To Appear in the Proceedings of the International Symposium on Computer Architecture (ISCA) – 2006

Improving Cost, Performance, and Security of Memory Encryption and Authentication *

Chenyu Yan†, Brian Rogers‡, Daniel Englender†, Yan Solihin‡, Milos Prvulovic†

†College of Computing
Georgia Institute of Technology
{cyan,denglend,milos}@cc.gatech.edu

‡Dept. of Electrical and Computer Engineering
North Carolina State University
{bmrogers,solihin}@ece.ncsu.edu

Abstract

Protection from hardware attacks such as snoopers and mod chips has been receiving increasing attention in computer architecture. This paper presents a new combined memory encryption/authentication scheme. Our new split counters for counter-mode encryption simultaneously eliminate counter overflow problems and reduce per-block counter size, and we also dramatically improve authentication performance and security by using the Galois/Counter Mode of operation (GCM), which leverages counter-mode encryption to reduce authentication latency and overlap it with memory accesses.

Our results indicate that the split-counter scheme has a negligible overhead even with a small (32KB) counter cache and using only eight counter bits per data block. The combined encryption/authentication scheme has an IPC overhead of 5% on average across SPEC CPU 2000 benchmarks, which is a significant improvement over the 20% overhead of existing encryption/authentication schemes.

1. Introduction

Data security concerns have recently become very important, and it can be expected that security will join performance and power as a key distinguishing factor in computer systems. This expectation has prompted several major industrial efforts to provide *trusted* computer platforms which would prevent unauthorized access and modification of sensitive or copyrighted information stored in the system. Unfortunately, such initiatives only provide a level of security against software-based attacks and leave the system wide open to hardware attacks [6, 7], which rely on *physically* observing or interfering with the operation of the system, for example by inserting a device on the communication path between the microprocessor and the main memory. Some of this communication path (e.g. the memory bus) is exposed outside the processor or main memory chips and can be tampered with easily [6, 7].

Clearly, software protection cannot adequately protect against hardware attacks, because program variables of the protection software itself can be stored in main memory and be subjected to hardware attacks. Instead, hardware *memory encryption and authentication* has been proposed [4, 5, 10, 11, 15, 16, 17, 19, 20, 21]. Memory encryption seeks to protect against passive (snooping) attacks on the secrecy/privacy of data and/or code. Memory authentication seeks to protect against active attacks on data integrity, which can modify existing signals and create new ones to cause programs to produce wrong results or behavior. Memory authentication is also needed to keep data and/or code secret because active attacks can tamper with memory contents to produce program behavior that discloses secret data or code [17].

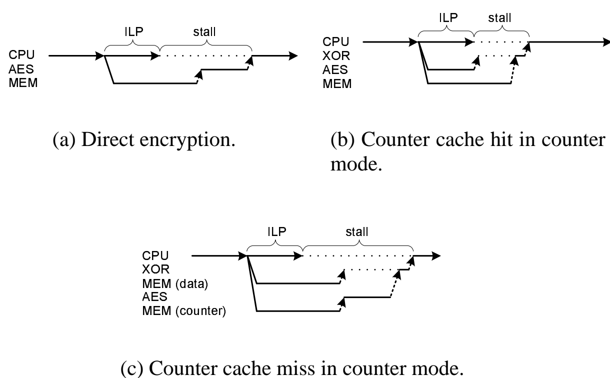
1.1. Background: Memory Encryption

Two main approaches have been proposed for memory encryption: direct encryption and counter mode encryption. In direct encryption, an encryption algorithm such as triple DES or AES [3] is used to encrypt each cache block as it is written back to memory and decrypt the block when it enters the processor chip again [5, 10, 11]. Although reasonably secure, direct encryption reduces system performance by adding decryption latency of a cryptographic algorithm (such as AES) to the already problematic L2 cache miss latency (Figure 1(a)). Counter mode encryption [2, 12] can be used to hide this additional AES latency [15, 16, 17, 19, 20, 21]. Instead of applying AES directly to data, counter mode encryption applies AES to a *seed* to generate a *pad*. Data is then encrypted and de-

* Yan and Prvulovic are supported in part by NSF Early Faculty Career Award CCF-0447783, NSF award CCF-0429802, and Georgia Institute of Technology; Rogers and Solihin are supported in part by NSF Early Faculty Career Award CCF-0347425 and North Carolina State University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, Georgia Tech, or NCSU.

crypted via a simple bitwise XOR with the pad. Although work on message encryption proves that the seed need not be secret to maintain secrecy of data encrypted in counter mode [1], it relies on a fundamental assumption that the same seed will never be used more than once with a given AES key [1, 12]. This is because the same seed and AES key would produce the same pad, in which case the attacker can easily recover the plaintext of a memory block if the plaintext of another block (or a previous value of the same block) is known or can be guessed.

To ensure unique seeds for memory encryption, we can keep a single *global counter* for all memory blocks which is incremented on any memory block write-back. Alternatively, per-block *local counters* can be used and the unique seed can be formed as a concatenation of data block's address and its local counter. The address part of the seed ensures that different locations are not encrypted with the same seed. The counter for a memory block is incremented on each write-back to that block to ensure that the seed is unique for each write-back to the same address. The counter value used to encrypt a block is also needed to decrypt it, so a counter value must be kept for each memory block. However, write-backs of a block could be frequent and counters can quickly become large. Thus, per-block counter storage must either be relatively large or a mechanism must be provided to handle counter overflow. In prior work [4, 15, 19, 20, 21], when a counter wraps around, the AES encryption key is changed to prevent re-use of the same pad. Unfortunately, the same encryption key is used to encrypt the entire memory, so a key change requires re-encryption of the entire memory. With a large memory, such re-encryption "freezes" the system for a noticeable time. For example, re-encryption of 4 GB of memory at a rate of 6.4 GB/second freezes the system for nearly one second. This is inconvenient for interactive applications and may be catastrophic in real-time systems. If small counters are used, such re-encryptions will occur frequently and become a significant performance overhead.



(a) Direct encryption.

(b) Counter cache hit in counter mode.

(c) Counter cache miss in counter mode.

Figure 1. Timeline of a L2 cache miss.

Using a larger counter field can reduce the frequency of these "freezes", but they degrade memory encryption performance. Counters are too numerous to keep them all on-chip, so they are kept in memory and cached on-chip in the *counter cache*, also called *sequence number cache* (SNC) in other studies. Counter mode hides the latency of pad generation by finding the block's counter in the counter cache and beginning pad generation while the block is fetched from memory, as in Figure 1(b). However, a counter cache miss results in another memory request to fetch the counter and delays pad generation as in Figure 1(c). A counter cache of a given size can keep more counters if each counter is small. As a result, a compromise solution is typically used where the counters are small enough to allow reasonably good counter cache hit rates, yet large enough to avoid very frequent re-encryptions. However, this compromise still results in using larger counter caches and still suffers from occasional re-encryption freezes.

1.2. Background: Memory Authentication

Current memory authentication schemes either have inadequate security protection or have high performance overheads. For example, authentication in XOM [5] cannot detect replay attacks. The log hash scheme in Suh et al. [19] employs lazy authentication in which a program is authenticated only a few times during its execution [19]. The Merkle tree scheme used in Gassend et al. [4] employs authentication in which instructions are allowed to commit even before their data is completely authenticated [4]. The Authenticated Speculative Execution proposed by Shi et al. [15] employs timely (i.e. non-lazy) authentication, but requires extensive modifications to the memory controller and on-chip caches. As pointed out by Shi et al. [17], *lazy* memory authentication sacrifices security because attacks can be carried out successfully before they are detected. Unfortunately, *timely* authentication delays some memory operations until authentication is complete. Furthermore, prior memory authentication schemes rely on MD-5 or SHA-1 to generate authentication codes but under-estimate their latency. Recent hardware implementations of MD-5 and SHA-1 show the latencies of more than 300ns [9], which is prohibitively expensive to use in timely authentication.

Unlike counter secrecy, which is unnecessary [1], undetected malicious modifications of counters in memory are a critical concern in counter mode memory encryption because they can be used to induce pad reuse and break the security of memory encryption. Although prior work addresses this concern to some extent through memory authentication, we find a flaw that has previously been ignored or unnoticed. We also find that this flaw can be avoided by authenticating the counters involved in data encryption, without a significant impact on performance.

1.3. Summary of Contributions

In this paper we present a new low-cost, low-overhead, and secure scheme for memory encryption and authentication. We introduce *split counter mode* memory encryption, in contrast with prior schemes which we refer to as *monolithic counters*. The counter in this new scheme consists of a very small per-block *minor counter* and a large *major counter* that is shared by a number of blocks which together form an *encryption page*[1]. Overflow of a minor counter causes only an increment of a major counter and re-encryption of the affected encryption page. Such re-encryptions are fast enough to not be a problem for real-time systems. They also result in much lower overall performance overheads and can be overlapped with normal processor execution using a simple additional hardware mechanism. Our major counters are sized to completely avoid overflows during the expected lifetime of the machine, but they still represent a negligible space and counter caching overhead because one such counter is used for an entire encryption page.

The second contribution of this paper is a significant reduction of memory authentication overheads, due to several architectural optimizations and our use of the combined Galois Counter Mode (GCM) authentication and encryption scheme [13]. GCM offers many unique benefits. First, it has been proven to be as secure as the underlying AES encryption algorithm [13]. Second, unlike authentication mechanisms used in prior work, GCM authentication can be largely overlapped with memory latency. Third, GCM uses the same AES hardware for encryption and authentication and GCM authentication only adds a few cycles of latency on top of AES encryption. In a recent hardware implementation [9], AES latencies of 36.48ns have been reported. This is a significant advantage compared to 300ns or more needed for MD5 or SHA-1 authentication hardware used in prior work on memory authentication. This low authentication latency, most of which can be overlapped with memory access latency, allows GCM to authenticate most data soon after it is decrypted, so program performance is not severely affected by delaying instruction commit (or even data use) until authentication is complete.

Finally, in this paper we identify and eliminate a pitfall of counter mode memory encryption schemes with local counters. This pitfall makes the system vulnerable to *counter replay attacks* in which an attacker forces a block to be encrypted with the same pad by rolling back the counter of the block. The attacker can perform this when a counter is replaced from the counter cache while its corresponding data block remains cached on-chip. To avoid such attacks, in addition to authenticating data, counters themselves need to be authenticated every time they are brought on-chip. We show that counter authentication can be achieved without significantly affecting performance.

The rest of this paper is organized as follows: Section 2 presents our split counter scheme and Section 3 presents our GCM authentication scheme, while Section 4 provides implementation details, Section 5 presents our evaluation setup, Section 6 discusses our evaluation results, and Section 7 presents our conclusions.

2. Split Counter Mode Encryption

The choice of a counter size is a major tradeoff in counter mode memory encryption. Small and medium-size counters can overflow and cause an entire-memory key change. This key changes results in a system "freeze" which, for small counters, can be frequent and cause significant performance overheads. Large counters do not overflow during the expected lifetime of the machine, but they incur larger storage overheads in main memory while performance suffers because fewer counters fit in the on-chip counter cache.

To keep overall counter sizes low and prevent costly key changes, we use a *split counter*, with a small (eight bits or less) per-block *minor counter* to reduce storage overheads and improve counter cache hit rates. We also use a large (64 bits) *major counter* that does not overflow for millennia and is shared by consecutive blocks that together form an *encryption page* which is a few kilobytes in size. The upper portion of Figure 2 illustrates the encryption process. When a block needs to be written back to memory, its major counter is concatenated with its minor counter to obtain its overall counter. For each encryption chunk (typically 16 bytes for AES encryption), a seed is obtained by concatenating the chunk's address, the block's counter, and a constant *encryption initialization vector* (EIV) [2]. For a 64-byte cache block size and 128-bit AES, there are four encryption chunks in a block. The encrypted chunks are then XORed with chunks of plaintext data. The figure shows that each 64-byte counter cache block stores a major counter (M) for a 4KB page and 64 7-bit minor counters ($m1, m2, \dots, m64$) for data blocks on that page. More detail on how to choose major and minor counter sizes is provided in Section 4.2.

When a minor counter overflows, we re-encrypt only its encryption page using the next major counter. Re-encryption of a relatively small encryption page is quick enough to avoid problems with real-time and interactive applications. As a result, split counter mode memory encryption eliminates problematic and costly entire-memory re-encryptions, while keeping the overall counter size small.

In addition to this main advantage, split counter mode memory encryption allows additional architectural opti-

[1]Our encryption page is similar in size to a typical system page (e.g. 4KB), but there is no other relationship between them. In particular, large system pages can be used to improve TLB hit rates without affecting the size of our encryption pages.

[2]The EIV can be unique per process, per group of processes that share data, unique per system, etc., depending on the needs for protection and sharing, and whether virtual or physical addresses are used.

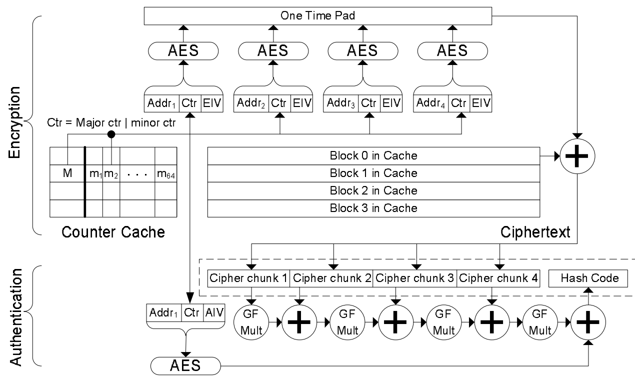


Figure 2. Split Counter Mode Memory Encryption and GCM Authentication Scheme.

mizations, one that dramatically reduces the overall re-encryption activity and the other to overlap normal processor and cache activity with re-encryption. Details of these optimizations are described in Section 4.2.

3. Memory Authentication with GCM

Memory authentication is needed to prevent hardware attacks that may compromise data integrity, such as attacks that modify data to change the application's behavior or produce erroneous results. Authentication is also needed to keep counter-mode memory encryption secure, because counters are stored in memory where an active attack can modify them (e.g. roll them back) and cause pad reuse. However, efficient, secure, and cost-effective memory authentication is difficult for several reasons.

The first reason for high overheads of memory authentication is that well-known authentication algorithms such as MD-5, SHA-1, or CBC-MAC have long authentication latencies, and this long-latency authentication begins when data arrives on-chip. As a result, the effective memory access latency is significantly increased if data brought into the processor chip can not be used before it is authenticated. On the other hand, use of data before it is authenticated presents a security risk [17]. Some use of data can safely be allowed before its authentication is complete, but only with relatively complex hardware mechanisms [15].

A second cause for the high overheads of memory authentication is a result of using of a Merkle tree [14]. A Merkle tree is needed in memory authentication to prevent replay attacks in which a data block and its authentication code in memory are replayed (rolled back to their previously observed values) together. Because the authentication code for the old data value matches the old value of the authentication code, the attack can remain undetected. In a Merkle tree, a leaf-level data block is authenticated by an authentication code. This code resides in a memory block which is itself authenticated by another code. If K codes fit in a block, the resulting K-ary tree eventually has a root authentication code, which can be kept in a special on-chip register where it is safe from tampering. This root code, in effect, prevents undetected tampering with any part of the tree. Codes at different levels of the tree can be cached to increase authentication efficiency. If each block (of data or authentication codes) is authenticated when it is brought on-chip, its authentication must proceed up the tree only until a tree node is found on-chip. Also, a change to a cached data or authentication code block does not need to immediately update the parent authentication node in the tree. The update can be performed when the block is written back to memory, at which time the update is propagated up the tree only to the first tree node which is still on-chip. Still, a data cache miss can result in misses at all levels of the authentication tree, in which case one block from each level must be brought from memory sequentially and authenticated in order to complete authentication of the data block. The resulting bus occupancy and authentication latency can be large, while effective caching of the multiple tree levels on-chip can be difficult. Also, if data and authentication codes are cached together this can result in significantly increased cache miss rates for data accesses.

The final cause of overheads is the size of authentication codes. The probability of an undetected data modification decreases in exponential proportion to the authentication code's size, but large authentication codes reduce the arity of the Merkle tree and increase both storage and performance overheads. For example, only four 128-bit AES-based authentication codes can fit in a 64-byte block, which for a 1GB memory results in a 12-level Merkle tree that represents a 33% memory space overhead.

We address the authentication latency problem by using Galois Counter Mode (GCM) [13] for memory authentication. This paper is, to our knowledge, the first to apply GCM in this setting. As illustrated in Figure 2, GCM is a counter-based encryption scheme which also provides data authentication. The encryption portion operates as a standard counter mode, by generating a sequence of pads from a seed and XORing them with the plaintext to generate the ciphertext. In our case, the plaintext is the data block and the seed is the concatenation of the block address, the counter value, and an initialization vector. Decryption is the same, except that plaintext and ciphertext are swapped. The authentication portion of GCM is based on the GHASH function [13], which computes a hash of a message ciphertext and additional authentication data based on a secret key. As shown in the lower half of Figure 2, the additional authentication data input is unused in memory authentication, and the GHASH function consists of the chain of Galois Field Multiplications and XOR operations on the chunks of the ciphertext. The final GHASH output is XORed with the authentication pad, which is generated by encrypting the concatenation of the block address, the counter, and another initialization vector. The resulting hash can be clipped to

fewer than 128 bits [13], depending on the desired level of protection.

We choose to use GCM because it has been studied extensively and shown to be secure [13], and because the latency to compute hash codes can be much less than using SHA-1 or MD5. As discussed in [13], the hashed data is sufficiently obscured as to be provably secure, assuming that the underlying block cipher is secure, and that no pad is ever repeated under the same key. We meet both conditions by using the AES block cipher and by ensuring that seeds are non-repeating since they are composed of an incrementing counter and the block address. The GHASH operation in GCM can be very fast, so the latency of GCM authentication can be much less than functions such as SHA-1 or MD5 because the heavy computation (generating the authentication pad using AES) can be overlapped with loading the data from memory. Once the ciphertext has been fetched, the GHASH function can be computed quickly on the ciphertext chunks because the field multiplication and XOR operations can each be performed in one cycle [13], and the final XOR with the authentication pad can be performed immediately afterwards because this pad has already been computed. Lastly, unlike SHA-1 or MD5 which require a separate authentication engine, GCM uses the same AES engine used for encryption.

To reduce the impact of the Merkle tree on authentication latency, we compute authentication codes of all needed levels of the authentication tree in parallel. Upon a data cache miss, we attempt to locate its authentication code on-chip. If the code is missing, we request its fetch from memory, begin generating its authentication pad, and attempt to locate the next-level code on-chip. This is repeated until an on-chip code is found. When the requested codes begin arriving from memory, they can be quickly authenticated as they arrive. Once the authentication chain from the data block to the original on-chip authentication code is completed, the data block can be used safely.

Finally, we consider increasing the arity of the Merkle tree by using smaller authentication codes. Smaller authentication codes reduce the memory space, bandwidth, and on-chip storage overheads of authentication, but degrade security in proportion to the reduction in code size. However, we note that the need for large authentication codes was established mostly to reliably resist even a long sequence of forgery attempts, e.g. in a network environment where each forged message must be rejected, but little can be done to prevent attacks. In contrast, a few failed memory authentications tell the processor that the system is under a hardware attack. Depending on the deployment environment, corrective action can be taken to prevent the attack from eventually succeeding. In a corporate environment, a technician might be called to remove the snooper from the machine and prevent it from eventually succeeding. In a game console, the processor may produce exponentially increasing stall cycles after each authentication failure, to make extraction of copyrighted data a very lengthy process. In both cases, it is assumed that the user or the software vendor is willing to tolerate a small but non-negligible risk of a small amount of data being stolen by a lucky guess. In many environments such a risk would be tolerable in exchange for significant reduction in performance overhead and cost.

4. Implementation

4.1. Caching of Split Counters

Our minor counters can be kept in a counter cache, similarly to how monolithic counters are cached in prior work. For major counters, a seemingly obvious choice is to keep them in page tables and on-chip TLBs. We note, however, that counters are needed only to service L2 cache misses and write-backs, and that large major counters may increase TLB size and slow down performance-critical TLB lookups. Another obvious choice is to keep major counters by themselves in a separate region of memory, and cache them on-chip either in a separate cache or in separate blocks of the counter cache. However, this complicates cache miss handling, because a single L2 cache miss can result in both a major and a minor counter cache miss.

As a result of these considerations, we keep major and minor counters together in memory and in the counter cache. A single counter cache block corresponds to an encryption page and contains the major counter and all minor counters for that page. With this scheme, a single counter cache lookup finds both the major and the minor counter. If the lookup is a miss, only one block transfer from memory is needed to bring both counters on-chip. Furthermore, we find that the ratio of counter-to-data storage can be easily kept at one byte of counters per block of data. An example for a 64-byte block size is shown in Figure 2, where a cache block stores one 64-bit major counter (M) and 64 seven-bit minor counters ($m1, m2, \ldots, m64$). If the L2 cache block size is also 64 bytes, a counter cache block corresponds to an encryption page of 4KB (64 blocks, 64 bytes each). As another example, a 32-byte block size in both the L2 and counter caches results in a counter cache block that stores one 64-bit major counter and 32 six-bit minor counters, with an encryption page size of 1KB. Our experiments indicate little performance variation across different block sizes, because reduced per-page re-encryption work with smaller encryption pages compensates for the increased number of re-encryptions caused by smaller minor counter size.

4.2. Optimizing Page Re-Encryption

With monolithic counters and with our new split counters, pad reuse on counter overflow must be prevented by changing another parameter used in pad generation. With monolithic counters, the only parameter that can be changed is the key, which is the same for an entire application and

its change results in entire-memory re-encryption. In our split counter approach, the major counter can be changed on minor counter overflow, and this change only requires re-encryption of one encryption page.

Memory access locality of most applications is such that some blocks are written back much more often than others. As a result, some counters advance at a much higher rate than others and overflow more frequently. Consequently, some pages are re-encrypted often, some rarely, and some never (read-only pages). With monolithic counters, the first counter that overflows causes re-encryption of the entire memory, so the rate of advance of the fastest-advancing counter controls the re-encryption rate for all blocks in the main memory. In contrast, with our split counters, the re-encryption rate of a block is determined by the rate of advance of the fastest-advancing counter on that page. Most pages are re-encrypted much less often than those that contain the fastest-advancing minor counters. This better-than-worst-case behavior of split counters results in significantly less re-encryption work than with monolithic counters. Our experimental results indicate that our split counter scheme with a total of eight counter bits per block (7-bit minor counters and 64-bit major counter shared by 64 blocks) on average results in only 0.3% of the re-encryption work needed when eight-bit monolithic counters are used.

In addition to performing less re-encryption work and splitting this work into smaller pieces to avoid lengthy re-encryption freezes, page re-encryption has an advantage over entire-memory re-encryption in that the re-encryption can be nearly completely eliminated from the processor's critical path. Conceptually, re-encrypting a block is a two-step process where the block is first decrypted by fetching it on-chip, and is encrypted again with a new major counter by writing it back to memory. In our page re-encryption, the first step (fetching blocks on-chip) only needs to be performed for those blocks that are not already on-chip. Our experimental results indicate that, on average, about half (48%) of the page's blocks are present on-chip when the page re-encryption is needed, which nearly halves the re-encryption latency and its use of memory, bus, and AES bandwidth. In contrast, in entire-memory re-encryption the blocks that are cached on-chip constitute only a small fraction of the main memory, and therefore do not noticeably reduce the re-encryption work.

Additionally, the second step (writing blocks back) does not require replacing already-cached blocks immediately. Since such blocks are likely still needed, we simply set such blocks to a dirty state, and let them be written back when they are naturally replaced from the cache. After the major counter for the page is changed and the minor counters are zeroed out, write-backs of such blocks will encrypt them with the new major counter. As a result of this "lazy" approach, re-encryption of on-chip blocks requires no extra memory reads or writes.

Finally, our encryption pages are small enough to permit tracking of the re-encryption status of each block within a page. Such tracking allows normal cache operation to proceed during page re-encryptions and nearly completely hides re-encryption latency. To accomplish this, our processor maintains a small number (e.g. eight) of *re-encryption status registers* (RSRs). Each RSR has a *valid* bit that indicates whether it is in-use or free. An RSR is tagged with an encryption page number, and it stores the *old major counter* for the page. An RSR corresponding to a page also maintains a *done* bit for each block on that page, to indicate whether the block has already been re-encrypted. Re-encryption of a page begins by finding a free RSR (with a zero *valid* bit), setting its *valid* bit to one, tagging the RSR with the page's number, copying the old major counter into the RSR, clearing all the *done* bits in the RSR, and incrementing the major counter in the counter cache. The RSR then issues requests to fetch the blocks of the page that are not already cached. As each block arrives from memory, the RSRs are checked. If the block's page matches an RSR and the block's *done* bit is not set, the block is decrypted using the old major counter from the RSR. Then the block's minor counter is reset, the *done* bit in the RSR is set, and the block is supplied to the cache and its cache state is set to dirty. To avoid cache pollution from blocks that are fetched by the RSR from memory, they are not cached and are immediately written back. Any write-back, regardless of whether it is cache-initiated or RSR-initiated, is performed normally, using the block's minor counter and its page's major counter from the counter cache. This completes re-encryption of a block if its page is being re-encrypted.

When the last *done* bit is set in the RSR, re-encryption of the page is complete and the RSR is freed by setting its *valid* bit to zero. To avoid re-encryption fetches of blocks that are already in-cache, the RSR looks up each block in the L2 cache before requesting that block from memory. For an already-cached block, the block's dirty bit is set and its *done* bit in the RSR is set immediately without re-fetching the block from memory.

With this support, the cache continues to service regular cache requests even for blocks in pages that are still being re-encrypted, and the processor is not stalled due to re-encryptions. An access to a block in a page that is being re-encrypted can either 1) find the block is already re-encrypted (*done* bit is one), in which case the access proceeds normally, or 2) find the block is being fetched by the RSR (*done* bit is zero), in which case the request simply waits for the block to arrive. Similarly, regular cache write-back of a block in a page that is being re-encrypted can proceed normally using the new major counter for the page.

We note that these optimizations would be difficult to achieve for entire-memory re-encryption, because it would be very costly to track the individual re-encryption status of the very large number of blocks involved in entire-memory

re-encryption. In our split counter approach, however, the optimizations can be applied relatively easily to completely avoid system freezes on re-encryptions and eliminate nearly all of re-encryptions' performance overhead.

In our scheme, cache operations can stall only when a write-back of a block causes another minor counter overflow while the block's page is still being re-encrypted, or when an RSR cannot be allocated because all RSRs are in use. The former situation is easily detected when a page's RSR allocation request finds a matching valid RSR, which can be handled by stalling the write-back until the RSR is freed. With a sufficiently large minor counters (larger than 4 bits), we find that the situation does not occur because a page re-encryption can be completed long before a new minor counter overflow triggers another re-encryption. The latter situation is also handled by stalling the write-back until an RSR becomes available. With a sufficient number of RSRs (e.g. 8), we find that the situation does not occur because there are typically very few pages that are being re-encrypted at the same time. Consequently, RSRs only introduce very small storage overheads of less than 150 bytes. Finally, RSR lookups do not introduce much overhead because in most cases it can be performed in parallel with cache misses.

4.3. Data and Counter Integrity Issues

As proven by [1], data secrecy can be maintained even if the counters in counter-mode encryption are themselves stored unencrypted. However, counter integrity must be protected because undetected counter tampering, such as rolling back the counter to its old value, may lead to pad reuse. We call such attacks *counter replay attacks*.

Protection of data integrity can help maintain counter integrity indirectly. The block's counter is used to decrypt the block which is then authenticated, and in GCM the counter is directly used in authentication of its data block. Because authentication would fail for a data block whose counter has been modified, we say that the counter is *indirectly authenticated* when the corresponding data block is authenticated.

In prior schemes, a data block is authenticated only when it is brought on chip. However, we observe that a data block may still reside on-chip while its counter is displaced from the counter cache to memory. When the data block is written back, the counter is re-fetched from memory to encrypt the block. However, the counter value from memory may have been changed to its older value to force pad reuse by an attacker. Therefore, a counter needs to be re-authenticated when it is brought on chip, before it is incremented and used for encrypting the block.

To ensure secrecy and integrity of the data, we build a Merkle tree whose leaf-level contains both the data and its *direct counters*. These are the counters directly involved in encryption and authentication of data blocks. Since the split counters in our scheme are small, the overhead for incorporating direct counters into the Merkle tree is also small. With GCM, in addition to direct counters, we need *derivative counters* which are used in authentication of non-leaf blocks in the tree. Since derivative counters are only used for authentication, data secrecy cannot be compromised by compromising the integrity of these counters.

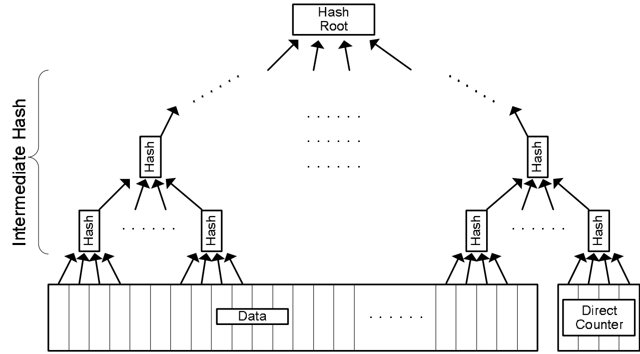


Figure 3. Data and counter Merkle Tree.

Figure 3 shows the resulting Merkle tree. The on-chip hash root guarantees the integrity of the data, the direct counters, and the other hash codes in the tree.

4.4. Other Implementation Issues

Dealing with Shared Data. In a multi-processor environment or for memory-mapped pages shared between the processor and I/O devices, data may be shared by more than one entity. Although dealing with such data is beyond the scope of this paper, we point out that recently proposed schemes for multiprocessor environments are based on counter-mode encryption [15, 21] and can easily be integrated with our split counters and GCM scheme.

Virtual vs. Physical Address. The block address that is used as a component of the block's encryption seed can be a virtual or physical address. Virtual addresses are more difficult to support because they are not directly available in the lowest level on-chip cache, and different processes may map the same location at different virtual addresses. Physical addresses are easier to use but require re-encryption when memory is paged from or to the disk. Our split counters and GCM mechanisms are orthogonal to these issues, and are equally applicable when virtual or physical addresses are used.

Key Management and Security. We assume a trusted operating system and a scheme that can securely manage keys and ensure they are not compromised. Our contributions are orthogonal to the choice of a key management scheme and a trusted platform, and can be used to complement the platform's protection against software attacks with low-cost, high-performance protection against hardware attacks and combined hardware-software attacks.

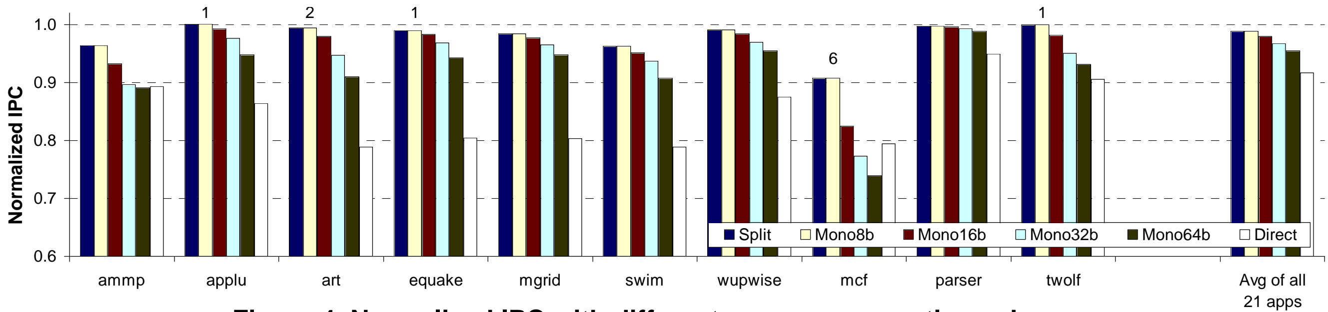


Figure 4. Normalized IPC with different memory encryption schemes.

5. Experimental Setup

We use SESC [8], an execution-driven cycle-accurate simulator, to model a three-issue out-of-order processor running at 5GHz, with 2-cycle 4-way set-associative L1 instruction and data caches of 16KB each, and with a unified 10-cycle 8-way set-associative L2 cache of 1MB. For counter-mode encryption and GCM, the processor also contains a 32KB, 8-way set-associative counter cache. All caches have 64-byte blocks. A block of our split counters consists of 64 7-bit minor counters and one 64-bit major counter, for a total block size of 64 bytes and an encryption page size of 4KB. The simulated processor-memory data bus is 128bits wide and runs at 600MHz, and below the bus the uncontended round-trip memory latency is 200 processor cycles. The 128-bit AES encryption engine we simulate has a 16-stage pipeline and a total latency of 80 processor cycles. This is approximately twice as fast as the AES engine reported in [9], to account for future technological improvements. The SHA-1 authentication engine is pipelined into 32 stages and has a latency of 320 processor cycles. This is more than 4 times as fast as reported in [9], to account for future technological improvements and possible developments that might give it an advantage over the AES engine used for GCM authentication. The default authentication code size is 64 bits, and we assume a 512MB main memory when determining the number of levels in Merkle trees. In addition to authenticating program data, we also authenticate counters used for encryption to prevent counter replay attacks described in Section 4.3. To handle page re-encryptions in our new split-counter mode, the processor is equipped with 8 re-encryption status registers (RSRs). Numerous other parameters (branch predictor, functional units, etc.) are set to reflect an aggressive near-future desktop machine, and all occupancies and latencies are simulated in detail.

Performance results in our evaluation are shown as normalized instructions-per-cycle (IPC), where the normalization baseline is a system without any memory encryption and authentication.

We use 21 of the SPEC CPU 2000 benchmarks [18], listed in Table 1. Only Fortran 90 benchmarks are omitted because we lack a Fortran 90 compiler for our simulator infrastructure. For each benchmark, we use its reference input set, in which we fast-forward 5 billion instructions and then simulate 1 billion instructions in detail.

SPECint 2000				SPECfp 2000		
bzip2	gap	mcf	twolf	ammp	applu	mgrid
crafty	gcc	parser	vortex	apsi	equake	swim
eon	gzip	perlbmk	vpr	art	mesa	wupwise

Table 1. Benchmarks used in our evaluation.

6. Evaluation

6.1. Split Counter Mode

Figure 4 compares the IPC of our split counter mode memory encryption (*Split*) with direct AES encryption (*Direct*) and with regular counter mode that uses 8-, 16-, 32-, and 64-bit counters (*Mono8b*, *Mono16b*, *Mono32b*, and *Mono64b*, respectively). No memory authentication is used, to isolate the effects of different encryption schemes. We only show individual applications that suffer at least a 5% performance penalty on direct AES encryption, but the average is calculated across all 21 benchmarks we use.

In our 1-billion-instruction simulations (less than one second on the simulated machine), we observe overflows of monolithic counters only in the *Mono8b* configuration and overflow of only minor counters in the *Split* configuration. Page re-encryptions in the *Split* configuration are fully simulated and their impact is included in the overhead shown in Figure 4. For *Mono8b*, we do not actually simulate entire-memory re-encryption, but rather assume it happens instantaneously and generates no memory traffic. However, we count how many times entire-memory re-encryption occurs and show the number above each bar. Note that our *Split* configuration with 7-bit minor counters and fully simulated page re-encryption has similar performance to the *Mono8b* configuration with zero-cost entire-memory re-encryption. From this we conclude that our hardware support for page re-encryption succeeds in removing the re-encryption latency from the processor's critical path.

To estimate the actual impact of entire-memory re-encryptions in long-running applications, we track the growth rate of the fastest-growing counter in each application. We use these growth rates to estimate the inter-

Apps	Counter Growth Rate (per second)					Estimated Time to Counter Overflow				
	Mono8b	Mono16b	Mono32b	Mono64b	Global32b (million)	Mono8b (seconds)	Mono16b (minutes)	Mono32b (days)	Mono64b (millennia)	Global32b (minutes)
applu	2090	2075	2035	1961	17.2	0.1	< 1	24	298,259	4
art	2039	2010	1943	1866	17.8	0.1	< 1	26	313,395	4
equake	1323	1314	1307	1272	3.2	0.2	< 1	38	459,914	22
mcf	1211	1101	1031	987	20.3	0.2	1	48	592,417	4
twolf	1079	1059	1026	1005	4.5	0.2	1	48	581,975	16
avg	633	626	577	596	5.9	0.4	2	86	981,417	12

Table 2. Counter growth rate and estimated time to overflow for different encryption schemes.

val between consecutive entire-memory re-encryptions with monolithic counters. The first four schemes in the tables use *locally incremented* counters, which are incremented when the corresponding data block is written back. It is also possible to use a *globally incremented* counter for encryption, where a single global counter is stored on-chip, incremented for every write-back, and used to encrypt the block. We note that the counter value used to encrypt the block must still be stored separately for each block so that the block can be decrypted. However, use of a global counter would eliminate the vulnerability we discuss in Section 4.3 without the need to authenticate direct counters.

Table 2 shows the counter growth rate and estimated time to counter overflow for the five applications with fastest-growing counters (applu, art, equake, mcf, and twolf). Averages across all benchmarks are also shown. We note that the growth rate decreases as counters become larger. This is the effect of lowered IPC with larger counters: the *number* of counter increments is nearly identical for all counter sizes, but with larger counter sizes the execution time is longer and the resulting counter increase *rate* is lower.

The global counter grows at the rate of write-backs in each application. With the 32-bit counter size, the global counter overflows in 12 minutes on average, much more frequently than in the 32-bit private counter scheme. We also noticed that although equake and twolf are among the top 5 for locally incremented counter growth rate, their numbers of write-backs per second are below the average. This is because these two applications have relatively small sets of blocks that are frequently written back, but the overall write-back rate is not very high.

Although few entire-memory re-encryptions were observed during the simulated one billion instructions in each application, we see that the counter overflow problem is far from negligible. Small 8-bit counters overflow up to ten times per second in some applications and every 0.4 seconds on average. Larger 16-bit counters overflow at least once per minute in some applications and every two minutes on average. Even 32-bit counters overflow more than once per month in some applications (applu and art in our experiments), which can still be a problem for real-time systems that cannot tolerate the freeze caused by an entire-memory re-encryption. We note, however, that 64-bit counters are free of entire-memory re-encryptions for many millennia.

With our split counters, we achieve the best of both worlds in terms of performance and counter overflow: small per-block minor counters allow small storage overhead in the main memory, good counter cache hit rates and good performance (Figure 4), while large per-page major counters prevent entire-memory re-encryptions for millennia (Table 2).

To help explain the performance of our split counters, we track the number of data cache blocks that are already resident on-chip when a page re-encryption is triggered. On average, we find that 48% of the blocks are already on-chip, which proportionally reduces the re-encryption work and overheads. The average time used for a page re-encryption is 5717 cycles. Note that normal processor execution continues during this time, although multiple (up to three) page re-encryptions can be in progress.

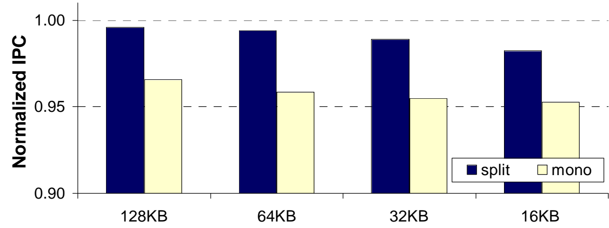
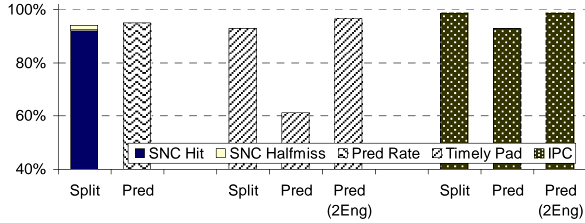


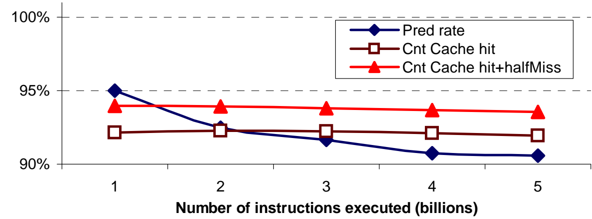
Figure 5. Sensitivity to counter cache size.

To determine how counter cache size affects the performance of our split counter mode memory encryption, we repeat the experiments from Figure 4 for different counter cache sizes from 16KB to 128KB. For the regular counter mode, we use 64-bit counters which do not cause entire-memory re-encryptions and system freezes. Figure 5 shows the average (across all 21 benchmarks) for each scheme. We see that, even with a 16KB counter cache, our split counter encryption (*split 16KB*) outperforms monolithic counters with 128KB counter caches (*mono 128KB*). The two schemes can keep the same number of per-block counters on-chip and have similar counter cache hit rates, but the *split 16KB* scheme consumes less bandwidth to fetch and write back its smaller counters.

Figure 6 compares our new split counter mode with counter prediction and pad precomputation scheme proposed in [16]. The counter prediction scheme associates a base counter with each page, and the counter for a block in that page is predicted as the base counter plus several small



(a) Performance indicators.



(b) Prediction and counter cache hit rate trends.

Figure 6. Comparison of split counters with counter prediction.

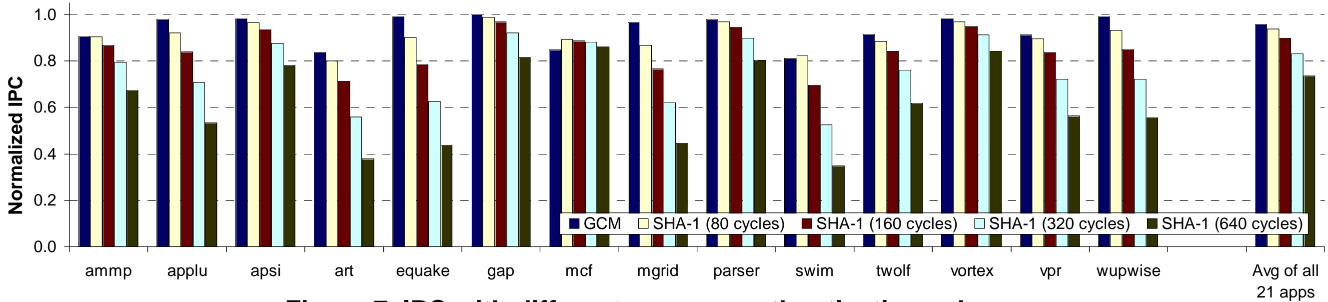


Figure 7. IPC with different memory authentication schemes.

increments. We note that the counter prediction scheme eliminates on-chip caching of its large 64-bit per-block counters, but they are still stored in memory and represent a significant overhead (e.g. with 64 bits per 64-byte block, the overhead is 1/8 of the main memory), while requiring modifications to the TLB and page tables. Moreover, this prediction involves pre-computing N pads with predicted counter values. This improves the prediction success rate, but increases AES engine utilization N-fold. We use N=5 in our experiments, as recommended in [16].

In Figure 6(a), the first group of results shows the hit and half-miss rate for the counter cache and the prediction rate for the counter prediction scheme. We observe that the counter prediction rate in the prediction scheme is slightly better than the counter cache hit rate in our scheme. The second group of results shows the percentage of timely pad pre-computations for memory read requests. In addition to results with one AES engine for our split counter scheme (*Split*) and counter prediction (*Pred*), we also show results for counter prediction with two AES engines (*Pred 2Eng*). Because it pre-computes five different pads for each block decryption, counter prediction requires significantly more AES bandwidth and, with only one AES engine, generates pads on time for only 61% of block decryptions. With two AES engines, counter prediction generates timely pads for 96% of block decryptions which is slightly better than the timely-pad rate of our scheme. We note that the area overhead for a deeply-pipelined AES engine could be quite significant [9]. The third group of results in Figure 6(a) shows the average normalized IPC. The *Pred 2Eng* scheme keeps large 64-bit counters in memory and fetches them with each data block to verify its predictions. The additional memory traffic offsets the advantage it has in terms of timely pad generation, and results in nearly the same performance as our split-counter scheme.

Figure 6(b) shows the trend of counter prediction rates in the counter prediction scheme and counter cache hit rates in our split counter scheme. As the application executes, our counter cache hit rate remains largely unchanged. In contrast, the counter prediction rate starts off with a high prediction rate, because all counters have the same initial value and are easily predicted. However, as counters change in value at different rates, their values become less predictable.

Note that our simulation results do not conflict with results reported in [16], where an extremely deeply pipelined AES engine is used to achieve very high AES bandwidth. Our additional experiments also confirm that the counter prediction scheme with two AES engines outperforms the Monolithic counter scheme with 64-bit counters. However, our split counter scheme with a 32kByte counter cache holds the same number of counters as a 256kByte cache with large monolithic counters, and needs far less bandwidth to fetch and write back its small counters.

6.2. GCM Authentication

Figure 7 compares our GCM memory authentication with SHA-1 memory authentication whose latency we vary from 80 to 640 cycles. No memory encryption is used, to isolate the effects of authentication, and the results are normalized to the IPC of a processor without any support for memory authentication. Note that no counter-mode encryption is used, so only GCM maintains per-block counters

needed for its authentication. As before, the average is for all 21 benchmarks, but we show individually only benchmarks with significant IPC degradation.

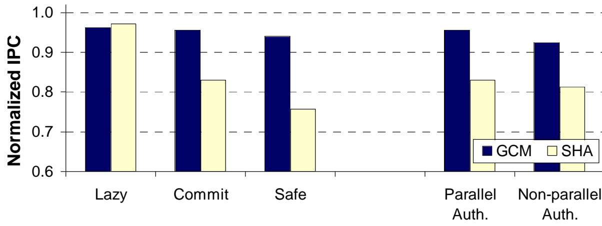


Figure 8. IPC with GCM and SHA-1 under different authentication requirements.

We observe that, in almost all cases, our GCM authentication scheme performs as well or slightly better than 80-cycle SHA-1, and it should be noted that 80 cycles is an unrealistically low latency for SHA-1. As the latency of SHA-1 is increased to more realistic values, the benefit of GCM authentication becomes significant, especially in *applu*, *art*, *equake*, *mgrid*, *swim*, and *wupwise*. On average, GCM authentication results in only a 4% IPC degradation, while SHA-1 with latencies of 80, 160, 320, and 640 cycles reduces IPC by 6%, 10%, 17%, and 26% respectively. The only case where GCM authentication performs relatively poorly is in *mcf*, due to additional bus contention caused by misses in the counter cache.

To determine how security requirements affect authentication performance, Figure 8 shows the IPC for our GCM scheme and for SHA-1 (with the default 320-cycle latency) with *Lazy* authentication in which an application continues without waiting for authentication to complete, *Commit* authentication in which a load that misses in the data cache cannot retire until its data has been authenticated, and *Safe* authentication in which a load stalls on a cache miss until authentication of the data fetched from memory is complete.

With *Lazy* authentication, the latency of authentication is largely irrelevant, and therefore bus contention for counter fetches and write-backs causes a slight degradation in GCM performance compared to SHA-1. However, as discussed in Section 3, this Lazy authentication presents a security risk, so a more strict form of authentication is desired. With *Commit* or *Safe* authentication, the latency of authentication becomes important and GCM has a considerable performance advantage. Even the strictest *Safe* authentication, which with SHA-1 results in a 24% IPC reduction, results in a tolerable 6% IPC reduction with GCM.

The second group of results in Figure 8 compares parallel authentication of all off-chip Merkle tree levels on a cache miss against sequential authentication of tree levels where the authentication of a level begins only when the previous level has been authenticated. Parallel authentication provides an average IPC increase of 3% for GCM and 2% for SHA-1. Although the IPC benefit seems modest, in terms of overhead reduction it is significant – with GCM, the IPC overhead of memory authentication is nearly halved with parallel tree-level authentication.

6.3. GCM and Split Counter Mode

Figure 9 shows our results when we use both memory encryption and memory authentication. Our combined GCM encryption and authentication scheme with split counters is shown as *Split+GCM*. We compare this scheme to a scheme that uses GCM with monolithic counters (*Mono+GCM*), a scheme that uses split-counter mode encryption with SHA-1 authentication (*Split+SHA*), a scheme that uses monolithic counters and SHA-1 authentication (*Mono+SHA*), and a scheme that uses direct AES encryption and SHA-1 authentication (*XOM+SHA*). As before, all IPCs are normalized to a system without any memory encryption or authentication. Only the benchmarks with significant differences among the schemes are shown individually, but again the average is for all 21 benchmarks. Our combined GCM mechanism with split counters results in an average IPC overhead of only 5%, compared to the 20% overhead with existing monolithic counters and SHA-1 authentication. As before, we note that split counters by themselves may seem a marginal improvement and that most of the benefit is due to the GCM authentication. However, we note that our split counters nearly halve the IPC overhead, from 8% in *Mono+GCM* to only 5% in *Split+GCM*.

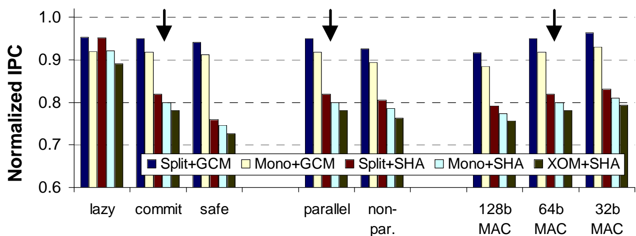


Figure 10. IPC with different authentication requirements.

We repeat the experiments from Figure 9 with different authentication requirements, with and without parallel authentication of Merkle tree levels, and using different authentication code sizes. These results are shown in Figure 10. The arrow in each set of experiments indicates our default configuration, and only one parameter is varied in each set of experiments. These results confirm our previous separate findings for GCM and split-counter mode, and indicate that our new combined scheme consistently outperforms previous schemes over a wide range of parameters, and that each of the two components of the new scheme (split-counter mode and GCM) also consistently provides performance benefits.

7. Conclusions

Protection from hardware attacks such as snoopers and mod chips has been receiving increasing attention in com-

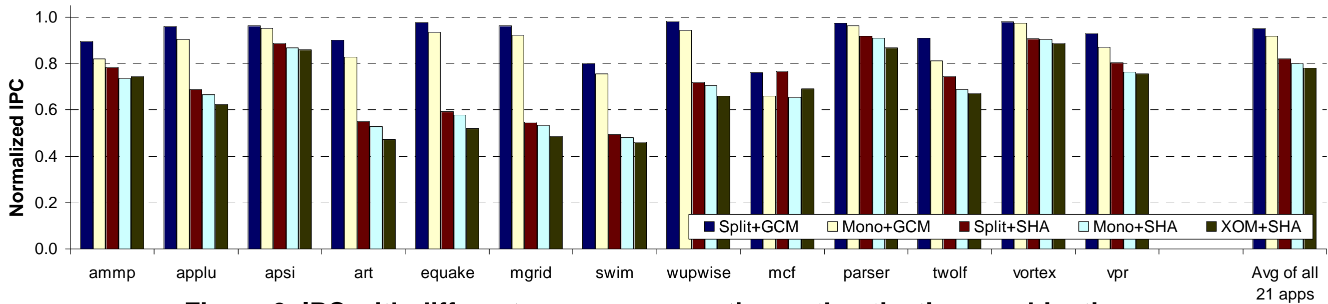


Figure 9. IPC with different memory encryption-authentication combinations.

puter architecture. In this paper we present a new combined memory encryption/authentication scheme. Our new split counters for counter-mode encryption simultaneously eliminate counter overflow problems and reduce per-block counter size to improve their on-chip caching. We also dramatically improve authentication performance and security by using GCM authentication, which leverages counter-mode encryption to reduce authentication latency and overlap it with memory accesses. Finally, we point out that counter integrity should be protected to ensure data secrecy.

Our results indicate that our encryption scheme has a negligible overhead even with a small (32KB) counter cache and using only eight counter bits per data block. The combined encryption/authentication scheme has an IPC overhead of 5% on average across SPEC CPU 2000 benchmarks, which is a significant improvement over the 20% overhead of existing encryption/authentication schemes. Our sensitivity analysis confirms that our scheme maintains a considerable advantage over prior schemes under a wide range of system parameters and security requirements.

References

[1] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption: Analysis of the des modes of operation. In *Proc. 38th Symp. on Foundations of Computer Science*, 1997.

[2] M. Dworkin. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. National Institute of Standards and Technology, NIST Special Publication 800-38C, 2004.

[3] FIPS Pub. 197. Specification for the Advanced Encryption Standard (AES). National Institute of Standards and Technology, Federal Information Processing Standards, 2001.

[4] B. Gassend, G. Suh, D. Clarke, M. Dijk, and S. Devadas. Caches and Hash Trees for Efficient Memory Integrity Verification. In *9th Intl. Symp. on High Performance Computer Architecture*, 2003.

[5] T. Gilmont, J.-D. Legat, and J.-J. Quisquater. Enhancing the Security in the Memory Management Unit. In *Proc. of the 25th EuroMicro Conf.*, 1999.

[6] A. Huang. *Hacking the Xbox: An Introduction to Reverse Engineering*. No Starch Press, San Francisco, CA, 2003.

[7] A. B. Huang. The Trusted PC: Skin-Deep Security. *IEEE Computer*, 35(10):103–105, 2002.

[8] J. Renau, et al. SESC. *http://sesc.sourceforge.net*, 2004.

[9] T. Kgil, L. Falk, and T. Mudge. ChipLock: Support for Secure Microarchitectures. In *Workshop on Architectural Support for Security and Anti-Virus*, 2004.

[10] D. Lie, J. Mitchell, C. Thekkath, and M. Horowitz. Specifying and Verifying Hardware for Tamper-Resistant Software. In *IEEE Symp. on Security and Privacy*, 2003.

[11] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. MItchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000.

[12] H. Lipmaa, P. Rogaway, and D. Wagner. Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption. http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes, 2000.

[13] D. A. McGrew and J. Viega. The Galois/Counter Mode of Operation (GCM). Submission to NIST Modes of Operation Process. http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes, 2004.

[14] R. Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, Department of Electrical Engineering, Stanford University, 1979.

[15] W. Shi, H.-H. Lee, M. Ghosh, and C. Lu. Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 123–134, 2004.

[16] W. Shi, H.-H. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High Efficiency Counter Mode Security Architecture via Prediction and Precomputation. In *32nd Intl. Symp. on Computer Architecture*, 2005.

[17] W. Shi, H.-H. Lee, C. Lu, and M. Ghosh. Towards the Issues in Architectural Support for Protection of Software Execution. In *Workshop on Architectureal Support for Security and Anti-virus*, pages 1–10, 2004.

[18] Standard Performance Evaluation Corporation. http://www.spec.org, 2004.

[19] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processor. In *Proc. of the 36th Intl. Symp. on Microarchitecture*, 2003.

[20] J. Yang, Y. Zhang, and L. Gao. Fast Secure Processor for Inhibiting Software Piracy and Tampering. In *Proc. of the 36th Intl. Symp. on Microarchitecture*, 2003.

[21] Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta. SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors. In *Intl. Symp. on High-Performance Computer Architecture*, 2005.