Appears in the Proceedings of the
40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), December 2007.

Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly [*]

Brian Rogers, Siddhartha Chhabra, Yan Solihin
Dept. of Electrical and Computer Engineering
North Carolina State University
{bmrogers, schhabr, solihin}@ncsu.edu

Milos Prvulovic
College of Computing
Georgia Institute of Technology
milos@cc.gatech.edu

Abstract

In today's digital world, computer security issues have become increasingly important. In particular, researchers have proposed designs for secure processors *which utilize hardware-based memory encryption and integrity verification to protect the privacy and integrity of computation even from sophisticated physical attacks. However, currently proposed schemes remain hampered by problems that make them impractical for use in today's computer systems: lack of virtual memory and Inter-Process Communication support as well as excessive storage and performance overheads. In this paper, we propose 1) Address Independent Seed Encryption (AISE), a counter-mode based memory encryption scheme using a novel seed composition, and 2) Bonsai Merkle Trees (BMT), a novel Merkle Tree-based memory integrity verification technique, to eliminate these system and performance issues associated with prior counter-mode memory encryption and Merkle Tree integrity verification schemes. We present both a qualitative discussion and a quantitative analysis to illustrate the advantages of our techniques over previously proposed approaches in terms of complexity, feasibility, performance, and storage. Our results show that AISE+BMT reduces the overhead of prior memory encryption and integrity verification schemes from 12% to 2% on average, while eliminating critical system-level problems.*

1. Introduction

With the tremendous amount of digital information stored on today's computer systems, and with the increasing motivation and ability of malicious attackers to target this wealth of information, computer security has become an increasingly important topic. An important research effort towards such computer security issues focuses on protecting the *privacy* and *integrity* of computation to prevent attackers from stealing or modifying critical information. This type of protection is important for enabling many important features of secure computing such as enforcement of Digital Rights Management, reverse engineering and software piracy prevention, and trusted distributed computing.

One important emerging security threat exploits the fact that most current computer systems communicate data in its plaintext form along wires between the processor chip and other chips such as the main memory. Also, the data is stored in its plaintext form in the main memory. This presents a situation where, by dumping the memory content and scanning it, attackers may gain a lot of valuable sensitive information such as passwords [12]. Another serious and feasible threat is *physical* or *hardware attacks* which involve placing a bus analyzer that snoops data communicated between the processor chip and other chips [7, 8]. Although physical attacks may be more difficult to perform than software-based attacks, they are very powerful as they can bypass any software security protection employed in the system. The proliferation of mod-chips that bypass Digital Rights Management protection in game systems has demonstrated that given sufficient financial payoffs, physical attacks are very realistic threats.

Recognizing these threats, computer architecture researchers have recently proposed various types of secure processor architectures [4, 5, 13, 14, 16, 17, 18, 19, 20, 22, 23, 24, 25, 26]. Secure processors assume that off-chip communication is vulnerable to attack and that the chip boundary provides a natural security boundary. Under these assumptions, secure processors seek to provide private and tamper-resistant execution environments [23] through *memory encryption* [5, 13, 14, 16, 18, 19, 20, 22, 23, 24, 25, 26] and *memory integrity verification* [4, 14, 16, 17, 18, 20, 22, 23, 24, 26]. The chip industry also recognizes the need for secure processors, as evident, for example, in the recent effort by IBM in the SecureBlue project [9] and Dallas Semiconductor [15]. Memory encryption protects computation privacy from *passive attacks*, where an adversary attempts to silently observe critical information, by encrypting and decrypting code and data as it moves on and off the processor chip. Memory integrity verification protects computation integrity from *active attacks*, where an adversary attempts to modify values in off-chip storage or communication channels, by computing and verifying Message Authentication Codes (MACs) as code and data moves on and off the processor chip.

Unfortunately, current memory encryption and integrity verification designs are not yet suitable for use in general purpose computing systems. In particular, we show in this paper that current secure processor designs are *incompatible* with important features such as *virtual memory*, *Inter-Process Communication* (IPC), in addition to having large *performance* and *storage* overheads. The challenges are detailed as follows:

Memory Encryption. Recently proposed memory encryption schemes for secure processors have utilized *counter-mode encryption* due to its ability to hide cryptographic delays on the critical path of memory fetches. This is achieved by applying a block cipher to a *seed* to generate a cryptographic *pad*, which is then bit-wise XORed with the memory block to encrypt or decrypt it. A seed is selected to be independent from the data block value so that pad generation can be started while the data block is being fetched.

In counter-mode encryption, the choice of seed value is critical for both security and performance. The security of counter-mode *requires* the uniqueness of each pad value, which implies that each

[*]This work is supported in part by the National Science Foundation through grants CCF-0347425, CCF-0447783, and CCF-0541080.

seed must be unique. In prior studies [16, 18, 19, 20, 23, 24, 25, 26], to ensure that pads are unique across different blocks in memory (*spatial uniqueness*), the block address is used as one of the seed's components. To ensure that pads are unique across different values of a particular block over time (*temporal uniqueness*), a counter value which is incremented on each write back is also used as a seed component. From the performance point of view, if most cache misses find the counters of the missed blocks available on-chip, either because they are cached or predicted, then seeds can be composed at the cache miss time, and pad generation can occur in parallel with fetching the blocks from memory.

However, using the address (virtual or physical) as a seed component causes a significant system-level dilemma in general purpose computing systems that must support virtual memory and Inter-Process Communication (IPC). A virtual memory mechanism typically involves managing pages to provide process isolation and sharing between processes. It often manages the main memory by extending the *physical memory* to *swap memory* located on the disk.

Using the physical address as a seed component creates re-encryption work on page swapping. When a page is swapped out to disk and then back into memory, it will likely reside at a new physical address. This requires the blocks of the page to be decrypted using their previous physical addresses and re-encrypted with their new physical addresses. In addition, encrypted pages in memory cannot be simply swapped out to disk as this creates potential pad reuse between the swapped out page and the new page at that physical address in memory. This leaves an open problem as to how to protect pages on disk. We could entrust the OS to encrypt and decrypt swapped pages in software if the OS is assumed to be authentic, trusted, and executing on the secure processor. However this is likely not the most desirable solution because it makes the secure processor's hardware-based security mechanisms contingent on a secure and uncompromised OS. Alternatively, we could rely on hardware to re-encrypt swapped pages, however this solution has its own set of problems. First, this requires supporting two encryption methods in hardware. Second, there is the issue of who can request the page re-encryptions, and how these requests are made, which requires an extra authentication mechanism.

Using virtual address as a seed component can lead to vulnerable pad reuse because different processes use the same virtual addresses. While we can prevent this by adding process ID to the seed [24], this solution creates a new set of serious system-level problems. First, this renders process IDs non-reusable, and current OSes have a limited range of process IDs. Second, shared memory based inter-process communication (IPC) mechanisms are infeasible to use (e.g. mmap). The reason is that different processes access a shared page in memory using different combinations of virtual address and process ID. This results in different encryptions and decryptions of the shared data. Third, other OS features that also utilize page sharing cannot be supported. For example, process forking cannot utilize the copy-on-write optimization because the page in the parent and child are encrypted differently. This also holds true for shared libraries. This lack of IPC support is especially problematic in the era of CMPs. Finally, storage is required for virtual addresses at the lowest level on-chip cache, which is typically physically indexed and tagged.

The root cause of problems when using address in seed composition is that address is used as a *fundamental component of memory management*. Using address also as a basis for *security* intermingles security and memory management in undesirable ways.

Memory Integrity Verification. Recently proposed memory integrity verification schemes for secure processors have leveraged a variety of techniques [4, 9, 14, 17, 20, 22, 23, 24]. However, the security of Merkle Tree-based schemes [4] has been shown to be stronger than other schemes because every block read from memory is verified individually (as opposed to [23]), and *data replay* attacks can be detected in addition to spoofing and splicing attacks, which are detectable by simply associating a single MAC per data block [14]. In Merkle Tree memory integrity verification, a tree of MAC values is built over the memory. The root of this tree never goes off-chip, as a special on-chip register is used to hold its current value. When a memory block is fetched, its integrity can be checked by verifying its chain of MAC values up to the root MAC. Since the on-chip root MAC contains information about every block in the physical memory, an attacker cannot modify or replay any value in memory.

Despite its strong security, Merkle Tree integrity verification suffers from two significant issues. First, since a Merkle Tree built over the main memory computes MACs on memory events (cache misses and writebacks) generated by the processor, it covers the physical memory, but not swap memory which resides on disk. Hence, although Merkle Tree schemes can prevent attacks against values read from memory, there is no protection for data brought into memory from the disk. This is a significant security vulnerability since by tampering with swap memory on disk, attackers can indirectly tamper with main memory. One option would be to entrust the OS to protect pages swapped to and from the disk, however as with memory encryption it requires the assumption of a trusted OS. Another option, as discussed in [22], is to associate one Merkle Tree and on-chip secure root per process. However, managing multiple Merkle Trees results in extra on-chip storage and complexity.

Another significant problem is the storage overhead of internal Merkle Tree nodes in both the on-chip cache and main memory. To avoid repeated computation of internal Merkle Tree nodes as blocks are read from memory, a popular optimization lets recently accessed internal Merkle Tree nodes be cached on-chip. Using this optimization, the verification of a memory block only needs to proceed up the tree until the first cached node is found. Thus, it is not necessary to fetch and verify all Merkle Tree nodes up to the root on each memory access, significantly improving memory bandwidth consumption and verification performance. However, our results show that Merkle Tree nodes can occupy as much as 50% of the total L2 cache space, which causes the application to suffer from a large number of cache capacity misses.

Contributions. In this paper, we investigate system-level issues in secure processors, and propose mechanisms to address these issues that are simple yet effective. Our first contribution is *Address Independent Seed Encryption* (AISE), which decouples security and memory management by composing seeds using *logical identifiers* instead of virtual or physical addresses. The logical identifier of a block is the concatenation of a *logical page identifier* with the page offset of the block. Each page has a logical page identifier which is distinct across the entire memory and over the lifetime of the system. It is assigned to the page the first time the page is allocated or when it is loaded from disk. AISE provides better security since it provides complete seed/pad uniqueness for *every* block in the system (both in the physical and swap memory). At the same time, it also easily supports virtual memory and shared-memory based IPC mechanisms, and simplifies page swap mechanisms by not requiring decryption and re-encryption on a page swap.

The second contribution of this paper is a novel and efficient extension to Merkle Tree based memory integrity verification that allows extending the Merkle Tree to protect off-chip data (i.e. both physical and swap memory) with a single Merkle Tree and secure root MAC over the physical memory. Essentially, our approach allows pages in the swap memory to be incorporated into the Merkle Tree so that they can be verified when they are reloaded into memory.

Finally, we propose Bonsai Merkle Trees (BMTs), a novel organization of the Merkle Tree that naturally leverages counter-mode encryption to reduce its memory storage and performance overheads. We observe that if each data block has a MAC value computed over the data and its counter, a replay attack must attempt to replay an old data, MAC, and counter value together. A Merkle Tree built over the memory is able to detect any changes to the data MAC, which prevents any undetected changes to counter values or data. Our key insight is that: (1) there are many more MACs of data than MACs of counters, since counters are much smaller than data blocks, (2) a Merkle Tree that protects counters prevents any undetected counter modification, (3) if counter modification is thus prevented, the Merkle Tree does not need to be built over data MACs, and (4) the Merkle Tree over counters is *much* smaller and significantly shallower than the one over data. As a result, we can build such a Bonsai Merkle Tree over the counters which prevents data replay attacks using a much smaller tree for less memory storage overhead, fewer MACs to cache, and a better worst-case scenario if we miss on all levels of the tree up to the root. As our results show, BMT memory integrity verification reduces the performance overhead significantly, from 12.1% to 1.8% across all SPEC 2000 benchmarks [21], along with reducing the storage overhead in memory from 33.5% to 21.5%.

In the remainder of this paper, we discuss related work in section 2. Section 3 describes our assumed attack model. Section 4 describes our proposed encryption technique while section 5 describes our proposed integrity verification techniques in detail. Section 6 shows our experimental setup, and section 7 discusses our results and findings. Finally, section 8 summarizes our main contributions and results.

2 Related Work

Research on secure processor architectures [4, 5, 9, 13, 14, 16, 17, 18, 19, 20, 22, 23, 24, 25, 26] consists of memory encryption for ensuring data *privacy* and memory integrity verification for ensuring data *integrity*. Early memory encryption schemes utilized *direct* encryption modes [5, 13, 14, 22], in which a block cipher such as AES [2] is applied directly on a memory block to generate the plaintext or ciphertext when the block is read from or written to memory. Since, on a cache miss for a block, the block must first be fetched on chip before it can be decrypted, the long latency of decryption is added directly to the memory fetch latency, resulting in execution time overheads of up to 35% (almost 17% on average) [25]. In addition, there is a security concern for using direct encryption because different blocks having the same data value would result in the same encrypted value (ciphertext). This property implies that the statistical distribution of plaintext values matches the statistical distribution of ciphertext values, and may be exploited by attackers.

As a result of these concerns, recent studies have leveraged *counter-mode* encryption techniques [16, 18, 19, 20, 23, 24, 25, 26]. Counter-mode encryption overlaps decryption and memory fetch by decoupling them. This decoupling is achieved by applying a block cipher to a *seed* value to generate a cryptographic *pad*. The actual encryption or decryption is performed through an XOR of the plaintext or ciphertext with this pad. The security of counter-mode depends on the guarantee that each pad value (and thus each seed) is only used once. Consequently, a block's seed is typically constructed by concatenating the address of the block with a per-block counter value which is incremented each time the block is encrypted [19, 23, 24, 25]. If the seed components are available on chip at cache miss time, decryption can be started while the block is fetched from memory. Per-block counters can be cached on chip [23, 24, 25] or predicted [19].

Several different approaches have previously been studied for memory integrity verification in secure processors. These approaches include a MAC-based scheme where a MAC is computed and stored with each memory block when the processor writes to memory, and the MAC is verified when the processor reads from memory [14]. In [23], a Log Hash scheme was proposed where the overhead of memory integrity verification is reduced by checking the integrity of a series of values read from memory at periodic intervals during a program's execution using incremental, multiset hash functions. Merkle Tree based schemes have also been proposed where a tree of MAC values is stored over the physical memory [4]. The root of the tree, which stores information about every block in memory, is kept in a secure register on-chip. Merkle Tree integrity verification is often preferable over other schemes because of its security strength. In addition to spoofing and splicing attacks, *replay* attacks can also be prevented. We note that the Log Hash scheme can also prevent replay attacks, but as shown in [20], the long time intervals between integrity checks can leave the system open to attack.

The proposed scheme in this study differs from prior studies in the following ways. Our memory encryption avoids intermingling security with memory management by using *logical identifiers* (rather than address) as seed components. Our memory integrity verification scheme extends Merkle Tree protection to the disk in a novel way, and our BMT scheme significantly reduces the Merkle Tree size. The implications of this design will be discussed in detail in the following sections.

3. Attack Model and Assumptions

As in prior studies on hardware-based memory encryption and integrity verification, our attack model identifies two regions of a system. The secure region consists of the processor chip itself. Any code or data on-chip (e.g. in registers or caches) is considered safe and cannot be observed or manipulated by attackers. The non-secure region includes *all* off-chip resources, primarily including the memory bus, physical memory, and the swap memory in the disk. We do not constrain attackers' ability to attack code or data in these resources, so they can observe any values in the physical and swap memory and on all off-chip interconnects. Attackers can also act as a man-in-the-middle to modify values in the physical and swap memory and on all off-chip interconnects.

Note that memory encryption and integrity verification cover code and data stored in the main memory and communicated over the data bus. Information leakage through the address bus is not protected, but separate protection for the address bus such as proposed in [3, 27, 28] can be employed in conjunction with our scheme.

We assume that a proper infrastructure is in place for secure applications to be distributed to end users for use on secure processors. Finally, we also assume that the secure processor is executing applications in the *steady state*. More specifically, we assume that the secure processor already contains the cryptographic keys and code necessary to load a secure application, verify its digital signature, and compute the Merkle Tree over the application in memory.

4. Memory Encryption

4.1. Overview of Counter-Mode Encryption

The goal of memory encryption is to ensure that all data and code stored outside the secure processor boundary is in an unintelligible form, not revealing anything about the actual values stored. Figure 1 illustrates how this is achieved in counter-mode encryption. When a block is being written back to memory, a *seed* is encrypted using a block cipher (e.g. AES) and a *secret key*, known only to the processor. The encrypted seed is called a cryptographic *pad*, and this pad is combined with the plaintext block via a bitwise XOR operation to generate the ciphertext of the block before the block can be written to memory. Likewise, when a ciphertext block is fetched from memory, the same seed is encrypted to generate the same pad that was used to encrypt the block. When the block arrives on-chip, another bitwise XOR with the pad restores the block to its original plaintext form. Mathematically, if P is the plaintext, C is the ciphertext, E is the block cipher function, and K is the secret key, the encryption performs $C = P \oplus E_K(Seed)$. By XORing both sides with $E_K(Seed)$, the decryption yields the plaintext $P = C \oplus E_K(Seed)$.

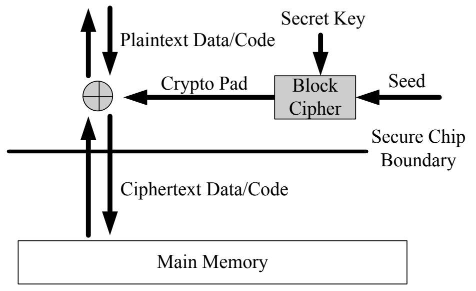


Figure 1. Counter-mode based memory encryption.

The *security* of counter-mode encryption relies on ensuring that the cryptographic pad (and hence the seed) is unique each time a block is encrypted. The reason for this is that suppose two blocks having plaintexts P_1 and P_2, and ciphertext C_1 and C_2, have the same seeds, that is $Seed_1 = Seed_2$. Since the block cipher function has a one-to-one mapping, then their pads are also the same, i.e. $E_K(Seed_1) = E_K(Seed_2)$. By XORing both sides of $C_1 = P_1 \oplus E_K(Seed_1)$ and $C_2 = P_2 \oplus E_K(Seed_2)$, we obtain the relationship of $C_1 \oplus C_2 = P_1 \oplus P_2$, which means that if any three variables are known, the other can be known, too. Since ciphertexts are known by the attacker, if one plaintext is known or can be guessed, then the other plaintext can be obtained. Therefore, the security requirement for seeds is that they must be *globally unique*, both spatially (across blocks) and temporally (versions of the same block over time).

The *performance* of counter-mode encryption depends on whether the seed of a code/data block that misses in the cache is available at the time the cache miss is determined. If the seed is known by the processor at the time of a cache miss, the pad for the code/data block can be generated in parallel with the off-chip data fetch, hiding the overhead of memory encryption.

Two methods to achieve the global uniqueness of seeds have been studied. The first is to use a *global counter* as the seed for all blocks in the physical memory. This global counter is incremented each time a block is written back to memory. The global counter approach avoids the use of address as a seed component. However, when the counter reaches its maximum value for its size, it will wrap around and start to reuse its old values. To provide seed uniqueness over time, counter values cannot be reused. Hence, when the counter reaches its maximum, the secret key must be changed, and the *entire physical memory* along with the *swap memory* must be decrypted with the old key and re-encrypted with the new secret key. This re-encryption is very costly and frequent for the global counter approach [24], and can only be avoided by using a large global counter, such as 64 bits. Unfortunately, large counters require a large on-chip *counter cache* storage in order to achieve a good hit rate and overlap decryption with code/data fetch. If the counter for a missed code/data cache block is not found in the counter cache, it must first be fetched from memory along with fetching the code/data cache block. Decryption cannot begin until the counter fetch is complete, which exposes decryption latency and results in poor performance.

To avoid the fast growth of global counters which leads to frequent memory re-encryption, prior studies use per-block counters [19, 23, 24, 25], which are incremented each time the corresponding block is written back to memory. Since each block has its own counter, the counter increases at an orders-of-magnitude slower rate compared to the global counter approach. To provide seed uniqueness across different blocks, the seed is composed by concatenating the per-block counter, the block address, and chunk id [1]. This seed choice also meets the performance criterion since block addresses can be known at cache miss time, and studies have shown that frequently needed block counters can be effectively cached on-chip [23, 24, 25] or predicted [19] at cache miss time.

However, this choice for seed composition has several significant disadvantages due to the fact that block address, which was designed as an underlying component of memory management, is now being used as a component of security. Because of this conflict between the intended use of addresses and their function in a memory encryption scheme, many problems arise for a secure processor when block address (virtual or physical) is used as a seed component.

4.2. Problems with Current Counter-Mode Memory Encryption

Most general purpose computer systems today employ virtual memory, illustrated in Figure 2. In a system with virtual memory, the system gives an abstraction that each process can potentially use all addresses in its virtual address space. A paging mechanism is used to translate virtual page addresses (that a process sees) to physical page addresses (that actually reside in the physical and swap memory). The paging mechanism provides *process isolation* by mapping the same page address of different processes to different physical pages (circle (2)), and *sharing* by mapping virtual pages

[1] A *chunk* refers to the unit of encryption/decryption in a block cipher, such as 128 bits (16 bytes). A cache or memory block of 64 bytes contains four chunks. Seed uniqueness must hold across chunks, hence the *chunk id*, referring to which chunk being encrypted in a block, is included as a component of the seed.

of different processes to the same physical page (circle (1)). The paging mechanism often extends the physical memory to the *swap memory* area in disks in order to manage more pages. The swap memory holds pages that are not expected to be used soon (circle (3)). When a page in the swap memory is needed, it is brought in to the physical memory, while an existing page in the physical memory is selected to be replaced into the swap memory.

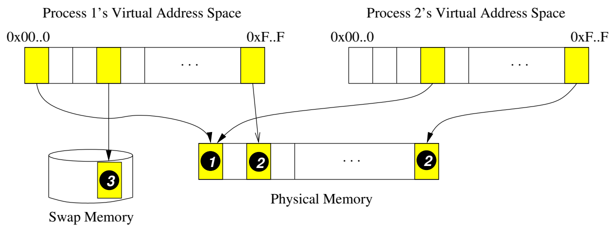

Process 1's Virtual Address Space
Process 2's Virtual Address Space
0x00..0
0xF..F
0x00..0
0xF..F
Swap Memory
Physical Memory

Figure 2. Virtual Memory management allows virtual pages of different processes to map to a common physical page for sharing purpose (1), the same virtual pages in different processes to map to different physical pages (2), and some virtual pages to reside in the swap memory in the disk (3).

The use of physical address in the seed causes the following *complexity* and possible *security* problems. The mapping of a virtual page of a process to a physical frame may change dynamically during execution due to page swaps. Since the physical address changes, the entire page must be first decrypted using the old physical addresses and then re-encrypted using the new physical addresses on a page swap. In addition, pages encrypted based on physical address cannot be simply swapped to disk or pad reuse may occur between blocks in the swapped out page and blocks located in the page's old location in physical memory. This leaves an open problem as to how to protect pages on disk.

The use of virtual address has its own set of critical problems. Seeds based on virtual address are vulnerable to pad reuse since different processes use the same virtual addresses and could easily use the same counter values. Adding process ID to the seed solves this problem, but creates a new set of system-level issues. First, process IDs can now no longer be reused by the OS, and current OSes have a limit on the range of possible process IDs. Second, shared-memory IPC mechanisms cannot be used. Consider that a single physical page may be mapped into multiple virtual pages in either a single process or in multiple processes. Since each virtual page will see its own process ID and virtual address combination, the seeds will be different and will produce different encryption and decryption results. Consequently, mmap/munmap (based on shared-memory) cannot be supported, and these are used extensively in glibc for file I/O and memory management, especially for implementing threads. This is a critical limitation for secure processors, especially in the age of CMPs. Third, other OS features that also utilize page sharing cannot be supported. For example, process forking cannot utilize the copy-on-write optimization because the page in the parent and child are encrypted differently. This also holds true for shared libraries. Finally, since virtual addresses are often not available beyond the L1 cache, extra storage may be required for virtual addresses at the lowest level on-chip cache.

One may attempt to augment counter-mode encryption with special mechanisms to deal with paging or IPC. Unfortunately, they would likely result in great complexity. For example, when physical address is used, to avoid seed/pad reuse in the swap memory, an authentic, secure OS running on the secure processor could encrypt and decrypt swapped pages in software. However this solution is likely not desirable since it makes the secure processor's hardware-based security mechanisms contingent on a secure and uncompromised OS. OS vulnerabilities may be exploited in software by attackers to subvert the secure processor. Alternatively, we could rely on hardware to re-encrypt swapped pages, however this solution has its own set of problems. First, this requires supporting two encryption methods in hardware. A page that is swapped out must first be decrypted (using counter mode) and then encrypted (using direct mode) before it is placed in the swap memory, while the reverse must occur when a page is brought from the disk to the physical memory. Second, there is the issue of who can request the page re-encryptions, and how these requests are made, which requires an extra authentication mechanism. Another example, when virtual address is used, is that shared memory IPC and copy-on-write may be enabled by encrypting all shared pages with direct encryption, while encrypting everything else with counter-mode encryption. However, this also complicates OS handling of IPC and copy-on-write, and at the same time complicates the hardware since it must now support two modes of encryption. Therefore, it is arguably better to identify and deal with the root cause of the problem: address is used as a *fundamental component of memory management*, and using the address also as a basis for *security* intermingles security and memory management in undesirable ways.

4.3. Address Independent Seed Encryption

In light of the problems caused by using address as a seed component, we propose a new seed composition mechanism which we call *Address-Independent Seed Encryption (AISE)*, that is free from the problems of address-based seeds. The key insight is that rather than using addresses as a seed component alongside a counter, we use *logical identifiers* instead. These logical identifiers are truly unique across the entire physical and swap memory and over time.

Conceptually, each block in memory must be assigned its own logical identifier. However, managing and storing logical identifiers for the entire memory would be quite complex and costly (similar to global counters). Fortunately, virtual memory management works on the granularity of pages (usually 4 Kbytes) rather than words or blocks. Any block in memory has two components: page address which is the unit of virtual memory management, and page offset. Hence, it is sufficient to assign logical identifiers to pages, rather than to blocks. Thus, for each chunk in the memory, its seed is the concatenation of a *Logical Page IDentifier* (LPID), the page offset of the chunk's block, the block's counter value, and the chunk id.

To ensure complete uniqueness of seeds across the physical and swap memory and over time, the LPID is chosen to be a *unique value* assigned to a page when it is *first allocated* by the system. The LPID is unique for that page across the system lifetime, and never changes over time. The unique value is obtained from an on-chip counter called the *Global Page Counter* (GPC). Once a value of the GPC is assigned to a new page, it is incremented. To provide true uniqueness over time, the GPC is stored in a *non-volatile* register on chip. Thus, even across system reboots, hibernation, or power optimizations that cut power off to the processor, the GPC retains its value. Rebooting the system does not cause the counter to reset and start reusing seeds that have been used in the past boot. The GPC is also chosen to be large (64 bits), so that it does not overflow for millenia, easily exceeding the lifetime of the system.

One may have concerns for how the LPID scheme can be used in systems that support multiple page sizes, such as when super pages (e.g. 16 MBs) are used. However, the number of page offset bits for a large page always exceeds the number of page offset bits for a smaller page. Hence, if we choose the LPID portion of the seed to have as many bits as needed for the smallest page size supported in the system, the LPID still covers the unit of virtual memory management (although sometimes unnecessarily covering some page offset bits) and provides seed uniqueness for the system.

The next issue we address is how to organize the storage of LPIDs of pages in the system. One alternative is to add a field for the LPID in page table entries and TLB entries. However, this approach significantly increases the page table and TLB size, which is detrimental to system performance. Additionally, the LPID is only needed for accesses to off-chip memory, while TLBs are accessed on each memory reference. Another alternative would be to store the LPIDs in a dedicated portion of the physical memory. However this solution also impacts performance since a memory access now must fetch the block's counter and LPID in addition to the data, thus increasing bandwidth usage. Consequently, we choose to co-store LPIDs and counters, by taking an idea from the split counter organization [24]. We associate each counter block with a page in the system, and each counter block contains one LPID and all block counters for a page.

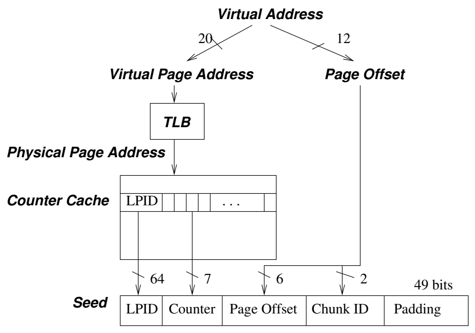


Figure 3. Organization of logical page identifiers.

Figure 3 illustrates the organization, assuming 32-bit virtual addresses, a 4-Kbyte page size, 64-byte blocks, 64-bit LPID, and a 7-bit counter per block. A virtual address is split into the high 20-bit virtual page address and 12-bit page offset. The virtual page address is translated into the physical address, which is used to index a counter cache. Each counter block stores the 64-bit LPID of a page, and 64 7-bit counters where each counter corresponds to one of the 64 blocks in a page. If the counter block is found, the LPID and one counter are used for constructing the seed for the address, together with the 8 high order bits of the page offset (6-bit block offset and 2-bit chunk id). Padding is added to make the seed 128 bits, which corresponds to the chunk size in the block cipher. Note that the LPID and counter block can be found using simple indexing for a given physical address.

In contrast to using two levels of counters in [24], we only use small per-block (minor) counters. We eliminate the major counter and use the LPID instead. If one of the minor counter overflows, we need to avoid seed reuse. To achieve that, we assign a new LPID for that page by looking up the GPC, and re-encrypt only that page. Hence, the LPID of a page is no longer static. Rather, a new unique value is assigned to a page when a page is first allocated *and* when a page is re-encrypted.

4.4. Dealing with Swap Memory and Page Swapping

In our scheme, no two pages share the same LPID and hence seed uniqueness is guaranteed across the physical and swap memory. In addition, once a unique LPID value is assigned to a page, it does not change until the page needs to be re-encrypted. Hence, when a page is swapped out to the disk, it retains a unique LPID and does not need to be re-encrypted or specially handled. The virtual memory manager can just move a page from the physical memory to the swap memory together with swapping its LPID and block of counters.

When an application suffers a page fault, the virtual memory manager locates the page and its block of counters in the disk, then brings it into the physical memory. The block of counters (including LPID) are placed at the appropriate physical address in order for the block to be directly indexable and storable by the counter cache. Therefore, the only special mechanism that needs to be added to the page swapping mechanism is proper handling of the page's counter blocks. Since no re-encryption is needed, moving the page in and out of the disk can be accomplished with or without the involvement of the processor (e.g. we could use DMA).

4.5. Dealing with Page Sharing

Page sharing is problematic to support if virtual address is used as a seed component, since different processes may try to encrypt or decrypt the same page with different virtual addresses. With our LPID scheme, the LPID is unique for each page and can be directly looked up using the physical address. Therefore, all page sharing uses can naturally be facilitated without any special mechanisms.

4.6. Advantages of AISE

Our AISE scheme satisfies the security and performance criteria for counter-mode encryption seeds, while naturally supporting virtual memory management features and IPC without much complexity. The LPID portion of the seed ensures that the blocks in every page, both in the physical memory and on disk are encrypted with different pads. The page offset portion of the seed ensures that each block within a page is encrypted with a different pad. The block counter portion of the seed ensures that the pad is unique each time a single block is encrypted. Finally, since the global page counter is stored in non-volatile storage on chip, the pad uniqueness extends across system boots.

From a performance perspective, AISE does not impose any additional storage or runtime overheads over prior counter-mode encryption schemes. AISE allows seeds to be composed at cache miss time since both the LPID and counter of a block are co-stored in memory and cached together on-chip. Storage overhead is equivalent to the already-efficient split counter organization, since LPID replaces the major counter of the split counter organization and does not add extra storage. On average, a 4 Kbyte page only requires 64 bytes of storage for the LPID and counters, representing a 1.6% overhead. Similar to the split counter organization, AISE does not incur entire-memory re-encryption when a block counter overflows. Rather, it only incurs re-encryption of a page when overflow occurs.

From a complexity perspective, AISE allows pages to be swapped in and out of the physical memory without involving page re-encryption (unlike using physical address), while allowing all types of IPC and page sharing (unlike using virtual address).

To summarize, memory encryption using our AISE technique retains all of the latency-hiding ability as proposed in prior schemes, while eliminating the significant problems that arise from including address as a component of the cryptographic seed.

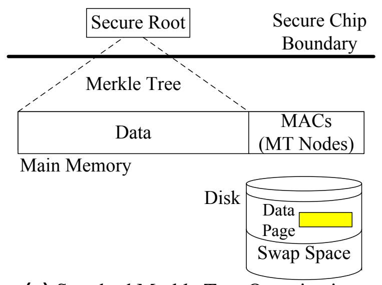
5. Memory Integrity Verification

The goal of a memory integrity verification scheme is to ensure that a value loaded from some location by a processor is *equal to* the *most recent* value that the processor last wrote to that location. There are three types of attacks that may be attempted by an attacker on a value at a particular location. Attackers can replace the value directly (*spoofing*), exchange the value with another value from a different location (*splicing*), and replay an old value from the same location (*replay*). As discussed in XOM [5], if for each memory block a MAC is computed using the value and address as its input, spoofing and splicing attacks would be detectable. However, replay attacks can be successfully performed by rolling back both the value and its MAC to their older versions. To detect replay attacks, Merkle Tree verification has been proposed [4]. A Merkle Tree keeps hierarchical MACs organized as a tree, in which a parent MAC protects multiple child MACs. The root of the tree is stored on-chip at all times so that it cannot be tampered by attackers. When a memory block is fetched, its integrity can be verified by checking its chain of MAC values up to the root MAC. When a cache block is written back to memory, the corresponding MAC values of the tree are updated. Since the on-chip MAC root contains information about every block in the physical memory, an attacker cannot modify or replay any value in the physical memory.

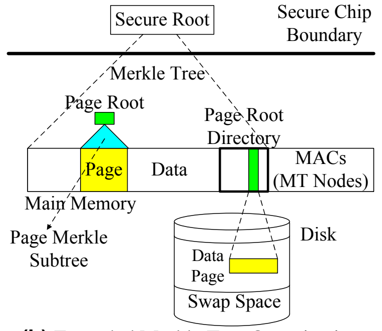
5.1. Extended Merkle Tree Protection

Previously proposed Merkle Tree schemes which only cover the physical memory, as shown in Figure 4(a), compute MACs on memory events (cache misses and write backs) generated by the processor. However, I/O transfer between the physical memory and swap memory is performed by an I/O device or DMA and is not visible to the processor. Consequently, the standard Merkle Tree protection only covers the physical memory but not the swap memory. This is a significant security vulnerability since by tampering with the swap memory in the disk, attackers can indirectly tamper with the main memory. We note that it would be possible to entrust a secure OS with the job of protecting pages swapped to and from the disk in software. However, this solution requires the assumption of a secure and untampered OS which may not be desirable. Also, as discussed in [22], it would be possible to compute the Merkle Tree over the virtual address space of each process to protect the process in both the memory and the disk. However this solution would require one Merkle Tree and on-chip secure root MAC per process, which results in extra on-chip storage for the root MACs and complexity in managing multiple Merkle Trees.

This security issue clearly motivates the need to extend the Merkle Tree protection to all off-chip data both in the physical and swap memory, as illustrated in Figure 4(b). To help explain our solution, we define two terms: *Page Merkle Subtree* and *page root*. A Page Merkle Subtree is simply the subset of all the MACs of the Merkle Tree which directly cover a particular page in memory. A page root is the top-most MAC of the Page Merkle Subtree. Note that the Page Merkle Subtree and page root are simply MAC values which make up a portion of the larger Merkle Tree over the entire physical memory.



(a) Standard Merkle Tree Organization



(b) Extended Merkle Tree Organization

Figure 4. Our novel Merkle Tree organization for extending protection to the swap memory in disk.

To extend Merkle Tree protection to the swap memory, we make two important observations. First, for each page, its page root is sufficient to verify the integrity of all values on the page. The internal nodes of the Page Merkle Subtree can be re-computed and verified as valid by comparing the computed page root with the stored, valid page root. Secondly, the physical memory is covered entirely by the Merkle Tree and hence it provides secure storage. From these two observations, we can conclude that as long as the page roots of all swap memory pages are stored in the physical memory, then the entire swap memory integrity can be guaranteed. To achieve this protection, we dedicate a small portion of the physical memory to store page root MACs for pages currently on disk, which we refer to as the *Page Root Directory*. Note that while our scheme requires a small amount of extra storage in main memory for the page root directory, the on-chip Merkle Tree operations remain the same and a single on-chip MAC root is still all we require to maintain the integrity of the entire tree. Furthermore, as shown in Figure 4(b), the page root directory itself is protected by the Merkle Tree.

To illustrate how our solution operates, consider the following example. Suppose that the system wants to load a page B from swap memory into physical memory currently occupied by a page A. The integrity verification proceeds as follows. First, the page root of B is looked up from the page root directory and brought on chip. Since this lookup is performed using a regular processor read, the integrity of the page root of B is automatically verified by the Merkle Tree. Second, page A is swapped out to the disk and its page root is installed at the page root directory. This installation updates the part of the Merkle Tree that covers the directory, protecting the page root of A from tampering. Third, the Page Merkle Subtree of A is invalidated from on-chip caches in order to force future integrity verification for the physical frame where A resided. Next, the page root of B is installed in the proper location as part of the Merkle Tree, and the Merkle Tree is updated accordingly. Finally, the data of page B can be loaded into the physical frame. When any value in B is loaded by the processor, the integrity checking will take place automatically by verifying data against the Merkle Tree nodes at least up to the already-verified page root of B.

5.2. Bonsai Merkle Trees

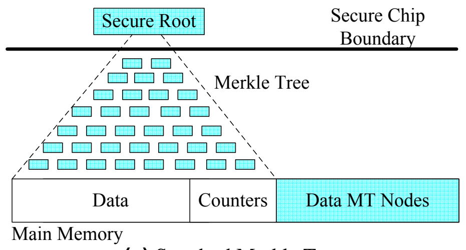
For our final contribution, we introduce Bonsai Merkle Trees (BMTs), a novel Merkle Tree organization designed to significantly reduce their performance overhead for memory integrity verification. To motivate the need for our BMT approach, we note a common optimization that has been studied for Merkle Tree verification is to cache recently accessed and verified MAC values on chip [4]. This allows the integrity verification of a data block to complete as soon as a needed MAC value is found cached on-chip. The reason being, since this MAC value has previously been verified and is safe on-chip, it can be trusted as if it were the root of the tree. The resulting reduction in memory bandwidth consumption significantly improves performance compared to fetching MAC values up to the tree root on every data access. However, the sharing of on-chip cache between data blocks and MAC values can significantly reduce the amount of available cache space for data blocks. In fact, our experiments show that for memory-intensive applications, up to 50% of a 1MB L2 cache can be consumed by MAC values during application execution, severely degrading performance. It is likely that MACs occupy such a large percentage of cache space because MACs in upper levels of a Merkle Tree have high temporal locality when the verification is repeated due to accesses to the data blocks that the MAC covers.

Before we describe our BMT approach, we motivate it from a security perspective. BMTs exploit certain security properties that arise when Merkle Tree integrity verification is used in conjunction with counter-mode memory encryption. We make two observations. First, the Merkle Tree is designed to prevent data replay attacks. Other types of attacks such as data spoofing and splicing can be detected simply by associating a single MAC value with each data block. Second, in most proposed memory encryption techniques using counter-mode, each memory block is associated with its own counter value in memory [18, 19, 23, 24, 25]. Since a block's counter value is incremented each time a block is written to memory, the counter can be thought of as a *version* number for the block. Based on these observations, we make the following claim:

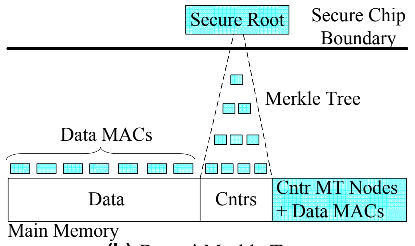
In a system with counter-mode encryption and Merkle Tree memory integrity verification, data values do not need to be protected by the Merkle Tree as long as (1) each block is protected by its own MAC, computed using a keyed hashing function (e.g. HMAC based on SHA-1), (2) the block's MAC includes the counter value and address of the block, and (3) the integrity of all counter values is guaranteed.

To support this claim, we provide the following argument. Let us denote the plaintext and ciphertext of a block of data as P and C, its counter value as ctr, the MAC for the block as M, and the secret key for the hash function as K. The MAC of a block is computed using a keyed cryptographic hash function H with the ciphertext and counter as its input, i.e. $M = H_K(C, ctr)$. Integrity verification computes the MAC and compares it against the MAC that was computed in the past and stored in the memory. If they do not match, integrity verification fails. Since the integrity of the counter value is guaranteed (a requirement in the claim), attackers cannot tamper with ctr without being detected. They can only tamper with C to produce C', and/or the stored MAC to produce to produce M'. However, since the attacker does not know the secret key of the hash function, they cannot produce a M' to match a chosen C'. In addition, due to the non-invertibility property of a cryptographic hash function, they cannot produce a C' to match a chosen M'. Hence, $M' \neq H_K(C', ctr)$. Since, during integrity verification, the computed MAC is $H_K(C', ctr)$, while the stored one is M', integrity verification will fail and the attack detected. In addition, attackers cannot replay both C and M to their older version because the old version satisfies $M^{old} = H_K(C^{old}, ctr^{old})$, while the integrity verification will compute the MAC using the fresh counter value whose integrity is assumed to be guaranteed ($H_K(C^{old}, ctr)$), which is not equal to $H_K(C^{old}, ctr^{old})$. Hence replay attacks would also be detected.

The claim is significant because it implies that we only need the Merkle Tree to cover counter blocks, but not code or data blocks. Since counters are a lot smaller than data (a ratio of 1:64 for 8-bit counters and 64-byte blocks), the Merkle Tree to cover the block counters is *substantially* smaller than the Merkle Tree for data. Figure 5(a) shows the traditional Merkle Tree which covers all data blocks, while Figure 5(b) shows our BMT that only covers counters, while data blocks are now only covered by their MACs.



(a) Standard Merkle Tree



(b) Bonsai Merkle Tree

Figure 5. Reduction in size of Bonsai Merkle Trees compared to standard Merkle Trees.

Since the size of the Merkle Tree is significantly reduced, and since each node of the Merkle Tree covers more data blocks, the amount of on-chip cache space required to store frequently accessed Bonsai Merkle Tree nodes is significantly reduced. To further reduce the cache footprint, we do not cache data block MACs. Since each data block MAC only covers four data blocks, it has a low degree of temporal reuse compared to upper level MACs in a standard

Merkle Tree. Hence, it makes sense to only cache Bonsai Merkle Tree nodes but not data block MACs, as we will show in Section 7.

Overall, BMTs achieve the same security protection as in previous schemes where a Merkle Tree is used to cover the data in memory (i.e. data spoofing, splicing, and replay protection), but with much less overhead. Also note that BMTs easily combine with our technique to extend Merkle Tree based protection to the disk. When a page of data is swapped out to disk, the counters must always be swapped out and stored as well. Therefore we simply keep a portion of memory to store the page roots for *Bonsai Page Merkle Subtrees* on the disk as described in the previous section.

6 Experimental Setup

We use SESC [10], an open source execution driven simulator, to evaluate the performance of our proposed memory encryption and integrity verification approaches. We model a 2GHz, 3-issue, out-of-order processor with split L1 data and instruction caches. Both caches have a 32KB size, 2-way set associativity, and 2-cycle round-trip hit latency. The L2 cache is unified and has a 1MB size, 8-way set associativity, and 10-cycle round-trip hit latency. For counter mode encryption, the processor includes a 32KB, 16-way set-associative counter cache at the L2 cache level. All caches have 64B blocks and use LRU replacement. We assume a 1GB main memory with an access latency of 200 processor cycles. The encryption/decryption engine simulated is a 128-bit AES engine with a 16-stage pipeline and a total latency of 80 cycles, while the MAC computation models HMAC [6] based on SHA-1 [1] with 80-cycle latency [11]. Counters are composed of a 64-bit LPID concatenated with a 7-bit block counter. So a counter cache block contains one LPID value along with 64 block counters (enough for a 4KB memory page). The default authentication code size used is 128 bits.

We use 21 C/C++ SPEC2K benchmarks [21]. We only omit Fortran 90 benchmarks, which are not supported on our simulator infrastructure. For each simulation, we use the reference input set and simulate for 1 billion instructions after fast forwarding for 5 billion. In each figure, we show individual result for benchmarks that have L2 miss rates higher than 20%, but the average is calculated across all 21 benchmarks. In our experiments, we ignore the effect of page swaps as the overhead due to page swaps with our techniques is negligibly small.

Finally, for evaluation purpose, we use timely but non-precise integrity verification, i.e. each block is immediately verified as soon as it is brought on chip, but we do not delay the retirement of the instruction that brings the block on chip if verification is not completed yet. Note that all of our schemes (AISE and BMT) are compatible with both non-precise and precise integrity verification.

7 Evaluation

To evaluate our approach, we first present a qualitative comparison of AISE against other counter-mode encryption approaches. Then we present quantitative results and analysis of AISE+BMT compared to prior approaches.

7.1. AISE: Qualitative Evaluation

Table 1 qualitatively compares AISE with other counter-mode encryption approaches, in terms of IPC support, cryptographic latency hiding capability, storage overheads, and other miscellaneous overheads. The first scheme, *Global Counter*, was discussed in Section 4.1. Like AISE, this scheme supports all forms of IPC and requires no special mechanisms to protect swap memory. However, global counters need to be large (64 bits) to avoid overflow and frequent entire-memory re-encryption (32-bit counters cause entire memory re-encryption every few minutes [24]). Thus, these large counters cache poorly, and predicting counter values is difficult because the values are likely non-contiguous for a particular block over time, resulting in little latency-hiding opportunity (we evaluate this in Section 7.2). In addition, the memory storage overhead for using 64-bit counters per-block is high at 12.5%.

The next two configurations represent counter-mode encryption using either physical (*Counter (Phys Addr)*) or virtual address (*Counter (Virt Addr)*) plus per-block counters to compose seeds. As shown in the table, while AISE is amenable to all forms of IPC, including shared-memory, virtual address based schemes cannot support this popular type of communication. In addition, virtual address schemes require this address to be stored in the lowest level cache so that it can be readily accessed, and physical address schemes require page re-encryption on page swaps. Finally, while AISE will work well with proposed counter caching and prediction schemes and require only small storage overheads, virtual and physical address schemes depend on the chosen counter size.

7.2. Quantitative Evaluation

In our first experiment, we compare AISE+BMT to another memory encryption and integrity verification scheme which can provide the same type of system-level support as our approach (e.g. shared memory IPC, virtual memory support, etc.). Figure 6 shows these results of AISE+BMT compared to the 64-bit global counter scheme plus standard Merkle Tree protection (global64+MT), where the execution time overhead is shown normalized to a system with no protection. While the two schemes offer similar system level benefits, the performance benefit of our AISE+BMT scheme is tremendous. The average execution time overhead of global64+MT is 25.9% with a maximum of 151%, while the average for AISE+BMT is a mere 1.8% with a maximum of only 13%. This figure shows that our AISE+BMT approach overwhelmingly provides the best of both worlds in terms of support of system-level issues and performance overhead reduction, making it more suitable for use in real systems.

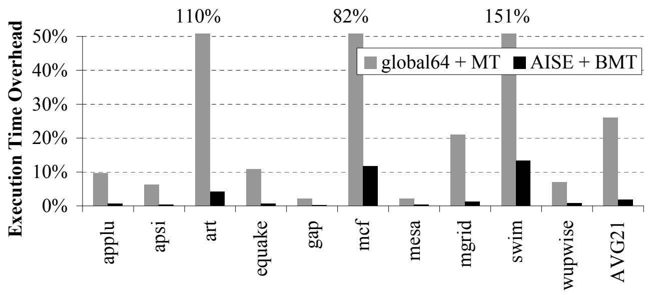


Figure 6. Performance overhead comparison of AISE with BMT vs. Global counter scheme with traditional Merkle Tree

To better understand the results from the previous figure, we next present figures which break the overhead into encryption vs. integrity verification components. Figure 7 shows the normalized execution time overhead of AISE compared to the global counter scheme with 32-bit and 64-bit counters (note that only encryption is being performed for this figure). As the figure shows, AISE by

Table 1. Qualitative comparison of AISE with other counter-mode encryption approaches

Encryption Approach	Global Counter	Counter (Phys Addr)	Counter (Virt Addr)	AISE
IPC Support	Yes	Yes	No shared-memory IPC	Yes
Latency Hiding	Caching: Poor, Prediction: Difficult	Depends on counter size	Depends on counter size	Good
Storage Overhead	High (64-bit: 12.5%)	Depends on counter size	Depends on counter size	Low (1.6%)
Other Issues	None	Re-enc on page swap	VA storage in L2	None

itself is significantly better from a performance perspective than the global counter scheme (1.6% average overhead vs. around 4% and 6% for 32 and 64-bit global counters). Recall also that 64-bit counters, which should be used to prevent frequent entire-memory re-encryptions [24], require a 12.5% memory storage overhead. Note that we do not show results for counter-mode encryption using address plus block counter seeds since the performance will be essentially equal to AISE if same-sized block counters are used. Since AISE supports important system level mechanisms not supported by address-based counter-mode schemes, and since the performance and storage overheads of AISE are superior to the global counter scheme, our AISE approach is an attractive memory encryption option for secure processors

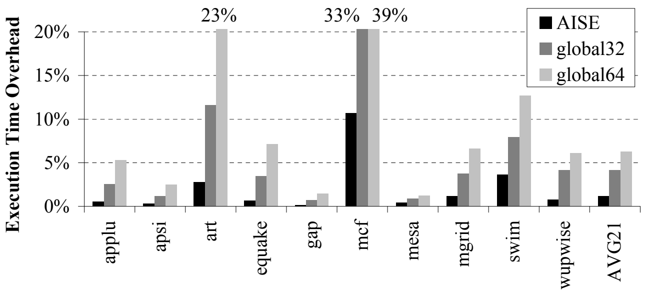


Figure 7. Performance overhead comparison of AISE versus the global counter scheme

To see the overhead due to integrity verification, Figure 8 shows the overhead of AISE only (the same as the AISE bar on the previous figure), AISE plus a standard Merkle Tree (AISE+MT), and AISE plus our BMT scheme (AISE+BMT). Note that we use AISE as the encryption scheme for all cases so that the extra overhead due to the different integrity verification schemes is evident. Our first observation is that integrity verification due to maintaining and verifying Merkle Tree nodes is the dominant source of performance overhead, which agrees with other studies [16, 24]. From this figure, it is also clear that our BMT approach outperforms the standard Merkle Tree scheme, reducing the overhead from 12.1% in AISE+MT to only 1.8% in AISE+BMT. Even for memory intensive applications such as art, mcf, and swim, the overhead using our BMT approach is less than 15% while it can be above 60% with the standard Merkle Tree scheme. Also, for every application except for swim, the extra overhead of AISE+BMT compared to AISE is negligible, indicating that our BMT approach removes almost all of the performance overhead of Merkle Tree-based memory integrity verification. We note that [24] also obtained low average overheads with their memory encryption and integrity verification approach, however for more memory-intensive workloads such as art, mcf, and swim, their performance overheads still approached 20% and they assumed a smaller, 64-bit MAC size. Since our BMT scheme retains the security strength of standard Merkle Tree schemes, the improved performance of BMTs is a significant advantage.

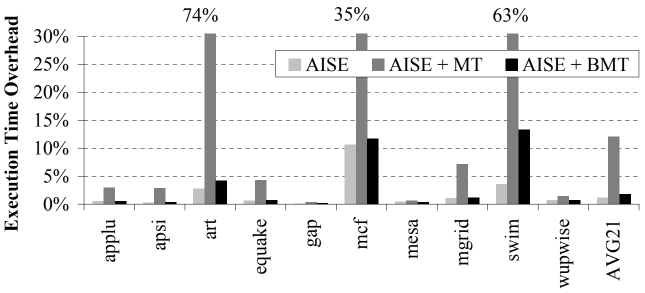


Figure 8. Performance overhead comparison of AISE with our Bonsai Merkle Tree vs. AISE with the Standard Merkle Tree

To understand why our BMT scheme can outperform the standard Merkle Tree scheme by such a significant amount, we next present some important supporting statistics. Figure 9 measures the amount of "cache pollution" in the L2 cache due to storing frequently accessed Merkle Tree nodes along with data. The bars in this figure show the *average* portion of L2 cache space that is occupied by data blocks during execution. For the standard Merkle Tree, we found that on average data occupies only 68% of the L2 cache, while the remaining 32% is occupied by Merkle Tree nodes. In extreme cases (e.g. art and swim), almost 50% of the cache space is occupied by Merkle Tree nodes. Note that for 128-bit MACs, the main memory storage overhead incurred by Merkle Tree nodes stands at 25%, so if the degree of temporal locality of Merkle Tree nodes is equal to data, then only 25% of the L2 cache should be occupied by Merkle Tree nodes. Thus it appears that Merkle Tree nodes have a higher degree of temporal locality than data. Intuitively, this observation makes sense because for each data block that is brought into the L2 cache, one or more Merkle Tree nodes will be touched for the purpose of verifying the integrity of the block. With our BMT approach, on the other hand, data occupies 98% of the L2 cache, which means that the remaining 2% of the L2 cache is occupied by Bonsai Merkle Tree nodes. This explains the small performance overheads of our AISE+BMT scheme. Since the ratio of the size of a counter to a data block is 1:64, the footprint of the BMT is very small, so as expected it occupies an almost negligible space in the L2 cache. Furthermore, since data block MACs are not cached, they do not take up L2 cache space.

Next, we look at the (local) L2 cache miss rate and bus utilization of the base unprotected system, the standard Merkle Tree, and our BMT scheme, shown in Figure 10. The figure shows that while the L2 cache miss rates and bus utilization increase significantly when the standard Merkle Tree scheme is used (average L2 miss rate from 37.8% to 47.5%, bus utilization from 14% to 24%), our BMT scheme only increases L2 miss rates and bus utilization slightly (average L2 miss rate from 37.8% to 38.5% and bus utilization from 14% to 16%). These results show that the impact of reduced cache pollution from Merkle Tree nodes results in a sizable

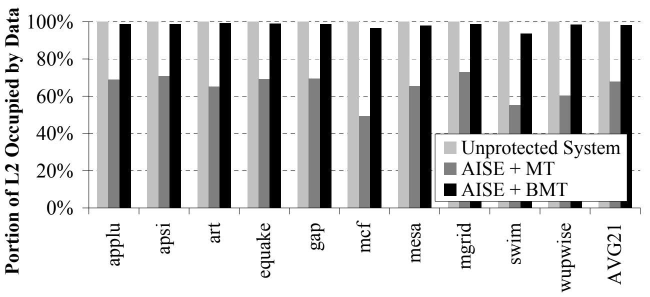
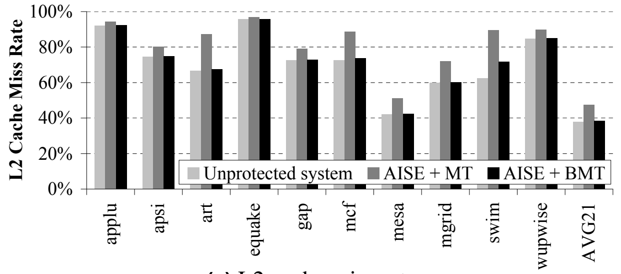
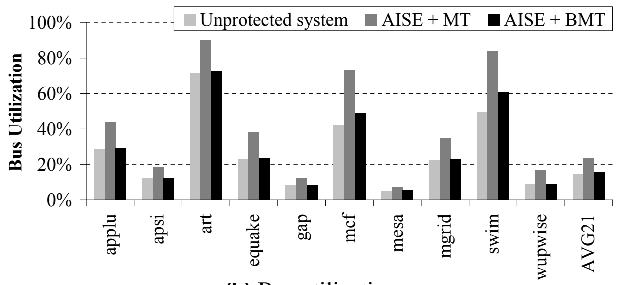


Figure 9. L2 cache pollution

reduction in L2 cache miss rates and bus utilization and thus the significant reduction of performance overheads seen in Figure 8.



(a) L2 cache miss rate

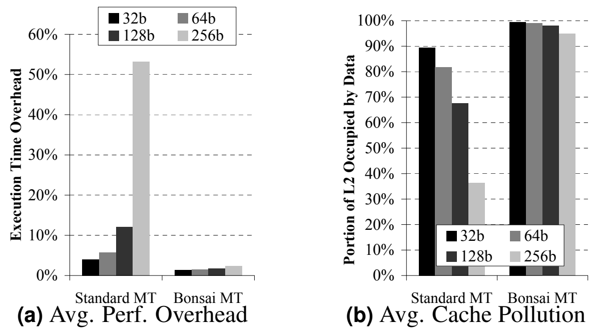


(b) Bus utilization

Figure 10. L2 cache miss rate and bus utilization of an unprotected system, standard Merkle Tree, and our BMT scheme

7.3. Sensitivity to MAC Size

In this section, we examine the sensitivity of the standard Merkle Tree (MT) and our BMT schemes to MAC size variations. The level of security of memory integrity verification increases as the MAC size increases since collision rates decrease exponentially with every one-bit increase in the MAC size. Security consortiums such as NIST, NESSIE, and CRYPTREC have started to recommend the use of longer MACs such as SHA-256 (256-bit) and SHA-384/512 (512 bits). However, it is possible that some uses of secure processors may not require a very high cryptographic strength, relieving some of the performance burden. Hence, Figure 11 shows both the average execution time overhead and fraction of L2 cache space occupied by data across MAC sizes, ranging from 32 bits to 256 bits. The figure shows that as the MAC size increases, the execution time overhead for MT increases almost exponentially from 3.9% (32-bit) to 53.2% (256-bit). In contrast, for BMT, the overhead remains low, ranging from 1.4% (32-bit) to 2.4% (256-bit). The overheads are related to the amount of L2 cache available to data, which is reduced from 89.4% (32-bit) to 36.3% (256-bit) for MT, but is only reduced from 99.5% (32-bit) to 94.9% (256-bit) for our BMT. Overall, it is clear that while large MACs cause serious performance degradation in standard Merkle Trees, they do not cause significant performance degradation for our enhanced BMT scheme.



(a) Avg. Perf. Overhead **(b)** Avg. Cache Pollution

Figure 11. Performance overhead comparison across MAC size

7.4. Storage Overheads in Main Memory

An important metric to consider for practical implementation is the required total storage overhead in memory for implementing a memory encryption and integrity verification scheme. For our approach, this includes the storage for counters, the page root directory, and MAC values (Merkle Tree nodes and per-block MACs). The percentage of total memory required to store each of these security components for the two schemes: global64+MT and AISE+BMT across MAC sizes varying from 32-bits to 256-bits is shown in Table 2.

Since each data block (64B) requires effectively 8-bits of counter storage (one 7-bit block counter plus 1-bit of the LPID), the ratio of counter to data storage is only 1:64 (1.6%) versus 1:8 (12.5%) if 64-bit global counters are used. This counter storage would occupy 1.23% of the main memory of the secure processor with 128-bit MACs. The page root directory is also small, occupying 0.31% of main memory with 128-bit MACs. The most significant storage overhead comes from Merkle Tree nodes, which grow as the MAC size increases. The traditional Merkle Tree suffers the most, with overhead as high as 25% of the main memory with 128-bit MACs and 50% for 256-bit MACs. The overhead for our BMT is both smaller and increases at a much slower rate as the MAC size increases (i.e. 20% overhead for 128-bit MACs and 33% for 256-bit MACs). The reason our BMT still has significant storage overheads is because of the per-block MACs (BMT nodes themselves require a very small storage). These overheads are still significant, however our scheme is compatible with several techniques proposed in [4] that can reduce this overhead, such as using a single MAC to cover not one block but several blocks. However, the key point here is that AISE+BMT is more storage-efficient than global64+MT irrespective of the MAC size used. AISE+BMT uses 1.6× less memory compared to global64+MT with 256-bit MACs with the gap widening to 2.3× with 32-bit MACs. Hence our scheme maintains a distinct storage advantage over global64+MT across varying levels of security.

Table 2. MAC & Counter Memory Overheads

		MT	Page Root	Counters	Total
256b	global64+MT	49.83%	0.35%	5.54%	55.71%
MAC	AISE+BMT	33.50%	0.51%	1.02%	35.03%
128b	global64+MT	24.94%	0.26%	8.31%	33.51%
MAC	AISE+BMT	20.02%	0.31%	1.23%	21.55%
64b	global64+MT	12.48%	0.15%	9.71%	22.34%
MAC	AISE+BMT	11.11%	0.17%	1.36%	12.65%
32b	global64+MT	6.24%	0.08%	10.41%	16.73%
MAC	AISE+BMT	5.88%	0.09%	1.45%	7.42%

8. Conclusions

We have proposed and presented a new counter-mode encryption scheme which uses address-independent seeds (AISE), and a new Bonsai Merkle Tree integrity verification scheme (BMT). AISE is compatible with general computing systems that use virtual memory and inter-process communication, and it is free from other issues that hamper schemes associated with counter-based seeds. Despite the improved system-level support, with careful organization, AISE performs as efficiently as prior counter-mode encryption.

We have proposed a novel technique to extend Merkle Tree integrity protection to the swap memory on disk. We also found that the Merkle Tree does not need to cover the entire physical memory, but only the part of the memory that holds counter values. This discovery allows us to construct BMTs which take less space in the main memory, but more importantly much less space in the L2 cache, resulting in a significant reduction in the execution time overhead from 12.1% to 1.8% across all SPEC 2000 benchmarks, along with a reduction in storage overhead in memory from 33.5% to 21.5%.

References

[1] FIPS Publication 180-1. Secure Hash Standard. National Institute of Standards and Technology, Federal Information Processing Standards, 1995.

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

[3] L. Gao, J. Yang, M. Chrobak, Y. Zhang, S. Nguyen, and H.-H. Lee. A Low-cost Memory Remapping Scheme for Address Bus Protection. In *Proc. of the 15th International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[4] B. Gassend, G. Suh, D. Clarke, M. Dijk, and S. Devadas. Caches and Hash Trees for Efficient Memory Integrity Verification. In *Proc. of the 9th International Symposium 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 Conference*, 1999.

[6] H. Krawczyk and M. Bellare and R. Caneti. HMAC: Keyed-hashing for message authentication. http://www.ietf.org/rfc/rfc2104.txt, 1997.

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

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

[9] IBM. IBM Extends Enhanced Data Security to Consumer Electronics Products. *http://domino.research.ibm.com/comm/pr.nsf/pages/news.20060410_security.html*, April 2006.

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

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

[12] A. Kumar. Discovering Passwords in Memory. *http://www.infosec-writers.com/text_resources/*, 2004.

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

[14] 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 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[15] Maxim/Dallas Semiconductor. DS5002FP Secure Microprocessor Chip. *http://www.maxim-ic.com/quick_view2.cfm/qv_pk/2949*, 2007 (last modification).

[16] B. Rogers, Y. Solihin, and M. Prvulovic. Efficient Data Protection for Distributed Shared Memory Multiprocessors. In *Proc. of the 15th International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[17] W. Shi and H.-H. Lee. Authentication Control Point and Its Implications for Secure Processor Design. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, 2006.

[18] 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 *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[19] W. Shi, H.-H. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High Efficiency Counter Mode Security Architecture via Prediction and Precomputation. In *Proc. of the 32nd International Symposium on Computer Architecture*, 2005.

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

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

[22] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proc. of the 17th International Conference on Supercomputing*, 2003.

[23] 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 Annual International Symposium on Microarchitecture*, 2003.

[24] C. Yan, B. Rogers, D. Englender, Y. Solihin, and M. Prvulovic. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *Proc. of the International Symposium on Computer Architecture*, 2006.

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

[26] Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta. SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[27] X. Zhuang, T. Zhang, H.-H. Lee, and S. Pande. Hardware Assisted Control Flow Obfuscation for Embedded Processors. In *Proc. of the 2004 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2004.

[28] X. Zhuang, T. Zhang, and S. Pande. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Proc. of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.