

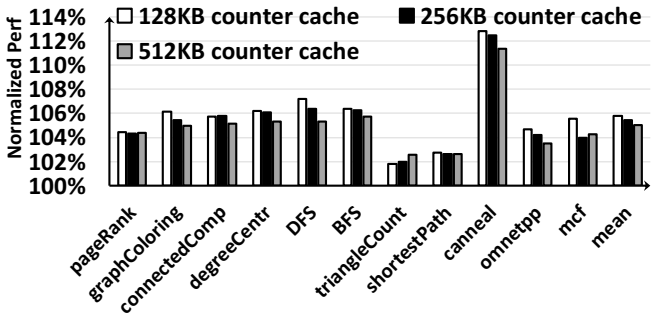


Figure 18: Performance of RMCC under 128KB, 256KB, 512KB counter cache, normalized to Morphable.

Sensitivity to Memoization Parameters: We simulate RMCC in Pintool to measure its behavior across the whole lifetime of applications.

Figure 19 show memoization hit rate under different bandwidth overhead budgets for memoizing AES calculations for L0 counters, across whole lifetime of each benchmark. In this final sensitivity analysis, we present memoization hit rate as the fraction of accessed counter values, regardless of counter cache hit or miss, that find their AES results are memoized in MC. Under 1% overhead budget, memoization hit rate is 92%; this is close to the 96% hit rate under 8% budget. As such, our primary evaluation uses 1% budget. All benchmarks benefit from $> 90\%$ memoization hit rate.

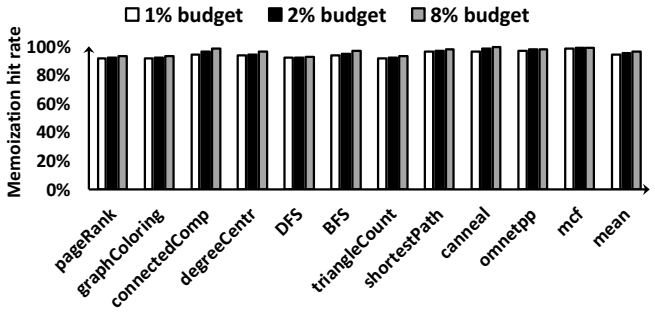


Figure 19: Memoization hit rate across application lifetime, under 1%, 2%, and 8% bandwidth overhead budgets,

RMCC suffers from memory traffic overhead due to incurring extra counter overflows (see Section IV-C2). RMCC caps this bandwidth overhead well for most benchmarks. Figure 20 shows the traffic overhead of RMCC over Morphable Counters under different bandwidth overhead budgets, normalized to total memory traffic of Morphable Counters. On average, RMCC incurs only 1.9% traffic overhead over Morphable under 1% overhead budget. RMCC's traffic overhead increases with higher budget. Under 8% budget, RMCC's overhead increases to 4%.

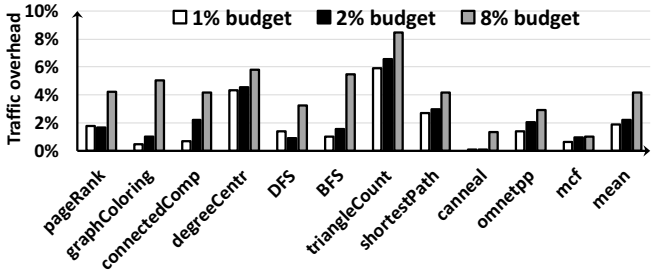


Figure 20: Memory traffic overhead of RMCC under 1%, 2%, and 8% bandwidth overhead budgets, across whole lifetime of each benchmark. This overhead is calculated by comparing total memory accesses under RMCC to total memory accesses under Morphable Counters.

To explore the effect of different memoization table organizations on RMCC, we also evaluate RMCC using different sizes of Memoized Counter Value Group (see Section IV-C), while keeping the total number of entries in the table the same (i.e., 128 entries). Figure 21 shows the hit rate of Memoization table of RMCC with different sizes for Memoized Counter Value Group. When the size of Memoized Counter Value Group is 8, RMCC achieves highest table hit rate 94%, on average. Figure 22 shows the traffic overhead for RMCC under different sizes of Memoized Counter Value Group. When the size of Memoized Counter Value Group

is 16, RMCC incurs the smallest traffic overhead - 2.1%, on average.

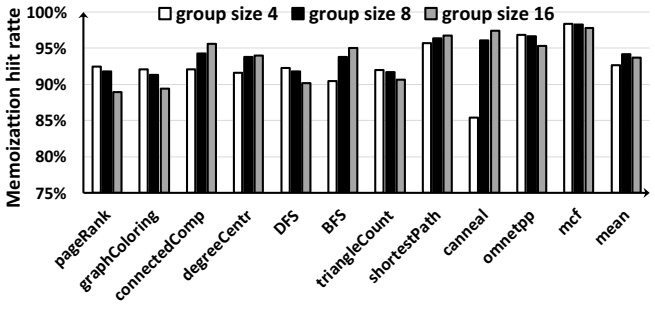


Figure 21: Memoization hit rate under different sizes of Memoized Counter Value Group under 1% overhead budget, across whole lifetime of each benchmark.

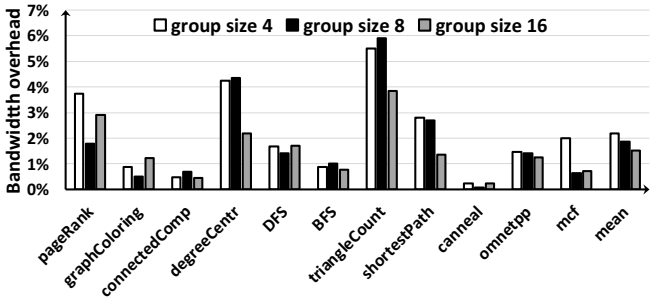


Figure 22: Memory traffic overhead of RMCC under different sizes of Memoized Counter Value Group under 1% overhead budget, across whole lifetime of each benchmark.

VII. Related Work

Speculative Verification: Some prior works [35][36] hide the latency of integrity verification by speculatively executing on data from DRAM before fully verifying them. The speculative execution is squashed if tampering is detected.

Speculative execution only hides the latency of verification, but not the latency of decryption; CPU cannot execute on ciphertext. As such, speculative execution only improves performance in memory access scenarios where verification is slower than decryption.

Also, speculation execution introduces security vulnerabilities of its own. To address them, the latest speculation execution, Poisionivy [35], makes deep pipeline changes (e.g. across registers, reorder buffer, MMU).

RMCC does not perform speculative verification (i.e. speculatively executing on younger load-dependent instructions while verifying the loaded data). Instead, RMCC speeds up verifying loaded data through memoization of AES results so that CPU can non-speculatively execute load-dependent instructions earlier.

OTP Pre-computation: To hide the latency overhead of decryption for memory accesses whose counters miss in the cache, OTP Prediction [37] proposes pre-computing (i.e., predictively computing) AES results using predicted counter values, similar to pre-fetching (i.e., predictively fetching) using predicted memory addresses. OTP Prediction predicts the counter values for other data blocks within a page after the 64B counter block for one of the page's data blocks arrives from DRAM. However, the state-of-the-art Morphable [6] that we use for performance comparison eliminates the need of this prediction. Under Morphable, a single 64B counter block holds all the counters for an entire page. As such, when the one counter block for a page arrives from DRAM, all counter values for the entire page are known; hence, no prediction is needed. OTP Prediction only predicts within a page because, intuitively, it is difficult to predict faraway unrelated counters across pages.

Pre-calculation for Persistent Memory: Many prior works [2][38][39][40] hide the latency of memory *encryption* (as opposed to decryption) in the context of persistent memory systems. In persistent memory programs, writes to persistent memory are on the critical path of program execution. To hide the latency of encryption for writes to persistent memory, Janus [2] provides a new software interface for programmers to explicitly initiate hardware encryption for persistent memory writes sooner. Our paper, however, focuses on hiding the latency of memory decryption and verification for reads from DRAM. RMCC is also fully software-transparent.

Lightweight Cryptography: Emerging lightweight ciphers, such as PRINCE [41] and QARMA [4], are faster than AES. However, trust is important to trusted computing; newer ciphers are not as trusted as AES, which has withstood longer and more scrutiny. Newer ciphers can be used where AES cannot. For example, ARM uses QARMA-64 for pointers, which are 64-bits; AES is undefined for 64-bit inputs. However, AES remains the standard for most systems.

VIII. Conclusion

Large and/or irregular workloads suffer from high counter miss rates. This paper addresses the latency overhead of cryptography calculations following counter misses. We propose RMCC to memoize cryptography calculations for hot counter values. When a missing counter arrives from memory, RMCC uses the counter value to look up a memoization table to quickly obtain the calculation memoized for the counter value to speed up cryptography calculations. To maximize memoization hit rate, we also propose a memoization-aware counter update. When writing a block to memory, RMCC increases the counter to a value whose AES result is recorded in the memoization table.

RMCC improves performance by 6%, on average, over the state-of-the-art (i.e., Morphable Counters). On average across the whole lifetimes of workloads, RMCC accelerates decryption and verification for 92% of counter misses.

ACKNOWLEDGMENT

We thank National Science Foundation NSF for supporting this work under grant 1850025. We thank Advanced Research Computing (ARC) at Virginia Tech for providing computational resources.

REFERENCES

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

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

[3] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 310–323.

[4] 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.

[5] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *33rd International Symposium on Computer Architecture (ISCA'06)*, 2006, pp. 179–190.

[6] 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.

[7] 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.

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

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

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

[11] J. Navarro, S. Iyer, and A. Cox, "Practical, transparent operating system support for superpages," in *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*, 2002.

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

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

[14] 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

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

[16] I. Anati, F. McKeen, S. Gueron, H. Haitao, S. Johnson, R. Leslie-Hurd, H. Patil, C. Rozas, and H. Shafi, "Intel software guard extensions (intel sgx)," in *Tutorial at International Symposium on Computer Architecture (ISCA)*, 2015.

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

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

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

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

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

[22] 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.

[23] 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.

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

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

[26] 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.

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

[28] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," Booz-allen and hamilton inc mclean va, Tech. Rep., 2001.

[29] A. R. Alameldeen, I. Wagner, Z. Chishti, W. Wu, C. Wilkerson, and S.-L. Lu, "Energy-efficient cache design using variable-strength error-correcting codes," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3, pp. 461–472, 2011.

[30] 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.

[31] J. Stojkovic, D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Parallel virtualized memory translation with nested elastic cuckoo page tables," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 84–97.

[32] 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.

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

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

[35] 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.

[36] W. Shi and H.-H. S. Lee, "Authentication control point and its implications for secure processor design," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 103–112.

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

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

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

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

[41] 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 *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2012, pp. 208–225.