Defending Against Attacks on Main Memory Persistence*

William Enck, Kevin Butler, Thomas Richardson, Patrick McDaniel, and Adam Smith
Systems and Internet Infrastructure Security (SIIS) Laboratory,
Department of Computer Science and Engineering, The Pennsylvania State University
{enck,butler,trichard,mcdaniel,asmith}@cse.psu.edu

Abstract

Main memory contains transient information for all resident applications. However, if memory chip contents survives power-off, e.g., via freezing DRAM chips, sensitive data such as passwords and keys can be extracted. Main memory persistence will soon be the norm as recent advancements in MRAM and FeRAM position non-volatile memory technologies for widespread deployment in laptop, desktop, and embedded system main memory. Unfortunately, the same properties that provide energy efficiency, tolerance against power failure, and "instant-on" power-up also subject systems to offline memory scanning. In this paper, we propose a Memory Encryption Control Unit *(MECU) that provides memory confidentiality during system suspend and across reboots. The MECU encrypts all memory transfers between the processor-local level 2 cache and main memory to ensure plaintext data is never written to the persistent medium. The MECU design is outlined and performance and security trade-offs considered. We evaluate a MECU-enhanced architecture using the SimpleScalar hardware simulation framework on several hardware benchmarks. This analysis shows the majority of memory accesses are delayed by less than 1 ns, with higher access latencies (caused by resume state reconstruction) subsiding within 0.25 seconds of a system resume. In effect, the MECU provides zero-cost steady state memory confidentiality for non-volatile main memory.*

1 Introduction

Main memory containing sensitive information persists for indefinite periods during system uptime [6]. Recently, Halderman et al. [15] demonstrated how to extend main memory persistence by "freezing" DRAM chips to maintain memory cell state after the system is powered off, allowing an adversary to retrieve any passwords or cryptographic keys that were not overwritten before system shutdown. While this attack provides an effective vector for key retrieval, the adversary must have physical access *before* the system is shut down. This precondition becomes unnecessary as new non-volatile memory technologies emerge.

Non-volatile memories such as MRAM (magnetic RAM) and FeRAM (ferro-electric RAM) [20] provide energy efficiency, tolerance of power failure, and "instant-on" power-up. These technologies are reaching maturity and manufacturers are already selling chips with up to 4-Mbit of storage [11, 12] to replace battery-backed SRAM in embedded systems. Recent advancements in speed [9] and capacity [21] make these technologies appropriate for main memory in laptops, desktops, and embedded systems. Because systems that use non-volatile main memory retain all state across reboots and suspends, users need not endure long boot cycles or memory restoration from slow secondary storage during resumption.

The characteristics of non-volatile main memory (NVMM) that provide these advantages also introduce new vulnerabilities–sensitive data can be extracted or modified by an adversary who gains access to the memory while the computer is not turned on or after reboot. Unlike the attack described by Halderman et al., no freezing is required, and the memory chips can be retrieved at any time. This work seeks to mitigate the vulnerabilities of persistent main memory while retaining the advantages of non-volatile memory. Note that these techniques are also effective against frozen volatile memory chips.

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 defines the problem and threat model. Section 4 describes our solution. Section 5 evaluates the performance impact of the MECU using SimpleScalar. Section 6 considers a number of practical issues in the use of the MECU and its application to next generation processors. Section 7 concludes.

2 Secure Memory Systems

Operating systems and applications assume memory does not survive across reboots. Sensitive information such as passwords and cryptographic keys commonly reside in main memory [6]. If this data is written to magnetic media (e.g., via swap operations), it may persist even

*This material is based upon work supported by the National Science Foundation under Grant No. CCF-0621429, CNS-0627551, and CNS-0643907. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

longer [14]. Therefore, best practice recommends ensuring memory plaintext never reaches disk. While data can be securely deallocated [7] and crash reports can be cleansed [3], encrypted swap [23] is still required for reused data.

The introduction of NVMM invalidates a basic assumption upon which operating system and application security is based. Therefore, it is imperative that the underlying architecture transparently preserve the security guarantees upon which the systems where built, i.e., mechanisms must be implemented within the hardware and BIOS. Our approach is unique in that it considers full memory encryption without OS interaction and provides optimizations specific to systems with NVMM. Many previous memory encryption architectures [8,17–19,22,31] were designed for a vertical set of applications, e.g., Digital Rights Management (DRM) and tamperproof computation for grid processing. As such, only the memory segments of "protected" applications are encrypted. This DRM model has two significant disadvantages: it often requires changes to the processor instruction set, operating system, and/or applications, and significant performance degradation results from processor stalls necessary for protection against *online* attacks. A similar side effect exists in architectures providing protection against bus sniffing [10]. Securing NVMM need not necessarily require protection against online attacks, therefore the associated performance penalty is avoidable.

While many previous systems do not directly provide full memory encryption appropriate for efficiently protecting systems with NVMM, lessons can be learned from their evolution. Execute Only Memory (XOM) [19], an early architecture designed to protect DRM applications, encrypted data directly, resulting in significant performance degradation. Suh et al. [31] improved performance by applying a variant of counter mode encryption to generate one time pads in parallel with memory lookups. However, in order to protect against online attacks, the secure processor must store one four-byte sequence counter for every protected 64-bytes of memory (for systems with 64-byte cache lines). The counters must be stored within the secure processor to avoid the overhead of performing two memory accesses per cache miss. As these storage requirements are often impractical, subsequent architectures minimized on-chip storage using caches [33] and prediction algorithms [25, 28]. Unfortunately, these techniques still result in a significant memory bottleneck throughout system run time. Further, storing counters in memory is insecure, therefore Yan et al. [32] ensure counter integrity using hash trees similar to architectures designed by Suh et al. [13, 31]. In addition to ensuring counter integrity, Yan et al. also split the counter into major and minor portions, thereby further decreasing storage size. While their architecture provides improved performance, the overhead due to processor stalls is constant throughout the system operation. Additionally, an architecture designed to protect the entire main memory must be careful when storing counters to memory, otherwise the counters may become inaccessible.

These preceding approaches fail to preserve the security guarantees that modern operating systems will place on NVMM. These operating systems require that the memory architecture defend against *offline* physical attacks and avoid run-time processor stalls–a unique combination of feature and performance that *no* memory system has previously achieved. Furthermore, the architecture must support all legacy software and hardware interfaces, including DMA and multiprocessors [24,29], and do so within a modest component footprint. We explore how these features are simultaneously achieved within our MECU-enhanced architecture in the following sections.

3 Non-Volatile Main Memory

Consider a commodity desktop machine with power management capabilities. During normal operation, the system is *active*, i.e., usable for processing data, performing reads and writes from memory, etc. When the system is not in use, it can move into a state of low power consumption, either automatically or through user invoked *suspension*. There are two different suspend modes: *powered suspend* and *unpowered suspend* (commonly known as *hibernate*). When a volatile memory system enters powered suspend mode, power-intensive components (e.g., displays and disk drives) are turned off, while reduced power is applied to others (e.g., main memory). Importantly, memory contents persists while in the low power state. When a system with volatile memory is placed into hibernate mode, main memory is transferred to secondary storage (e.g., disk) and power cut off, effectively zeroing the physical memory. When the system is resumed, the memory is restored from secondary storage. Conversely, architectures with NVMM need not provide any facilities to retain memory state within power management services, as contents remain indefinitely (even across system reboots).

Two attack vectors are enabled by the introduction of NVMM into current architectures—an *online attack* where a booted operating system accesses a previously booted operating system's memory, and an *offline attack* where the physical memory is probed by an adversary while the system is powered off, e.g., through regular read-out ports or via more sophisticated techniques such as optical probing of the memory with a laser and electromagnetic analysis [26]. We do not seek to protect main memory in normal operation, as solutions already exist [1]. Additionally, we do not consider hibernation as solutions such as encrypt-on-hibernate and modifying file caches [5] address these issues. For clarity in distinguishing between a reboot and suspend, we introduce the concept of an *OS instance*. We assume that the system has the ability to suspend operations as it transitions into suspend mode and to subsequently resume its previous state. The system thus has the same OS instance before the system is suspended and after it resumes, but a different instance after it reboots.

An OS may reboot systematically or abruptly. In an online attack, the new OS instance attempts to access the previous instance's memory. Traditional OS security models assume volatile main memory not survive across a reboot (while undesirable, this is not always the case [7, 15]). We require that this characteristic also hold for non-volatile systems. The potential for abrupt power loss mandates that the system always remain in a protected state: any attempt to provide protection solely at suspend or shutdown could be trivially circumvented by an adversary who cuts power before the security mechanisms are applied.

The vulnerabilities introduced by the use of NVMM lead to the following informal design goals for the MECU. First, a MECU-enhanced system must be resilient to physical attacks on suspended memory. In particular, we principally desire to protect *confidentiality* of the memory: an adversary must not be able to derive the content of main memory when the system is suspended or powered off. We defer issues of *integrity* in the initial MECU design and analysis, but sketch possible solutions in Section 6. Second, no operating system state should be retrievable after shutdown or reboot. Third, protections must be maintained without support from, or trust in, the operating systems running on the host. Fourth, the protections must require little change to the hardware architecture and operate virtually invisibly to the rest of the system architecture. We term these latter two goals *transparency*. Finally, the solution must induce little overhead on memory accesses. Note that this work is restricted to the security of main memory only. Systems that expose sensitive data on disk will still be vulnerable. For example, past systems have shown that virtual memory paged to disk can expose a significant amount of sensitive information [23]. Other system artifacts such as network traffic can similarly expose information [2]. Such vulnerabilities are outside the scope of the current work.

4 MECU Design

The threat model outlined above requires that data can never be present in main memory in the clear: an adversary able to abruptly cut power could thereafter read any plaintext data present in memory. Therefore, we adopt an approach in which data is encrypted when written by the processor to main memory and decrypted when read, i.e., all memory operations are mediated by the MECU. This has the advantage of transparency, where no changes to either the processor or memory organization are necessary to achieve the desired security. To be more precise, we introduce a MECU on the memory bus between the processor-local layer 2 memory cache and NVMM. Figure 1 depicts the MECU's placement in the architecture.

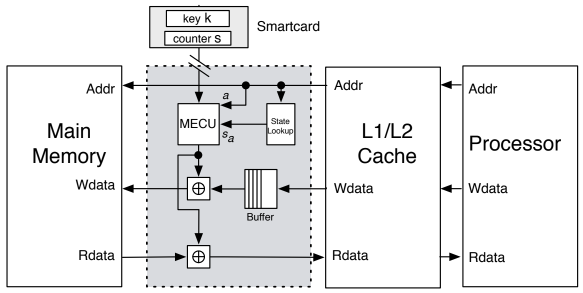


Figure 1. A MECU-enhanced architecture

The central design challenge of this approach is to ensure that the mediation of memory operations is both secure and efficient. The naive implementation of processor writes through the MECU encrypt data directly using a suitable encryption algorithm (e.g., DES, AES) then write the resulting ciphertext to memory. Read operations reverse this operation, decrypting the ciphertext before use. Because writes and reads would be delayed by cryptographic operations, unacceptable delays would be introduced. These delays would ultimately lead to processor stalls (idle periods where processor waits for memory operations to complete).

Encryption overhead can be mitigated by creating pads that are XORed with plaintext to perform encryption. Here, the creation of the pad is the computationally expensive (and potentially offline) operation. Also illustrated in Figure 1, we apply this approach in the MECU, where we mask the pad computation costs in read operations by parallelizing the memory fetch operation and pad creation. The MECU computes the pad while the cache line is being fetched from memory.[1] Because the pad can be created faster than the fetch delay, the pad is ready when the data arrives on the bus from main memory. Therefore, the observable overhead for each memory access is only the one or two gate delays (depending on the fabrication technology) needed to XOR the data with the pad.

Memory writes are similar to reads. The MECU generates and applies the pad for each memory write as described above. In this case there is no latency to mask the pad creation overhead. The MECU uses a write buffer to mask both the encryption and memory delay, similar to methods of reducing write latencies in write-through caches. Instead of waiting for the pad creation operation to complete before writing, the data is written into a MECU internal buffer. When the pad is created (some cycles later), the data is XORed and written to main memory.

While this high-level approach of XORing pads generated in parallel to memory access exists in previous systems [28, 31, 33], pad generation techniques vary. Securely and efficiently storing and accessing seed informa-

[1]For simplicity, we refer to cache line sized blocks in main memory as a *cache line.* We assume a 64 byte cache line size in all discussions and experiments below, but all results remain valid for any cache line size.

tion presents a nontrivial architectural challenge, and previous systems incur significant performance or storage penalties. As described below, characteristics specific to NVMM systems allow optimizations to provide memory confidentiality with nominal overhead.

4.1 Pad Generation

As mentioned above, we assume that the adversary can read the entire contents of main memory (and the memory on the MECU) each time the computer is suspended. Let M_t denote the unencrypted, logical contents of the memory at time t, and let C_t be the real contents seen by the adversary who can access the raw NVMM (C_t consists of one ciphertext for each cache-line-sized block of main memory, plus an array of "state counters" stored on the MECU – see below). The standard notion of confidentiality in cryptography is *semantic security*: namely, for any two sequences of logical memory contents $M_0, M_1, M_2, ...$ and $M'_0, M'_1, M'_2, ...$, the adversary should not be able to tell which of the two sequences of plaintexts was actually encrypted. The scheme described here achieves a slight variant: as cache lines are only re-encrypted upon re-write, the adversary may learn that certain portions of the memory were not written to during a particular resume cycle.

We first outline the scheme, then discuss the notion of security and implementation. The main components are:

(a) A master key k. This key is refreshed (i.e., generated at random) when the system is rebooted.

(b) A *state counter* s (16 bits will typically suffice). This counter is reset to 0 on reboot, and incremented by 1 on each resume. (Thus, it counts the number of resume cycles since the last reboot).

(c) An array of 16-bit *timestamps*, one per memory block (for now, think of blocks as cache lines). The entry s_a records the value held by the state counter the last time block a of the memory was written to (i.e., s_a is the number of the last resume cycle during which the processor wrote to block a). To be clear: timestamps do not record physical time, only the state counter value.

The array of timestamps **(c)** is stored on the MECU itself. The master key **(a)** and state counter **(b)** are stored on a removable device such as a smart card (see below). This device is assumed to be removed on suspend.

The pads used for encryption are created by applying a pseudorandom function $\mathcal{F}_k(\cdot)$ to the pair (a, s_a). Intuitively, this ensures that each pad is indistinguishable from a uniformly random string, even given all the other pads used in the system (even on different suspends). It also ensures that a given pad is never used to encrypt different messages on different suspends.

Blocks are only re-encrypted when written to by the processor. Therefore, an adversary who observes the memory on successive suspends can learn that whether a particular block was overwritten. To specify the security properties more precisely, we define the *write footprint* of a particular sequence of resume/suspend cycles to be a sequence of sets of memory blocks $S_0, S_1, S_2, ...$, where S_i is the set of memory blocks written to during the ith resume cycle. In our scheme, a passive adversary learns the write sequence but nothing else. More specifically, the scheme maintains *completeness:* assuming that the adversary is passive and does not modify any information stored in the MECU or main memory, the system behaves as expected. It also maintains *security*, as described below.

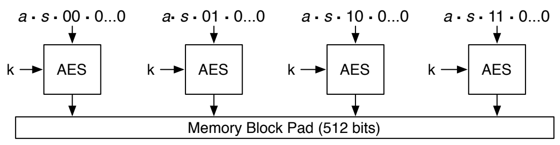

$a \cdot s \cdot 00 \cdot 0...0$
$a \cdot s \cdot 01 \cdot 0...0$
$a \cdot s \cdot 10 \cdot 0...0$
$a \cdot s \cdot 11 \cdot 0...0$
k
AES
k
AES
k
AES
k
AES
Memory Block Pad (512 bits)

Figure 2. A simple pseudorandom function $\mathcal{F}_k(a, s)$ implemented using AES.

Let A be a passive adversary who observes the main memory and MECU contents on every suspend cycle (over multiple reboots), and does not have direct access to the key k. Consider choosing at random between two runs of the system with *identical write footprints*, and giving A access to one of the two runs. The probability that the adversary can guess which of the two runs she is observing is at most $\frac{1}{2} + O(\epsilon)$, where ϵ is the advantage a related adversary A' would have at distinguishing $\mathcal{F}_k(\cdot)$ from a random function. The adversary A' simply simulates the system (that is, encryption plus the adversary A), making appropriate queries to $\mathcal{F}_k$. The running time of A' is the running time of the OS plus that of A. Hence, if the pseudorandom function family $\mathcal{F}_.(\cdot)$ is secure against a polynomially-bounded adversary, then the MECU prevents leakage of any information beyond the write footprint of a particular run.

As described above, any secure pseudorandom function $\mathcal{F}$ from a large enough input space (enough to contain the address of a memory block and a state counter) to a large enough output space (the size of a memory block) will suffice for generating pads. The main efficiency requirement is that the PRF be fast enough for the pad to be generated in the difference between the round-trip time from the MECU to the main memory and the time necessary to fetch the timestamp s_a from the MECU's memory. This ensures the pad will be ready before main memory responds and minimizes the delay observed by the CPU.

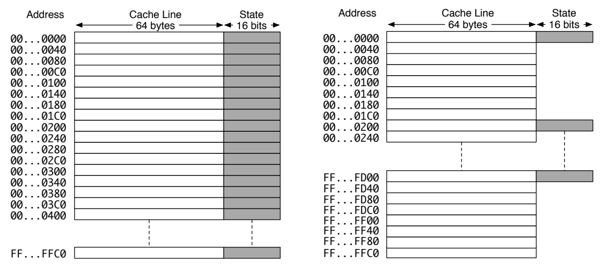
A particular PRF, based on AES, is described in Figure 2. To evaluate $\mathcal{F}_k(a, s)$, one calls AES with key k on several inputs constructed from (a, s) by appending extra digits. For example, in an architecture with 64-byte memory blocks as in Figure 2, $\mathcal{F}$ makes 4 parallel calls to AES. If $\epsilon_{PRP}(q)$ is the probability that the adversary A can distinguish AES$_k$ from a truly random family of *permutations* using q queries, then the probability that A can distinguish AES$_k$ from a random family of *functions* is at most $\epsilon_{PRP}(q) + \binom{q}{2} \cdot 2^{-128}$ (extra term due to the birthday paradox). This implementation is convenient since fast hardware implementations of AES exist and their timings are well-studied. To understand the speed disparity between memory access times and in-hardware AES, consider the high-speed Rambus DRAM (RDRAM). Access time requirements for a 64-byte cache line are 131.25 ns, based on a 3.75 ns clock cycle for the memory bus [16]. Given a low-end desktop machine with a 1 GHz processor, an AES encryption will require 44 cycles, or 44 ns, far below the memory access speed [28, 31, 33]. That said, our simulations indicate that the PRF evaluation is not an efficiency bottleneck in our proposed MECU architecture, so it is likely that other PRF implementations would work equally well.

As mentioned above, we propose using a 16-bit state counter to prevent pads being reused to encrypt different data. The state counter size has an effect on the maximum uptime for an OS instance. For example, a two-byte state counter supports up to 65,536 separate suspend/resume cycles before an OS reboot must be forced. In this case, the OS could be suspended and resumed an average of 179 times a day, or 7.5 times an hour, every hour for a year before requiring a OS reboot. For all practical purposes, this is an infinite number of suspensions,[2] and thus the counters are at least as large as needed for these systems. Smaller state counters may be more desirable, but we defer consideration to future work. We adopt conservatively large 16-bit state counters in all experiments discussed below.

The security of the encryption scheme relies on keeping k secret. Storing k on the MECU is problematic since we assume the adversary has access to the MECU contents during suspends. After considering several alternatives, we decided to place k on a removable smart card (or similar removable storage directly connected to and controlled by the MECU firmware). This ensures that the system is resilient to an offline physical attack as long as (*i*) the smart card is removed during suspends and *(ii)* the circuitry which uses k when the system is live bears no memory once power has been suspended. We consider the practical use and implications of the smart card in latter sections of this paper.

4.2 Storage Optimization

For many architectures, the burden of providing a state counter for every cache line may be unmanageable. To illustrate, a system with a 4 gigabyte RAM memory requires 128 megabytes of non-volatile state counters internal to the MECU—a considerable design and manufacturing challenge. These costs can be mitigated by sharing a state counter between multiple cache lines. Here the MECU organizes contiguous cache lines into *memory blocks* sharing a single state counter. Figure 3 juxtaposes the individual and shared counter strategies. The cost savings can be substantial: in the above example, sharing a counter among 64 lines drops the requirements from 128MB to 2MB.



(a) Per cache line state counters. (b) Shared cache line state counters.

Figure 3. Optimizing storage via shared state counters (in-MECU storage shown in gray).

Shared state counters require further changes to the MECU design and operation. When a counter is updated for one cache line, all other cache lines within that block must be encrypted with a pad based on the new state counter value. Hence, the MECU must retrieve, re-encrypt, and write back to main memory each associated cache line following a counter update. Fortunately, because counters are only updated after the system is resumed, each memory block must only be re-encrypted once per system resume.

Shared state counters exhibit subtle trade-offs between performance and storage costs. In the degenerate case, all of physical memory would map to a single state counter. In this case, as soon as one cache line is written to memory after a resume, all of memory must be re-encrypted. However, this is a one-time cost (per resume). By grouping cache lines into smaller blocks, we allow for *lazy re-encryption*, wherein only the cache lines spatially close to accessed memory must be re-encrypted. As the breadth of memory access increases, more blocks will be re-encrypted, effectively diffusing the one-time cost into a series of smaller costs. Section 5 empirically explores these trade-offs.

The optimized state counter storage can thus be computed using the following equation:

$$Size = \frac{S_{mem} \cdot \log_2(N_{state})}{S_{line} \cdot N_{lines}}$$

where S_{mem} is size of the byte-addressable physical address space (2^{32} for a 32-bit processor)[3], S_{line} is the size of a cache line in bytes (typically 64), N_{state} is the number of states supported, and N_{lines} is the number of cache lines in a memory block. Modifying the number of states only logarithmically affects in-MECU storage, while an inverse linear relationship exists between the number of cache lines per memory block and storage size. Thus, storage requirements are better decreased by increasing the number of cache lines per memory block, rather than reducing the number of states.

[2]The probability any current system survives 65,000 suspends without software or hardware failure requiring OS reboot is very, very low.

[3]Note that this only needs to be equivalent to the maximum memory size supported by the system (e.g., an embedded system may be designed to support 256 MB of physical memory, so an S_{mem} of 2^{28} is sufficient).

4.3 Active Attacks

The MECU provides protections against a limited active adversary. Because the key and global state counter are stored on the removable smart card (Figure 1), an adversary with access to a machine during suspends cannot change either the key k or the global state counter s. In particular, this prevents forced re-use of a one-time pad. However, it does not prevent *chosen-ciphertext* attacks, which subtly use auxiliary information that the adversary has about the information stored in memory and the way in which it is processed. Preventing general active attacks requires a significantly more involved and time-intensive solution.

The MECU's ability to preserve confidentiality is predicated somewhat on the integrity of its internal state. An adversary manipulating an unprotected MECU could decrement the global state counter s while the system is offline, thereby forcing the system to regenerate a previously used pad on subsequent writes of the same cache lines. This would lead to the exposure of multiple cache lines encrypted under the same pad. The adversary could then use information about one plaintext to expose the other–thus nullifying confidentiality. Storing s on the removable device prevents this. A more subtle attack is possible, however, by modifying the memory and then observing the behavior of the system. Assume the adversary knows that either a "*sell*" or "*buy*" order for a stock is encoded in a cache line. The adversary can then apply "*sell*" $\oplus$ "*buy*" to change the order from "*sell*" to "*buy*" or vice-versa. If M_t = "*buy*" in the cache line, the adversary modifies the cache line as such: $F_k(a, s_a) \oplus M1 \oplus (\text{"}sell\text{"} \oplus \text{"}buy\text{"}) = F_k(a, s_a) \oplus \text{"}sell\text{"}$ The adversary does not know the original value or how it changed, but the user's behavior will change depending on the original value. If the adversary can observe the system behavior at run time, they may use that side-channel to compromise the confidentiality of the original value, such as listening to a broker calling to stop a "sell" order, or capturing network packets associated with the order. Alternatively, no side-channel is necessary if the change in the user's behavior is reflected in the memory locations to which he writes (e.g. he sends email to either his broker or his system administrator, resulting in writes to mail archives stored in different known memory blocks). The adversary may observe the user's behavior by looking at the memory contents the memory on consecutive suspends (in other words, the write footprint itself may form a useful side-channel). Avoiding this chosen-ciphertext attack requires adding integrity checking to the critical path from the main memory to the CPU. We consider this in greater detail in Section 6.1.

5 Performance Evaluation

This section explores the performance of the MECU and considers the performance trade-offs and costs of its realization in hardware. We begin in the next section by describing our SimpleScalar simulation of an MECU-enhanced architecture. The simulation environment is then used to empirically measure the overheads of the MECU and to characterize the design trade-offs of our approach.

Table 1. Simulation parameters. Cache latencies in parentheses.

Parameter	Value
L1 D-Cache	32KB, 8-way with 64B block (1)
L1 I-Cache	32KB, 8-way with 64B block (1)
L2 Unified Cache	256KB, 4-way with 64B block (6)
L2 Write buffer	16 lines
Memory Bus Width	8B
Memory Latency	80 cycles with 2 cycle inter-chunk latency
AES Latency	44 cycles
State Lookup	3 cycles
XOR Latency	1 cycle

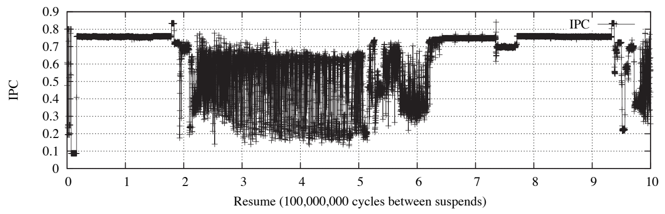
5.1 SimpleScalar Simulation

The SimpleScalar simulation framework [4] is widely used to study the complexity and performance of microarchitecture designs. The platform supports SPEC CPU2000 [30] benchmarks designed to evaluate designs by exercising the memory hierarchy. We used SimpleScalar/PISA to analyze the impact of the MECU on system performance, and updated the architectural component influencing the L2 cache miss latency to model MECU-related delays, defining a fixed write buffer size. We then added tunable parameters to allow a wide range of simulations for calculating the MECU delay.

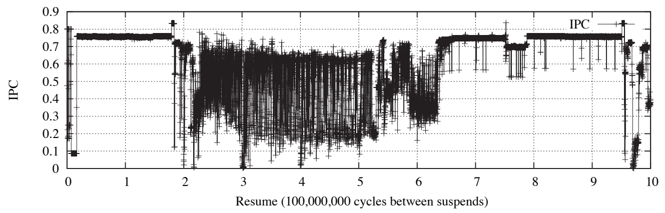
Memory block re-encryption was modeled by keeping track of the number of resumes since reboot. We refer to the number of resumes as the *simulation state*; a suspend/resume cycle is simulated by incrementing a state iterator. The simulator begins in state zero, and all memory blocks therefore also have an initial state of zero. The first time a memory block is written after system resume, the processor stalls as the block is re-encrypted. We model this stall in the simulation. The memory block state is then immediately updated to the current simulation state, indicating subsequent writes in this state do not require re-encryption.

5.2 Performance

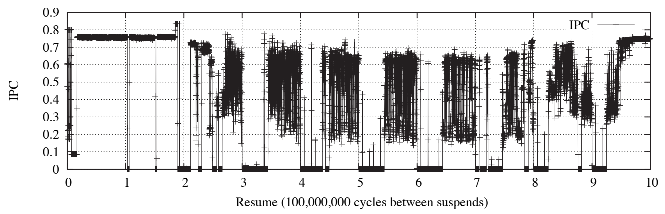
The SPEC integer benchmarks explore system performance by exercising memory in diverse ways. We chose a subset of the SPEC benchmarks representing a diversity of application types: bzip2, gcc, parser, twolf, and vpr. The bzip2 benchmark consistently uses a large amount of memory, while the gcc benchmark's memory use is more sporadic. The parser, twolf, and vpr benchmarks have lower consumption, but introduce vastly different access memory workloads. We measure the performance of the system by observing the number of *instructions per cycle* (IPC). This metric is the processor's instruction throughput that varies slightly depending on the types of instructions executed. However, for the purposes of our analysis, IPC is an



(a) IPC for no MECU



(b) IPC for a MECU with a memory block size of 256 cache lines



(c) IPC for a MECU with a memory block size of 65,536 cache lines. For suspend cycles 3 through 7, the stall averages 45,480,000 processor cycles. For suspend cycles 8 and 9, the stall is 25,000,000 processor cycles.

Figure 4. IPC with and without a MECU (bzip2)

accurate metric for measuring processing throughput. The MECU does not impede memory throughput, as pad generation occurs in parallel with memory access; rather, we consider latency due to re-encryption. Table 1 overviews parameters common to all experiments.

5.2.1 Analysis of Processor Overhead

Our first set of experiments attempted to characterize the processor performance as a function of the state counter block size. Figure 4 shows simulation runs of 1 billion cycles for systems running the bzip2 benchmark with no MECU, a MECU with a block size of 256 cache lines, and a MECU with a block size of 65,536 cache lines. Each subfigure shows the IPC over time, with a simulated system suspend and resume every 100 million cycles (marked with the integer 1-10 on the x-axis time scale). The sample rate was 100,000 cycles. The y-axis shows the instructions per cycle, where the theoretical maximum is 1.

Visual inspection of the traces shows the impact of processor stalls caused by re-encryption following system resumes. The initial performance of the system is largely stable in all three traces for the first 200 million cycles. This is because few memory writes occur in initial process setup. The traces diverge significantly after the second suspend. The non-MECU and 256 cache line sized block traces show similar behavior, with the latter exhibiting prominent but brief processor stalls immediately after resumes 2, 3, and 4. The 65,536 cache line sized block MECU shows considerable processing oscillation, where long periods of stalling occur following every resume. This observation is not entirely surprising, as immense block size requires 4 MB of memory be processed for each re-encryption. The impact of memory block size on performance is more readily observed when IPC is averaged over the suspend cycle. We ran a second series of tests for all benchmarks under various cache line sizes. Figures 5 and 6 show the percent decrease in IPC compared to a baseline system with no MECU in two of these tests. Similar performance costs as reported above are observable, with the decreases in IPC becoming more pronounced as the block size increases. One interesting element is the apparent performance increase in later resume cycles of the bzip2 experiments, caused by increased memory address locality in this phase of the process—with fewer used addresses, less re-encryption is needed.

Block sizes of 256 and smaller perform similarly in the bzip2, with a maximum overhead of approximately 9% IPC reduction observed in the seventh resume cycle. The gcc benchmark is representative of the other benchmarks (hence discussion of other benchmark results is omitted for brevity). Block sizes of 4,096 cache lines and smaller in the gcc experiment all produced similar performance results. The maximum overhead was observed in the sixth resume cycle, with a block size of 4,096 incurring a 4.4% overhead and a block size of 256 incurring only a 2.1% overhead.

Based on these results, we assert that for systems running a wide range of applications, including those with intensive memory requirements, a memory block size of 256 cache lines provides a good balance of storage requirements versus incurred overhead. If demanding and wide-ranging memory requirements are only rarely encountered, storage requirements can be further minimized by using blocks of 4,096, with only minimal additional overhead.

5.2.2 Time to Quiescence

The reduced IPC is a direct effect of processor stalls due to memory block re-encryption. Our simulations showed that after some period of time, all memory blocks were re-encrypted with new state counters and no further encryption operations were necessary. We term this the *time to quiescence*. This duration bounds impact of the MECU; once the system reaches steady state, the overhead is negligible.

To more fully observe quiescence, we increased the time period between suspends to 1 billion processor cycles and simulated 5 suspend/resume cycles. This was long enough for each benchmark to access its full range of memory blocks. The first cycle is discounted as no re-encryptions occur; memory is initially seeded at this stage. Figure 7 summarizes the time to quiescence as indicated by processor cycles, averaged over the 4 remaining cycles.

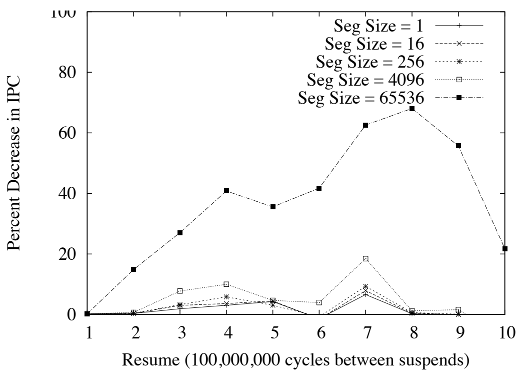


Figure 5. Percent Decrease in IPC per power cycle in bzip2 benchmark

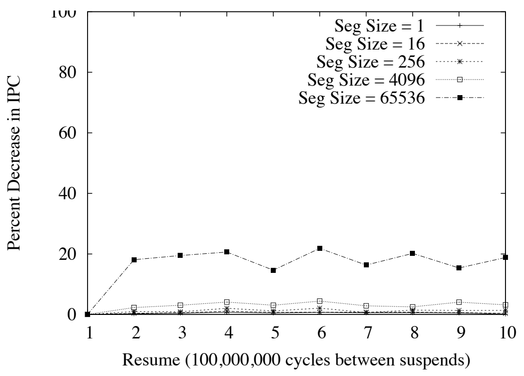


Figure 6. Percent Decrease in IPC per power cycle in gcc benchmark

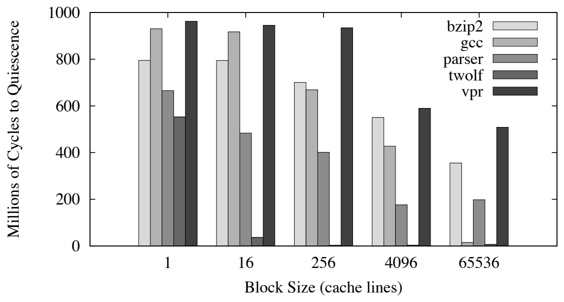


Figure 7. Average Time to Quiescence

In general, increasing the memory block size reduces the time to quiescence, as larger blocks contain a larger proportion of the address space, and accessing any address within that block causes the entire block to be re-encrypted. This time to quiescence is highly dependent on the breadth of memory access; for example, a memory block size of 65,536 under the bzip2 benchmark takes almost twice as long to quiesce as a block size of 4,096 under the parser benchmark. Additionally, for a given memory space, not all memory is necessarily re-encrypted; there could be memory in a system that is accessed once and subsequently left unused for large lengths of time, potentially the duration of the system's operation before resetting. Quiescence merely indicates for how long an application will be subject to the effects of memory re-encryption. If a system can tolerate short stalls, larger memory block sizes may be appropriate.

5.2.3 System Level Costs

Each memory segment is re-encrypted with a new pad at most once per system resume. There is generally a high degree of spatial locality to an application's memory accesses; thus, only a small set of memory blocks tends to be used. Because of our lazy re-encryption optimization, blocks are only re-encrypted when first accessed after resume. Once the initial segment re-encryption has been performed, subsequent reads and writes to memory incur an overhead of only one or two gate delays, less than 200 ps. Hence, from a system perspective, the impact of a MECU is minimal.

Consider again the degenerate case, where one memory block comprises the entire memory space. When the system resumes from a suspend state, all of system memory is then re-encrypted. Because encryption of the memory can occur in parallel to its access, the delay is equivalent to accessing the entire memory space twice, once to read and once to write back the encrypted memory. This latency is similar to that caused by the *Power On Self Test* (POST) system RAM check performed by many PC BIOSes.[4] While noticeable as a delay, even under this extreme case the delays are not onerous to the point of making the system unusable.

In practice, the delays will be minimal. As previously discussed, the time to quiescence is a function of processor cycles, and is thus dependent on the processor's clock frequency for real time values. Table 2 shows the time to quiescence with the bzip2 benchmark by processor frequency. The 10 MHz column represents an embedded system with a very slow clock, while the 100 MHz column represents an embedded system with medium power. The 1,000 MHz column represents a low-end desktop computer and the 3,000 MHz column is representative of the typical processor speed of a moderately-specified desktop machine (e.g., a Dell Dimension E520). These experiments are reasonably predicated on the assumption that the speed of memory scales proportionally to processor speed, i.e., it is not the source of latency. These experiments centrally show that the time to quiescence in all environments will increase as the ratio of processor speed to memory bandwidth increases. Furthermore, there is a trade-off between shorter times to quiescence and instantaneous process stalls: because the stall time per memory block is directly proportional to block size, a shorter time to quiesence will cause a greater instanteneous processor stalling cost.

Of all simulated sizes, a memory block of 256 cache lines offers a good trade-off btetween storage requirements,

[4]Note that the "Quick Boot" feature provided by some manufacturers does not perform this RAM check.

Table 2. System Level MECU Impact (bzip2)

Block Size	State Storage	Time to Quiescence*			
		10 MHz	100 MHz	1,000 MHz	3000 MHz
1	128 MB	79.50 s	7.950 s	0.7950 s	0.2650 s
16	8 MB	79.43 s	7.943 s	0.7943 s	0.2648 s
256	512 KB	70.03 s	7.003 s	0.7003 s	0.2333 s
4096	32 KB	55.05 s	5.505 s	0.5505 s	0.1835 s
65536	2 KB	35.53 s	3.553 s	0.3553 s	0.1184 s

*Assuming a constant memory to processor speed ratio

IPC impact, and time to quiescence, the latter achieved after only 0.23 seconds on a moderately-specified desktop machine. Embedded systems are often more resource constrained, therefore sacrificing performance after resume is likely to be desirable.[5] High-end system with high performance requirements would likely be designed with more MECU-internal storage.

6 MECU Extensions

6.1 Main Memory Integrity

Our initial design and simulation did not preserve the integrity of main memory. If the adversary knows the plaintext corresponding to a memory address, the ciphertext can be replaced without detection while the system is suspended. This integrity attack does not result in a loss of confidentiality, but results from a previous loss of memory confidentiality, at which point in the system is already compromised. To illustrate, recall that a ciphertext c_{a_i} (as written to main memory) for address a_i is computed from the plaintext p_{a_i}, state counter s_i and key k as $c_{a_i} = \mathcal{F}_k(a_i, s_i) \oplus p_{a_i}$. Now assume the adversary knows the underlying plaintext of the cache line (p_{a_i}) and wants to replace it with malicious data m. She computes $c_{a_i} \oplus p_{a_i} \oplus m$ and writes that value to the cache line. When the system resumes and the cache line is read, the MECU will recover the plaintext by computing $p_{a_i} = \mathcal{F}_k(a_i, s_i) \oplus c_{a_i}$ to recover the plaintext. Substituting the malicious cache line value and original c_{a_i} gives

$$p_{a_i} = \mathcal{F}_k(a_i, s_i) \oplus ((\mathcal{F}_k(a_i, s_i) \oplus p_{a_i}) \oplus p_{a_i} \oplus m) = m$$

Because the pads and original plaintext cancel out, we are left with the malicious cache line value. Note again that this attack is contingent on the adversary having already breached the memory's confidentiality. Additionally, the scope of the attack is limited to the memory locations known to the adversary. Other memory locations, even within the same cache line, remain integrity-protected.

Integrity verification has been long studied within the context of general main memory protections. Proposals include hashing over cache lines and storing values in separate or statically-allocted memory [18], creating a Merkle hash tree of memory segments and caching the results [13, 31], and using GCM encryption to simultaneously encrypt and hash memory blocks [24]. Adapting these solutions to our architecture requires a careful analysis of the performance and storage tradeoffs, and we defer further consideration of these issues for future work.

6.2 Moving to 64-bit Processors

The storage calculations to this point have assumed that the system has a 32-bit processor. As 64-bit processors are deployed, their addressable memory space expands to as much as 2^{64} bytes, or 16 exabytes (16.8 million TB). Encrypting this entire memory space enormously increases the MECU's storage requirements. Current transitional architectures only support a subset of the 64-bit physical address space, e.g., Intel EM64T 64-bit processor extensions currently support 1 TB (2^{40} bytes) of physical memory, 256 times as large as the 32-bit address space. A MECU operating under the same assumptions given in Section 5 would require 8 MB of internal storage. However, if the system uses a cache line of 128 bytes already existing in some Intel Pentium 4s, this reduces to 4 MB, a reasonable amount given the 1 TB protected. As architectures scale to support larger addressable memory sizes, the MECU will scale with the support of larger available cache lines.

6.3 Direct Memory Access

Many hardware architectures allow for I/O transfers directly between main memory and devices such as disk drive controllers or graphics cards, a process called direct memory access (DMA). With the MECU solution of memory encryption, memory would require decryption before being transferred on the DMA bus, while main memory could not be accessed by the processor during an I/O transfer due to the MECU decrypting to the cache. This renders DMA infeasible. We propose implementation of a second MECU for architectures that support DMA that resides between the main memory and the DMA bus, which would then be able to decrypt and encrypt data going to and from the I/O devices while the processor would be free to access memory without being hampered. This solution requires state counters to be shared between MECUs, which conserves storage. While the shared table will not allow simultaneous access, the blocking caused by table locks would be minimal in comparison to the overhead induced by coherency protocols. Furthermore, this approach can be extended to an arbitrary number of MECUs, thereby supporting multiprocessor and multi-core architectures.

6.4 Power Failures

While non-volatile memory systems are attractive due to their resilience to power failure, memory consistency could still be affected if a power failure occurs while the MECU is encrypting a memory block. Therefore, capacitors must be present for the MECU to provide the power necessary for the memory block encryption to be completed. These considerations are not limited to the MECU: for a system to provide resilience against power failure, all components must respond gracefully, such as hard disks parking

[5]Embedded systems with a 10 MHz clock traditionally have a 16-bit address space. This greatly lowers MECU storage requirements.

their drive heads to avoid crashing them into disk platters if power is lost [27].

7 Conclusion

We have designed an efficient MECU to achieve the same level of security provided by traditional volatile main memory systems, and evaluated the performance impact using the SimpleScalar framework. Introducing the MECU into a system's architecture introduces overhead of only 9% in the worst case and less than 2% for average workloads for a period of less than 0.25 s after system resumption, based on a moderately-specified desktop. During regular operation, the costs of encryption and decryption are less than 1 ns. In effect, the MECU provides zero-cost steady state encryption of main memory. As non-volatile memory technologies emerge, systems can reap the benefits of non-volatility while maintaining security.

References

[1] Advanced Micro Devices, Inc. AMD I/O virtualization technology (IOMMU) specification, rev 1.00, Feb. 2006. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf`.

[2] S. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communications Review*, 2(19), Apr. 1989.

[3] P. Broadwell, M. H. N., and Sastry. Scrash: A System for Generating Secure Crash Information. In *Proceedings of the 12th USENIX Security Symposium*, pages 273–284, 2003.

[4] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, 1996.

[5] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving Operating System Crashes. In *ASPLOS*, 1996.

[6] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security Symposium*, 2004.

[7] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *USENIX Security Symposium*, 2005.

[8] G. Duc and R. Keryell. CryptoPage: an Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection. In *ACSAC*, 2006.

[9] EE Times. NEC claims world's fastest MRAM. `http://www.eetimes.com/showArticle.jhtml?articleID=204400328`, November 30, 2007.

[10] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, C. Anguille, C. Buatois, and J. Rigaud. Hardware Engines for Bus Encryption: a Survey of Existing Techniques. In *DATE*, 2005.

[11] Freescale Semiconductor. Fast Non-Volatile RAM Products. `http://www.freescale.com/files/memory/doc/fact_sheet/BRMRAMSLSCLTRL.pdf`, 2007.

[12] Fujitsu. Fujitsu Starts Volume Production of 2 Mbit FRAM Chips. `http://www.fujitsu.com/emea/news/pr/fme_20070418.html`, April 18, 2007.

[13] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *HPCA-9*, 2003.

[14] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *USENIX Security Symposium*, 1996.

[15] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *USENIX Security Symposium*, 2008.

[16] C. Hampel. High-Speed DRAMs keep pace with high-speed systems. `http://www.dewassoc.com/performance/memory/hampel-rambus.htm`. Accessed Jan. 2006.

[17] T. Kgil, L. Falk, and T. Mudge. ChipLock: Support for Secure Microarchitectures. *ACM SIGARCH Computer Architecture News*, 33(1):134–143, Apr. 2005.

[18] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for Protecting Critical Secrets in Microprocessors. In *ISCA*, 2005.

[19] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *ISCA*, 2000.

[20] G. Muller, N. Nagel, C.-U. Pinnow, and T. Rohr. Emerging Non-Volatile Memory Technologies. In *ESSDERC*, 2003.

[21] Physorg. Toshiba develops new MRAM device which opens the way to giga-bits capacity. `http://www.physorg.com/news113591322.html`, November 6, 2007.

[22] J. Platte and E. Naroska. A Combined Hardware and Software Architecture for Secure Computing. In *Proceedings of the 2nd Conference on Computing frontiers*, May 2005.

[23] N. Provos. Encrypting Virtual Memory. In *Proceedings of the 9th USENIX Security Symposium*, Aug. 2000.

[24] B. Rogers, M. Prvulovic, and Y. Solihin. Efficient data protection for distributed shared memory multiprocessors. In *PACT*, 2006.

[25] B. Rogers, Y. Solihin, and M. Prvulovic. Memory Predecryption: Hiding the Latency Overhead of Memory Encryption. *ACM SIGARCH Computer Architecture News*, 33(1):27–33, Mar. 2005.

[26] D. Samyde, S. Skorobogatov, R. Anderson, and J.-J. Quisquater. On a New Way to Read Data from Memory. In *Proceedings of IEEE Security in Storage Workshop*, 2003.

[27] W. Sereinig. Motion-Control: the Power Side of Disk Drives. In *ICCD*, 2001.

[28] W. Shi, H.-H. S. L, M. Ghosh, C. Lu, and A. Boldyreva. High Efficiency Counter Mode Security Architecture via Prediction and Precomputation. In *ISCA*, 2005.

[29] W. Shi, H.-H. S. Lee, M. Ghosh, and C. Lu. Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems. In *PACT*, 2004.

[30] Standard Performance Evaluation Corp. SPEC CPU2000 V1.3. `http://www.spec.org/cpu2000/`, 2000.

[31] G. E. Suh, D. Clarke, B. Gassend, M. van Gijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processors. In *MICRO-36*, 2003.

[32] C. Yan, B. Rogers, D. Englender, Y. Solihin, and M. Prvulovic. Improving cost, performance, and security of memory encryption and authentication. In *ISCA*, 2006.

[33] J. Yang, L. Gao, and Y. Zhang. Improving Memory Encryption Performance in Secure Processors. *IEEE Trans. Comp.*, 54(5):630–640, May 2005.