Silent Shredder: Zero-Cost Shredding for Secure Non-Volatile Main Memory Controllers

Amro Awad
North Carolina State University
ajawad@ncsu.edu

Pratyusa Manadhata
Hewlett Packard Labs
pratyusa.k.manadhata@hpe.com

Stuart Haber
Hewlett Packard Labs
stuart.haber@hpe.com

Yan Solihin
North Carolina State University
solihin@ncsu.edu

William Horne
Hewlett Packard Labs
william.horne@hpe.com

Abstract

As non-volatile memory (NVM) technologies are expected to replace DRAM in the near future, new challenges have emerged. For example, NVMs have slow and power-consuming writes, and limited write endurance. In addition, NVMs have a data remanence vulnerability, i.e., they retain data for a long time after being powered off. NVM encryption alleviates the vulnerability, but exacerbates the limited endurance by increasing the number of writes to memory.

We observe that, in current systems, a large percentage of main memory writes result from data shredding in operating systems, a process of zeroing out physical pages before mapping them to new processes, in order to protect previous processes' data. In this paper, we propose *Silent Shredder*, which repurposes *initialization vectors* used in standard counter mode encryption to completely eliminate the data shredding writes. Silent Shredder also speeds up reading shredded cache lines, and hence reduces power consumption and improves overall performance. To evaluate our design, we run three PowerGraph applications and 26 multi-programmed workloads from the SPEC 2006 suite, on a gem5-based full system simulator. Silent Shredder eliminates an average of 48.6% of the writes in the initialization and graph construction phases. It speeds up main memory reads by 3.3 times, and improves the number of instructions per cycle (IPC) by 6.4% on average. Finally, we discuss several use cases, including virtual machines' data isolation and user-level large data initialization, where Silent Shredder can be used effectively at no extra cost.

Keywords Encryption; Hardware Security; Phase-Change Memory; Data Protection

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '16, April 2–6, 2016, Atlanta, Georgia, USA..
Copyright © 2016 ACM 978-1-4503-4091-5/16/04. . . $15.00.
http://dx.doi.org/10.1145/2872362.2872377

1. Introduction

With the arrival of the era of big data and in-memory analytics, there is increasing pressure to deploy large main memories. The traditional use of DRAM as main memory technology is increasingly becoming less attractive, for several reasons. The need to refresh volatile DRAM cells incurs large power consumption, which limits the amount of DRAM we can deploy in the same package. Second, scaling down DRAM cell size becomes difficult as the charge in DRAM cell capacitor needs to be kept constant to meet the retention time requirements [27]. Considering these DRAM limitations, computer system designers are rightfully considering emerging Non-Volatile Memories (NVMs) as replacements for DRAM. NVMs are non-volatile and require no refresh power; some of them have a read latency comparable to DRAM, while at the same time they may scale better than DRAM [6, 8, 41].

Currently, there are still serious challenges in using NVMs as the main memory. Writing to NVM cells is often slow, requires large power consumption, and has limited write endurance, e.g., 10-100 million writes in Phase Change Memory [6, 8, 30, 41, 45]. Another serious challenge is the data remanence vulnerability, where NVMs retain data for NVMs a long time after a system is powered off whereas DRAM loses data quickly. Obtaining an NVM chip and scanning it can reveal data in memory. Memory encryption has been proposed to ward off data remanence attacks [16, 43]. While effective in dealing with the vulnerability, memory encryption worsens the write endurance problem. A good encryption scheme must have the *diffusion* property; the change in one bit in original data should

change many bits in the encrypted data [36]. However, as pointed out by Young et al. [43], techniques to reduce the number of writes in NVMs, such as Data Comparison Write (DCW) and Flip-N-Write (FNW), lose effectiveness due to diffusion because these techniques are inspired by the observation that few bits will have their values changed after successive cache line writes.

Therefore, with memory encryption, reducing the number of writes is of paramount importance. The goal of this paper is to find opportunities to reduce the number of writes to an encrypted NVM used as main memory. Specifically, we focus on the problem of ***data shredding***, which is one of the most frequent operating system (OS) operations [14]. Data shredding is the strategy to initialize to zero each physical memory page before mapping it to a process [13, 32, 35]. The OS performs shredding to avoid inter-process data leak, where a process accidentally or intentionally reads the old data of another process. Also, hypervisors perform shredding to avoid inter-virtual machine (VM) data leak in virtualized systems. Our experiments show that data shredding can contribute to a large portion of the overall number of main memory writes, significantly costing processor cycles and memory bandwidth. Furthermore, up to 40% of the page fault time can be spent in page zeroing [38].

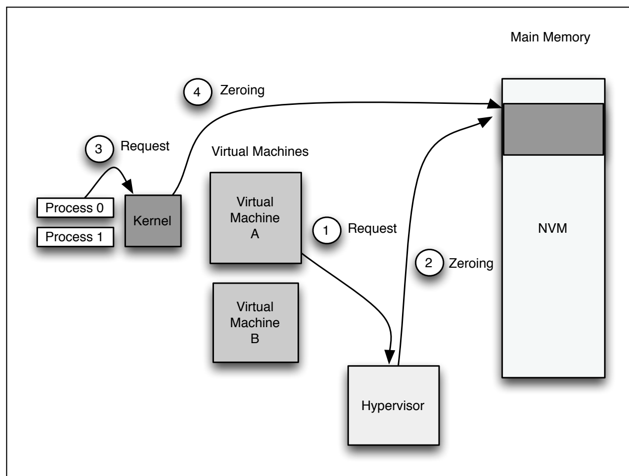

Figure 1. Example of duplicate data sherdding.

The frequency of shredding will further increase where there are many running processes or many virtual machines sharing a system, which is becoming increasingly more common in servers, due to virtualization and server consolidation. Data shredding can also occur multiple times for a single memory page. For example, as shown in Figure 1, hypervisors manage memory allocations across virtual machines in virtualized systems. As shown in Step 1, a VM requests host physical memory pages. However, the hypervisor needs to zero out the data to prevent inter-VM data leak [7, 9], as shown in Step 2. Furthermore, when a process requests memory pages, the Kernel/OS inside the VM zeroes out physical pages before mapping them to a new process to prevent inter-process data leak, as shown in Steps 3 and 4, respectively. The impact of data shredding necessitates that we rethink how data shredding should be implemented.

Virtual machines and local nodes' kernels may prefer to request memory allocations with large granularity, for several reasons. The hypervisor or resource manager does not need to go through a large number of extra page table walks beyond the page table walks that occur at the virtual machine or local node level, hence reducing the cost of address translation misses. Another reason is to reduce the intervention between virtual machines or local nodes and resource managers, hence a better scalability. Accordingly, for instance, assume a system with hundreds of terabytes of memory and a large number of nodes, memory allocations can be in order of gigabytes or terabytes. Zeroing out such large amount of memory would be very slow. Furthermore, there is a chance that the virtual machine or the local node never writes the whole space, but shredding the whole assigned physical space is still required.

Previous work targeted speeding up zeroing pages in the context of DRAM main memory [21, 34, 42]. However, since writes to DRAM are cheap, these studies proposed speeding up the zeroing operation, but not eliminating it. For example, Jiang et al. [21] suggest offloading the zeroing to a DMA engine, while Sehsadri et al. [34] shift the zeroing to within the DRAM. However, in both studies, the writes to the main memory still occur, and hence these methods are not suitable for use in NVM-based main memory.

In this paper, we present a secure NVMM controller, *Silent Shredder*, that *completely* eliminates writes to NVM due to data shredding. Silent Shredder eliminates shredding-related writes to NVM-based main memory by exploiting *initialization vectors (IVs)* used in symmetric encryption. We prevent data leakage and eliminate shredding-related writes at the same time with minimal cost. One key insight for avoiding shredding-related writes is that to protect data privacy of a process, it is sufficient for the OS to make a data page unintelligible when the page is allocated to another process. By making the page unintelligible rather than zeroing it, the old data of a process is protected from being read by another process. The second insight is that when data in a newly allocated page is read, for software compatibility, the controller should skip the actual reading from NVM and instead should supply a zero-filled block to the cache. Silent Shredder achieves these goals by repurposing the IVs: it manipulates IVs to render data pages unintelligible and to encode the shredded state of a page. Note that Silent Shredder works with all modes of symmetric encryption that use IVs, including the counter mode used in recent secure NVM proposals. By avoiding shredding-related writes to NVM and the subsequent reads from NVM, Silent Shredder improves NVM's write endurance, increases read performance, and reduces power consumption. All these are achieved using low-cost

modifications to a standard counter-mode encrypted memory system.

To evaluate our design, we use gem5, a detailed full-system simulator, to run 3 graph analytics applications from the PowerGraph framework and 26 multi-programmed workloads from the SPEC 2006 benchmark suite. Silent Shredder eliminates about *half* (48.6%, on average) of the writes in the initialization and graph construction phases. Furthermore, it speeds up the memory reads by an average of 3.3 times and improves the instructions per cycle (IPC) by an average of 6.4%.

The rest of the paper is organized as follows. We briefly describe NVM technologies, data shredding, and encrypted NVMs in Section 2. In Section 3, we present a motivational example. In Section 4, we introduce several design options for eliminating data shredding and discuss their advantages and disadvantages. Later, we motivate our choice of Silent Shredder as the preferred design and then show that our design can be implemented at zero cost when using secure NVMM controllers. We introduce our evaluation methodology in Section 5 and present evaluation results and a parameter sensitivity study in Section 6. In Section 7, we discuss several use cases, security concerns, and remediation for Silent Shredder. We discuss related work in Section 8 and conclude with a summary in Section 9.

2. Background

In this section, we briefly describe emerging NVM technologies, memory encryption, and data shredding in OSes.

2.1 Non-Volatile Memories (NVMs)

Phase-Change Memory (PCM), Spin-Transfer Torque (STT-RAM) and Memristor are among emerging non-volatile memory technologies, some of which are considered as candidates for replacing DRAM for use as the main memory. Their read latencies and densities are either competitive or comparable with DRAM, and they may scale better than DRAM. However, NVMs suffer from slow and power consuming writes, and generally have limited write endurance, e.g., 10-100 million writes with Phase Change Memory [30]).

When used as main memory, NVMs may provide *persistent memory*, where regular store instructions can be used to make persistent changes to data structures in order to keep them safe from crashes or failures. Persistent memory enables *persistent memory allocations* [11, 23, 24, 39], and may allow future systems to fuse storage and main memory [1, 4, 26]. When an application or a VM requests and uses a persistent page, the OS should guarantee that its page mapping information is kept persistent, so the process or the VM can remap the page across machine reboots [24, 39]. There has been research on explicit software programming frameworks for exploiting persistent memory, e.g., ATLAS [15].

However, NVMM suffers from a serious security vulnerability: it retains data long after a system is powered off. Obtaining physical access to NVMM (through theft, repair, or improper disposal) allows attackers to read the plaintext of data [16, 43]. Accordingly, NVMM should be paired with at least some form of memory encryption.

2.2 Memory Encryption

There are two general approaches for encrypting NVMM. One approach assumes the processor is not modified, and the NVMM controller encrypts the main memory content transparently to the processor [16]. To minimize performance overheads, cold data, i.e, infrequently used data, stays encrypted, but hot data is proactively decrypted and stored in plaintext in memory. Another approach assumes that the processor can be modified and the processor chip is the secure base. Any data sent off chip in the main memory is encrypted. There are many example systems using the latter approach, including some recent studies [31, 40, 43].

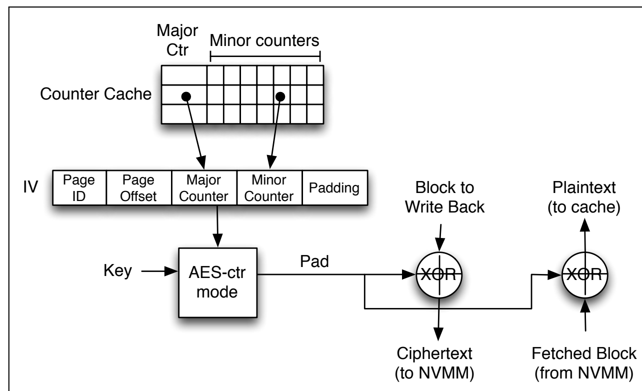

Figure 2. State of the art counter mode encryption. AES is shown for encryption/decryption but other cryptographic algorithms are possible.

There are several encryption modes that can be used to encrypt the main memory. One mode is *direct encryption* (also known as electronic code book or ECB mode), where an encryption algorithm such as AES is used to encrypt each cache block as it is written back to memory and decrypt the block when it enters the processor chip again. Direct encryption reduces system performance by adding decryption latency to the last level cache (LLC) miss latency. Another mode is *counter mode* encryption, where the encryption algorithm is applied to an *initialization vector* (IV) to generate a one-time pad. This is illustrated in Figure 2. Data is then encrypted and decrypted via a simple bitwise XOR with the pad. With counter mode, decryption latency is overlapped with LLC miss latency, and only the XOR latency is added to the critical path of LLC miss. In state-of-the-art design, the IV of a counter mode encryption consists of a unique ID of a page (similar to a page address, but is unique across the main memory and swap space in the disk), page offset

(to distinguish blocks in a page), a per-block *minor* counter (to distinguish different versions of the data value of a block over time), and a per-page *major* counter (to avoid overflow in counter values).

For the rest of the paper, we assume counter mode processor-side encryption, similar to earlier papers [31, 40, 43], because counter mode is more secure: Encryption in ECB mode, without the use of counters or IVs is vulnerable to several attacks based on the fact that identical blocks of plaintext will encrypt to identical blocks of ciphertext, wherever they occur in memory; vulnerabilities include dictionary and replay attacks. Memory-side encryption, e.g. with secure DIMMs, is vulnerable to bus snooping attacks. In addition, processor-side encryption makes it easier to interface with the OS, use per-user and per-application keys, and expose key management to users and applications (where appropriate).

Counters are kept in the main memory, but are also cached on chip with one major counter of a page co-located in a block together with all minor counters of a page. For example, in [40], a 4KB page has a 64-bit major counter that is co-located with 64 7-bit minor counters in a 64-byte block. Any parts of the IV are not secret and only the key is secret. The security of counter mode depends on the pad not being reused, hence at every write back, a block's minor counter is incremented prior to producing a new IV to generate a new pad. When a minor counter overflows, the major counter of the page is incremented and the page is re-encrypted [40]. Finally, counter mode integrity needs to be ensured to avoid attackers from breaking the encryption. While counters are not secret, they must be protected by Merkle Tree to detect any tampering to their values [31].

2.3 Data Shredding

Avoiding data leak between processes requires clearing, e.g., zeroing, a memory page of a process before allocating it to another process. The cost of data shredding is high. For example, a recent study showed that up to 40% of the page fault time is spent in page zeroing [38]. Different operating systems adopt different zeroing strategies. For example, in FreeBSD, the free pages are zeroed out early and kept in a memory pool ready for allocation. Once a process needs a new page to be allocated, a new virtual page is mapped to one of the zeroed physical pages [2]. In Linux, the kernel guarantees that a newly allocated virtual page is initially mapped to a special physical page called the *Zero Page*. The Zero Page is shared by all new allocated pages. The actual allocation of a physical page happens when the process writes to the page for the first write, using a mechanism referred to as copy-on-write (COW). During COW, a physical page (from another process) is zeroed out and the virtual page is remapped to it [13]. Similar mechanisms are deployed by hypervisors to avoid data leak between virtual machines [7].

The zeroing step itself can be implemented with different mechanisms [18]. One mechanism relies on using temporal store instructions, e.g., `movq` in the x86 instruction set, which brings each block into the cache before writing zero to it. The use of temporal store can cause cache pollution, where useful blocks are evicted and cause subsequent cache misses. The impact of cache pollution increases when dealing with large pages, such as 2MB and 1GB size pages. In addition, the mechanism works well if all cache blocks of the page are used shortly after zeroing; however, often this is not the case. Also, without additional mechanisms, zeroing blocks of the page in the cache is not secure as it does not persist the modifications to the NVMM right away. Hence, upon system failures, the zeroing may be lost, and data leak may occur.

Another zeroing mechanism is using non-temporal store instructions, such as `movntq` in x86. Such store instructions bypass the entire cache hierarchy and write directly to the main memory; any blocks that are currently cached are invalidated. Non-temporal stores avoid cache pollution, but they also come with a risk of invalidating soon-to-be accessed or initialized blocks. Some overheads remain unchanged, e.g, the number of store instructions used, and latency and power from using memory bus bandwidth.

There have been proposals to improve zeroing performance. Some studies have proposed offloading the zeroing step to the DRAM controller or the DMA engine, so that the CPU time is not wasted in zeroing pages and cache pollution is avoided [18, 21, 34]. Such techniques are effective in DRAM main memory, but with NVMM, the zeroing still results in high performance overheads due to high power consumption and latency of writes in NVMs, and reduced write endurance. Since our paper targets memory controllers for NVMM, we consider techniques that zero out pages in main memory and bypass the caches entirely, but with a significant reduction in the actual zeroing operations.

3. Motivation

We start with an example to study the impact of kernel shredding on system performance. Assume a simple code that allocates and initializes $SIZE$ bytes of memory, as shown in the code snippet in Figure 3. As explained earlier

```
// Allocating SIZE bytes
char * ALLOC= (char *) malloc(SIZE);
// Setting all allocated memory to 0
// Point 0
memset(ALLOC,0,SIZE);
// Point 1
memset(ALLOC,0,SIZE);
// Point 2
```

Figure 3. A code sample for initializing allocated memory.

in Section 2, the Linux kernel allocates and zeroes out a physical page right after the first write. Accordingly, the first store instruction in the first `memset` causes a page fault,

the page fault handler then allocates a new physical page and zeroes it out, and finally maps it to the application process. As the application's `memset` resumes after the page fault, if the region is the size of a page, page zeroing is redundantly repeated. Overall, the first memset incurs the following delay: kernel page allocation, kernel zeroing, and program zeroing. In contrast, the second memset only has the program zeroing delay. Since program zeroing follows a similar mechanism as kernel zeroing, the second `memset`'s delay is a good proxy of kernel zeroing latency. `memset` is more optimized than the kernel zeroing process- it uses non-temporal stores when the size of the memory to be initialized is bigger than the LLC, and hence avoids cache pollution. Hence it takes less time than kernel zeroing and in our experiments, `memset`'s time is a conservative proxy for kernel zeroing time. Figure 4 shows the time spent to execute the first vs. the second `memset`.

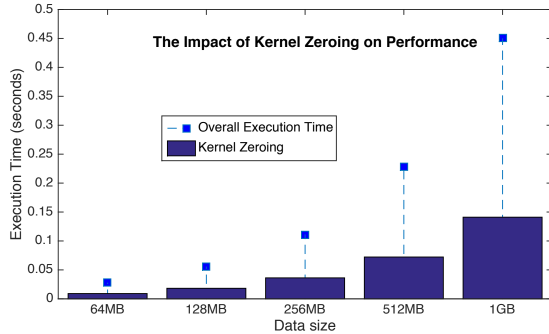

Figure 4. The impact of kernel zeroing on performance.

The overall execution time is the time taken by the first memset (which includes page zeroing, page faults, and program zeroing), and the height of the blue bars is the time taken by the second memset (which only incurs program zeroing). We observe that on average, roughly 32% of the first `memset` time is spent in kernel zeroing. Our observation is consistent with previous studies that showed that up to 40% of the page fault time is spent in kernel zeroing [38]. Note that writing latency for NVMs is multiple times slower than that of DRAM, and hence the page zeroing is expected to become dominant and to contribute for most of the page fault time.

Now let us examine real applications. Figure 5 shows the number of writes to the main memory for several graph analytics applications from the PowerGraph suite [20]. The number of writes was collected using performance counters. A system call was added to collect the number of times the kernel calls the `clear_page` function. For all real system experiments, we use Linux kernel 3.16.2 on a 12-core Intel Xeon E5-2620 CPU machine with 16GB main memory. The data sets we use are publicly released data sets from Netflix and Twitter [10, 44].

For each application, the first bar represents the number of writes when using temporal kernel zeroing (zero-

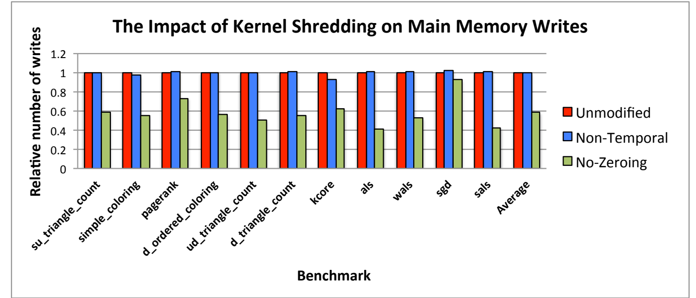

Figure 5. Impact of zeroing on the overall number of writes to the memory.

ing in caches). The second bar represents the number of writes when using non-temporal kernel zeroing (bypassing the cache hierarchy). Finally, the third column shows the number of writes to the main memory when zeroing is avoided in the main memory. We obtained the third bar by counting the number of writes caused by kernel zeroing in the non-temporal case and deducting it from the overall number of writes. We observe that in big data applications, such as graph analytics, a large percentage of the overall number of writes to the main memory is caused by kernel zeroing. The main reason behind this is that many data analytics applications exhibit a write-once read-many times behavior, where the graph is rarely modified after the graph construction phase.

In summary, we found that kernel zeroing can contribute very significantly to the execution time and the number of main memory writes, regardless of whether temporal or non-temporal writes are used. While such overheads are costly with DRAM main memory, they are multiple times more costly with NVMM due to the slow and power consuming nature of writes, and the reduction in NVMs' lifetime until write endurance limit is reached.

4. Silent Shredder Design

In this section, we explore several design alternatives for data shredding and motivate the design choices in Silent Shredder.

4.1 Attack Model

Silent Shredder is designed to protect against attacks carried out by attackers who have physical access to the computers. The attackers can scan a computer's bus or main memory, tamper with main memory content, or play man in the middle between the computer's processor and the main memory. This attack model is the same as the models assumed in prior memory encryption studies [31, 40, 43]. Such attacks are possible due to several reasons such as lax physical security guarding server rooms in cloud computing facilities, improper disposal of computers at the end of their service, and malicious administrators or repairmen. Attackers may

insert a bus snooper or a memory scanner to read out data communicated off the processor chip. They may also insert devices to overwrite the content of the main memory, or act as a man in the middle by returning false data requested by the processor chip. They may conduct known-plaintext attacks by observing the encrypted value of an expected data item, or employ dictionary-based attacks by guessing original values according to encrypted values' frequency.

We leave out side-channel attacks because they are beyond the scope of this paper and there are many side-channel protection mechanisms in the literature that can be added to Silent Shredder as needed.

Silent Shredder requires the OS to communicate page zeroing to hardware. This requires minimal modifications to the OS's page fault handler; however, it requires the OS to be trusted, as an untrusted OS can maliciously avoid page zeroing in order to cause data leak between processes. If the OS is not trusted, then processes must run in secure enclaves. Although beyond the scope of this paper, we believe that Silent Shredder can be adapted relatively easily to work with secure enclaves. For example, the hardware can notify Silent Shredder directly when a page from an enclave is going to be deallocated. Further discussion about several security concerns when using counter-mode encryption and their remediations can be found in Section 7.1.

4.2 Skipping the Zeroing Writes

The main goal of shredding data is to prevent data leakage between processes or VMs when a page is reused. One key observation we make is that page zeroing is only one way to prevent data leakage. Writing any unintelligible or random content into the page before the page is reused achieves the same goal, as the process which gets the page will not be able to read any meaningful data from the page. In traditional (non-encrypted) memory, however, there is no advantage to writing a random pattern to a page compared to writing zeros to the page. In fact, writing zeros to a page is more efficient as various instruction sets may provide a special instruction to zero an entire cache block (e.g., `dcbz` in PowerPC).

However, in encrypted memory, the nature of cryptographic encryption can be exploited to initialize a reused page with a random pattern *for free*. For example, consider a page of data encrypted with the key K_1. Before the page is reused, if we change the key to a different one, K_2, then subsequent decryption of the page will result in unintelligible data. The nature of encryption ensures that the new decrypted data is not correlated with the initial data, nor is it any easier to break encryption when attackers read data decrypted with a different key. Most importantly, the entire initialization writes to the reused page are completely eliminated, resulting in significant performance improvement, power savings, and write endurance improvement.

The above technique can be achieved by giving every process a unique key, instead of the entire system using one key. However, giving each process a unique encryption key seriously complicates key management. First, the architecture and the OS need to ensure that no key is reused and all keys are securely stored. Second, it complicates data sharing among processes: shared pages need to be encrypted/decrypted using a common key that is known to all sharing processes. Third, from a practical perspective, the memory controller has no easy way to identify the process that owns the physical address of the memory read/write request; hence it is unable to decrypt the data without the correct key. Addressing this requires the process ID or the process-specific key to be passed along with every memory request.

These challenges make the deployment of such a design costly. Thus, in Silent Shredder, we assume that processes share a single cryptographic key. In order to provide a way to initialize a reused page with a random pattern without using process-specific keys, we exploit the feature of counter-mode encryption where the initialization vector (IV) is what is encrypted to generate a pad. Essentially, if a block is encrypted with an IV, IV_1, but decrypted with a different IV, IV_2, then even when the two IVs differ by just one bit, the decrypted data is unintelligible. This is because $Data \neq D_K(IV_2) \oplus \big(E_K(IV_1) \oplus Data\big)$ where E_K and D_K are encryption and decryption using the key K. Therefore, to initialize a reused page with unintelligible random data, it is sufficient to modify the page's IVs. Note that this is true of any encryption mode that uses IVs, though we focus on counter mode in this paper.

The next logical question is which parts of the IV should be modified. The page ID and the page offset components in an IV should not be modified as they ensure spatial uniqueness of the IV of a block. This leaves us with the per-page major counter and the per-block minor counter. We now discuss several possible approaches with different trade offs. One option is to increment all the minor counters in a page. This changes the IVs of all blocks in a page and avoids actual page zeroing. However, this option has drawbacks. One drawback is that since minor counters are typically small, e.g., 7 bits are recommended [40], incrementing minor counters induces a high probability of minor counter overflow, which causes expensive page re-encryption. Essentially, a block in a page can only be written back from the LLC to the NVMM $2^7 = 128$ times before the page needs to be re-encrypted. Page re-encryption involves reading all blocks of a page from memory into the cache, incrementing the page's major counter, resetting all minor counters of the page, and writing back all blocks of the page from the cache to the NVMM. Page re-encryption is an expensive operation in DRAM main memory and is significantly more expensive in NVMM. Hence it should be avoided whenever possible.

To avoid increasing the frequency of page re-encryption, we can pursue a second option, where we increment the major counter and leave all minor counters unchanged. Since the major counter of a page is used to encrypt or decrypt all blocks in the page, incrementing the major counter is

sufficient to initialize a reused page to a random pattern. Accordingly, we do not increase the frequency of page re-encryption since minor counters are unchanged.

A major drawback with both of these techniques is that they assume that the OS or applications are not implemented with the expectation that a newly-allocated page will have zero values. However, our observation is that this assumption is not valid in modern systems. For example, in Linux, we find that the `libc` system library's runtime load (`rtld`) code contains an error-checking assertion that verifies that pointer variables in a newly-allocated page have the value of zero (`NULL`). Hence the two techniques give rise to software compatibility challenges.

However, we can pursue another approach when such assertion removal is not possible or not desirable. In this situation, we would still like to avoid zeroing writes but allow Silent Shredder to return a zero value when reading any cache block from a newly allocated page. This requires a third option as follows. When a reused page is initialized, the major counter of the page is incremented, and simultaneously all the minor counters are reset to zero. We reserve the value zero in a block's minor counter to indicate that when the block is read, zeros are returned to the processor instead of a random pattern. The zero minor counter is not used during regular page re-encryption; for example, when a minor counter overflows, the minor counter will be reset to 1 instead of zero. Upon an LLC miss, the minor counter value for the block is checked. If it is zero, then the cache block is not fetched from memory. Instead, a zero-filled block is allocated in the cache, and returned to the `ld` or `st` instruction that accesses the block. This approach has the side benefit of reducing page re-encryption frequency as it lengthens the time until page re-encryption. For the rest of the paper, we use the third option to accomplish page shredding, where the major counter is incremented and minor counters are reset.

Note that without the Silent Shredder mechanism, shredding a memory page in conventional NVMM will cause every minor counter on a page to be incremented, *in addition to* writing zeros to the main memory. Thus, Silent Shredder does not increase the number of writes to counters, but it completely avoids page zeroing writes.

4.3 Silent Shredder's Design

We now discuss Silent Shredder's overall design. Figure 6 illustrates the high-level mechanism. First, when the OS wants to shred a page, p, the OS uses a mechanism to give a hint to the hardware, e.g., by writing p's physical address to a memory-mapped register in the Memory Controller (MC). This is shown in the figure as Step 1. The MC then sends an invalidation request to the cache and coherence controller of remote caches and also the local cache (Step 2). The coherence controller sends out invalidation requests to any blocks in the page that may be shared in the caches of other cores. In addition, the block containing p's counters is also invalidated from other cores' counter caches. This step is necessary for keeping the caches coherent, even though the blocks will not be actually written. When a remote core has a cache miss, it will be forced to reload updated counters for the page and the missing block in the page.

After the invalidation, the counters can be changed (Step 3) by incrementing the major counter of the page p and resetting all the minor counters to zero. Finally, the counter cache controller acknowledges (Step 4) the completion of updating counters, and the MC signals the completion to the processor. At this time, the zeroing is completed without any blocks in the page ever written to the NVMM.

Note that steps 2, 3, 4 and 5 are correctness requirements for any kind of bulk zeroing that bypasses the cache hierarchy, e.g., non-temporal zeroing and DMA engine support. Most modern integrated memory controllers have the ability to invalidate and flush cache lines; DMA region writes should invalidate any stale data in the cache hierarchy [3, 34]. Furthermore, a simple serializing event, e.g., `sfence`, can be implemented to ensure that all invalidations have been posted and that the counter values have been updated. For example, in Linux, non-temporal zeroing stores are followed by `sfence` to guarantee that all writes have been propagated to main memory. A similar approach can be adopted in Silent Shredder, e.g., following shredding the command with `sfence` or `pcommit`, where the memory controller guarantees that serializing events such as `sfence` or `pcommit` will not be marked completed until all invalidations are posted and all counters are updated.

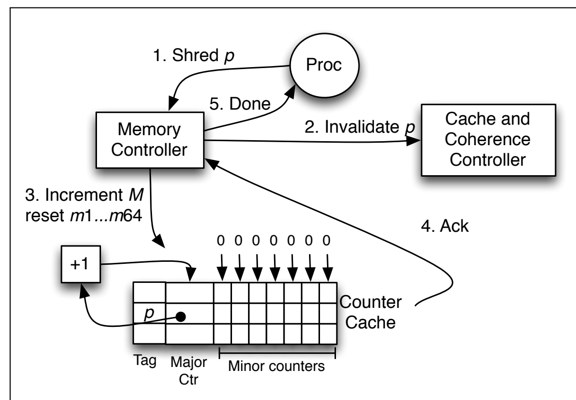

Figure 6. Silent Shredder's shredding mechanism.

Figure 7 shows how a LLC miss is serviced. Suppose the LLC misses on a block, x. The block address of the miss is passed to the counter cache (Step 1), resulting in the minor counter being read out. The minor counter for the block x is then compared to zero. If the minor counter value is not zero, then a request to fetch the block x is passed on to the memory controller (Step 3a), which then obtains x from the NVMM, decrypts it, and returns it to the LLC (Step 4). On the other hand, if the value of the minor counter is zero, then a zero-filled block is returned to the LLC.

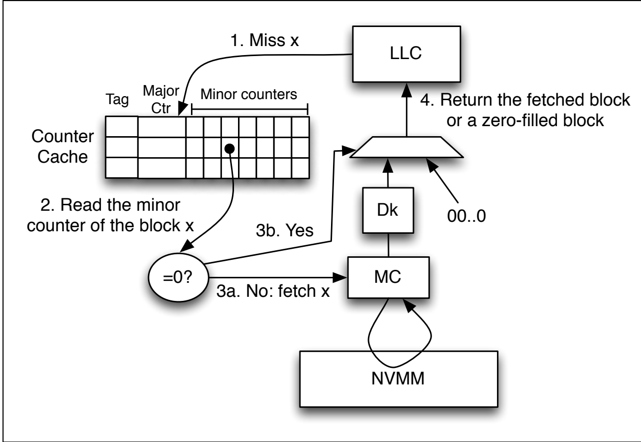

Figure 7. Silent Shredder's mechanism for servicing an LLC miss.

Overall, Silent Shredder's mechanism achieves the following savings compared to traditional page zeroing mechanisms:

1. During page shredding, none of the cache blocks of the page are written.
2. When any of the 64 blocks in a shredded page is read for the first time, the NVMM is not accessed, and instead a zero-filled block is installed in the cache.

Both of these savings improve performance, both during and after shredding. Furthermore, the first item reduces power consumption and improves write endurance of NVMM due to skipping writes.

Silent Shredder's Extensibility: Silent Shredder can be implemented with any encryption mode that uses IVs: CBC, PCBC, CFB, OFB, and CTR. Such encryption modes have the well-known property that encrypted data cannot be decrypted without both the encryption key and the IV that were used to encrypt it. In our scheme, Silent Shredder zeros out the minor counter and increments the major counter in the IV, thereby changing the IV originally used for encryption, making the encrypted data totally irrecoverable.

Counter Cache Persistency: Similar to any scheme that relies on IVs, Silent Shredder needs to maintain the counter cache's persistency by using either a battery-backed write-back counter cache, or a write-through counter cache. For the rest of the paper, we assume a battery-backed write-back counter cache. However, even in case of write-through counter cache, the overhead for counter cache is minimal per page zeroing (64B block per 4096B page write).

Data Recovery: In our case, recovering data in case of disasters is the same as with secure DIMMs. For security reasons, encryption keys need to be stored outside the DIMM or stored securely in the DIMM, e.g. encrypted with a master recovery key. Similar mechanisms to recover keys apply to both cases. In addition, the IVs must be backed up to NVMM. Once the keys and the IVs are obtained, the data can be recovered. In addition, redundancy and key sharing can be used for recovery.

5. Evaluation Methodology

We use a modified version of Linux kernel 3.4.91 in our experiments. Specifically, we replace the `clear_page` function used to clear new physical pages with our shredding mechanism. We use a memory-mapped I/O register for writing the physical address of the page to be shredded. We use the default page size of 4KB. Shredding any page larger than 4KB, such as 2MB and 1GB, can be done by sending a shred command for each 4KB of the large page. In Linux, the function for clearing large pages, `clear_huge_page`, already calls the `clear_page` function for each 4KB. Therefore, no more modifications are needed. We run our modified Linux kernel on gem5, a full system cycle-accurate simulator [12]. We extended gem5's memory controller to include a counter cache and a memory-mapped I/O register. We run 26 SPEC CPU2006 benchmarks with the reference input [5]. We also run 3 benchmarks from PowerGraph benchmark suite: page rank, simple coloring and kcore [20]. All benchmarks were compiled with default settings and optimizations.

Our goal is to study the potential benefits for Silent Shredder under environments with high level of data shredding as a proof of concept. Data shredding is a performance bottleneck during graph construction and initialization phases. Hence we checkpoint the PowerGraph benchmarks at the beginning of the graph construction phase, and the SPEC benchmarks at the beginning of the initialization phase. In the baseline configuration without Silent Shredder, we assume that when receiving a shred command, shredded cache blocks are invalidated and written back (if dirty) to the main memory. In other words, the baseline shredding uses non-temporal stores. Such an assumption guarantees that similar number of instructions are executed when comparing Silent Shredder with the baseline; similar kernel binary and checkpoints are used. For fair comparison, we assume that for both the baseline and Silent Shredder, invalidation of all cache blocks is sent right after receiving the shredding command.

We warm up the system caches for 1 billion cache accesses and then run the applications until at least one core has executed 500 million instructions (a total of approximately 4 billion instructions for an 8-core system). For SPEC benchmarks, we run an instance of the benchmark on each core.

The system configuration used in our simulations is shown in Table 1. Similar to the expected trend of modern cache hierarchies, our system has a 4-level cache hierarchy [37]. L4 and L3 caches are shared among all cores, while L1 and L2 caches are private for each core. We also assume a battery-backed writeback counter cache.

Each valid cache block in the counter cache maps to a single physical page in the main memory. Each block contains 7-bit minor counters, one for each cache block in

Table 1. Configurations of the baseline system.

Processor	
CPU	8 cores x86-64 processor, 2GHz clock
L1 Cache	2 cycles, 64KB size, 8-way, LRU, 64B block size
L2 Cache	8 cycles, 512KB size, 8-way, LRU, 64B block size
L3 Cache	25 cycles, 8MB size, 8-way, LRU, 64B block size
L4 Cache	35 cycles, 64MB size, 8-way, LRU, 64B block size
Coherency Protocol	MESI
Main Memory	
Capacity	16 GB
# Channels	2 channels
Channel bandwidth	12.8 GB/s
Read Latency	75 ns
Write Latency	150 ns
Counter Cache	10 cycles, 4MB size, 8-way, 64B block size
Operating System	
OS	Gentoo
Kernel Version	3.4.91

that page, and a 64-bit major counter for that page. In Section 6.4, we discuss the reason behind our choice of the capacity of the counter cache to be 4MB. The latency of the counter cache was obtained using the CACTI 6.0 tool [25].

6. Evaluation

In this section, we evaluate the potential benefits that come from our design. We start by showing the reduction of the number of main memory writes as a result of using Silent Shredder. Later, we show how our design also reduces read traffic and improves performance as additional benefits.

6.1 Write Reduction

The main goal of Silent Shredder is to reduce the number of writes occurring as a result of the kernel zeroing process for shredding previous data. We find that an average of 48.6% of the main memory writes in the initialization phase could be eliminated. Figure 8 shows the percentage of writes to the main memory that we could save by using Silent Shredder.

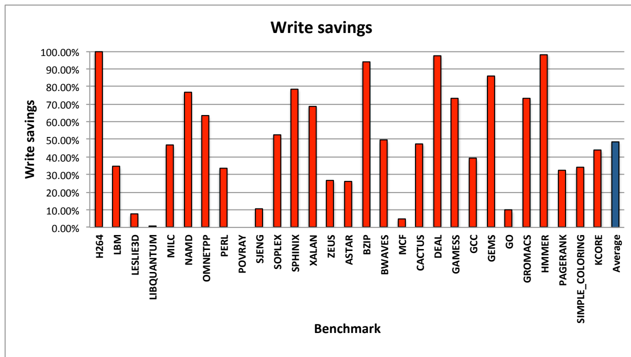

Figure 8. Savings on writes to the main memory.

From Figure 8, we observe that kernel zeroing contributes to large percentage of main memory writes for some applications. A few applications, e.g., H264, DealII and Hmmer, have a very small ratios of main memory writes to instructions. Accordingly, we find that for such applications the majority of main memory writes result from the kernel zeroing operation. Note that since we simulate the full system, the kernel zeroing can be a result of the application initialization code itself, loading system libraries, writing metadata or any other system activities. Since our work targets shredding from the kernel perspective, we count all kernel shredding writes, regardless of the semantics of the program.

Eliminating a significant number of writes is very important; it reduces the memory traffic, increases performance by eliminating slow and power consuming NVMM writes, and increases the lifetime of the NVMM. Our evaluation focuses on the start up or graph initialization phase which is important because the graphs get rarely modified after initialization, hence significant percentage of the main memory writes occur at this phase. The startup phase is not the only scenario where Silent Shredder is effective. In a system that is highly loaded, data shredding will occur frequently because the high load from multiple workloads are placing a high pressure on the physical memory. For example, most data center servers today are not very good in terms of energy proportionality (idle power is approximately 50% of peak power), and therefore peak energy efficiency is achieved when the data centers are highly loaded resulting in very high processor utilization rates [19]. A highly loaded system will suffer from a high rate of page faults, and page fault latency is critical in this situation.

6.2 Read Traffic Reduction

Another important advantage of using Silent Shredder is the ability to recognize shredded cache blocks from their minor counter values. Once a read to a shredded block is detected, a zero-filled block will be returned to the cache hierarchy without the need to read the invalid data from the main memory. We find a surprisingly large percentage of the initialization read traffic can be eliminated.

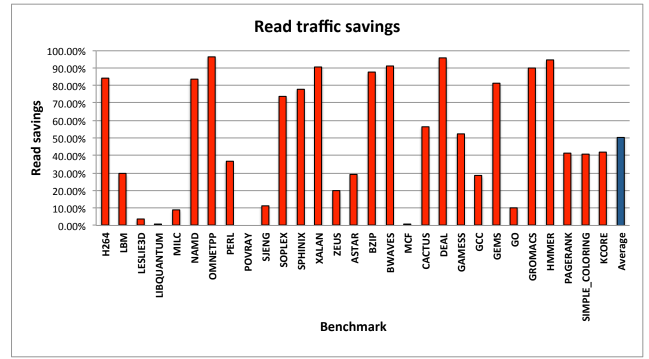

Figure 9. Percentage of the saved main memory read traffic.

As shown in Figure 9, an average of 50.3% of the read traffic during the initialization phase is due to reading shredded pages. Reducing the read traffic is crucial for systems that run memory-intensive benchmarks with limited memory bus bandwidth. Identifying that a block is shredded happens quickly: any shredded block read request can be completed

once its minor counter value is read. Figure 10 shows the speed up ratio of the average memory read latency for each benchmark. On average, Silent Shredder achieves an average memory read speed up ratio of $3.3\times$ across the benchmarks.

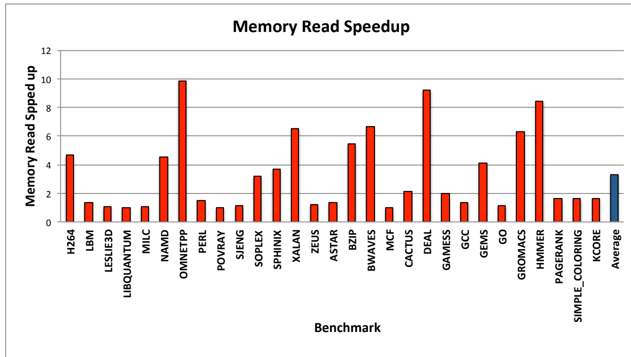

Figure 10. The speed up of the main memory reads.

6.3 Overall Performance Improvement

As mentioned earlier, in addition to eliminating shredding writes, Silent Shredder also speeds up reading shredded cache lines. These two improvements together improve the IPC. Figure 11 shows the relative IPC when using Silent Shredder (higher is better), normalized to the baseline.

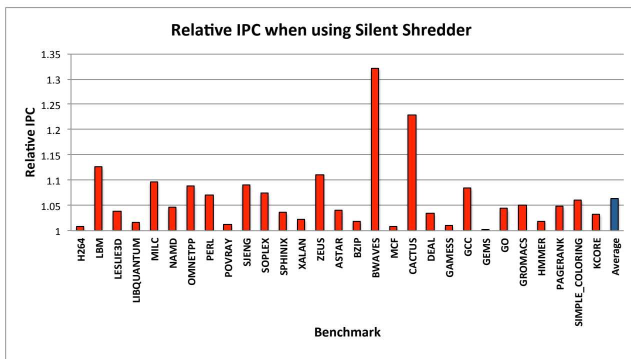

Figure 11. Overall IPC improvement.

We observe an average IPC improvement of 6.4% among all benchmarks, with a maximum of 32.1% for `Bwaves`. The overall IPC improvement depends on the fraction of instructions that access either main memory or shredded cache blocks.

6.4 Counter Cache Size

The IV cache should be fast enough so that encryption, specifically OTP generation, starts as soon as possible. Accordingly, the IV cache needs to be as small as possible while achieving a high hit rate.

From Figure 12, we observe that increasing the cache size beyond 4MB has little impact on reducing the cache miss rate. However, increasing the cache size brings significant reduction in the cache miss rate until reaching a capacity of 4MB. Accordingly, we believe that a size of 4MB is the smallest cache size that achieves a reasonable cache miss rate.

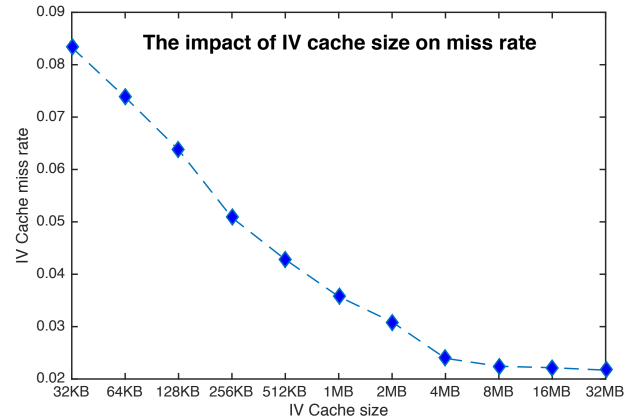

Figure 12. The impact of cache size on miss rate.

7. Discussion

In this section, we start by discussing several security concerns when using Silent Shredder and counter-mode encryption in general, and describe possible remediations. We then discuss different use cases and applications that can benefit from Silent Shredder at no extra cost.

7.1 Security Concerns

Several security concerns can arise when using counter-mode encryption in general. Other concerns arise when relying on hardware to achieve operating system privileged tasks. In this section, we discuss the concerns and their remediations.

- ***Tampering with Memory and Encryption Counter Values***: Since data is already encrypted, tampering with the memory values can cause unpredictable behavior. However, tampering with or replaying the counter values can be a security risk. While out of the scope of this paper, to avoid such risk, techniques such as Bonsai Merkle Trees can be used for authenticating the counter values with low overhead (about 2%) [31, 40]. The only information that could be leaked to an adversary is whether a page is shredded or not; a minor counter of 0 will indicate shredding, however, such information is useless. If we want to hide such information, the only modification needed is to encrypt the IV counters when writing to the main memory.
- ***Losing Counter Cache Values in Case of Power Loss***: In counter-mode encryption, the counter cache must be backed up to NVMM whenever there is a crash or power

loss. Hence the counter cache must be battery-backed and guaranteed to be persistent. In addition, the kernel should have a mechanism to verify the integrity of the stored counter values. If the integrity has been violated, then the kernel should report an error and stop execution of the system. Further error handling mechanisms are needed to recover from this error, which are beyond the scope of this paper.

Another way for achieving persistence of the counter cache would be a write-through implementation, i.e., any update to the counter cache is also updated in NVMM. In the latter case, non-temporal stores or DMA engine zeroing write zeros to NVMM *in addition to* the counter cache block. However, in Silent Shredder only the counter cache block is written to NVMM.

- ***Using the Shred Command from User Space***: As explained earlier, the Shred command hints the memory controller with a physical page address to shred. Allowing user-space applications to issue shred commands can be a security risk. Accordingly, we implement the shred command so that it can be executed only by the kernel (kernel-mode) and only when the kernel allocates a new physical page to a process. Any attempt to write the memory-mapped I/O register of the memory controller from a user-space process will cause an exception.

7.2 Other Use Cases for Silent Shredder

While our main goal in this paper is data shredding, Silent Shredder can be used in other areas.

- ***Virtual Machine Isolation in Virtualized Environments***: In VMs, zeroing out pages is used heavily to shred any physical page before allocating to a VM [7]. This zeroing happens in addition to the kernel shredding occurring inside the VM to protect applications' data. VMs typically request large pools of pages and prefer large pages due to the reduced translation overhead; large pages skip one or more levels of translation and hence speed up the page table walk process [29]. Furthermore, large allocations can reduce the number of hypervisor interruptions. Silent Shredder can speed up the VM requests by completely avoiding zeroing out pages. In case of highly-shared systems, the hypervisor continuously reclaims physical memory pages from VMs in a process called memory ballooning, in order to be able to satisfy requests from VMs with higher memory demand [7]. Accordingly, data shredding will be very frequent and hence Silent Shredder can be of substantial use.
- ***Initializing Large Blocks of Memory***: Some applications tend to zero out large amount of allocated memory, e.g., a sparse matrix, as part of the initialization phase. Such initialization can benefit from the hardware support provided by Silent Shredder. Applications would invoke a system call to provide the kernel with the starting virtual address and the number of pages to be zeroed out. The system call would be translated by the kernel shred command with the corresponding physical address.
- ***Zero Initialization in Managed Programming Languages***: In managed languages, such as Java and C#, the language specifications require initializing new objects with zeros. Similarly, unmanaged native languages, such as C++, have started using zero initialization to improve memory safety [28]. Silent Shredding can be used to zero initialize allocated memory pages with low overhead. It only requires a system call to the kernel, providing it with the virtual address of the page(s) desired for zero initialization. The kernel will simply submit a shred command with the physical address of each page to be zero initialized.

8. Related Work

In this section, we discuss prior work related to our work. Since our work is interdisciplinary and has intersections with different research problems, we summarize previous work by the problems they addressed.

Data Shredding: Data shredding is the process of erasing data previously written by a process to prevent interprocess data leak. Chow et al. [18] discuss the importance of shredding data during page deallocation and investigate both temporal and non-temporal ways to zero out pages for shredding purposes. Currently, modern operating systems deploy shredding by zeroing out pages [2, 13, 32, 35]. Our work achieves the shredding process at zero-cost by eliminating the need to write to the main memory.

Improving Initialization Performance: Jiang et al. [21] suggest offloading the zeroing process to a dedicated DMA engine close to the memory controller. Their design reduces both cache pollution and wasted processor time. Seshadri et al. [34] observe that significant memory bus bandwidth could be wasted by initialization. Accordingly, they suggest the zeroing operation to occur within DRAM by dedicating a zero row inside each DRAM subarray. Their design, RowClone, efficiently reduces the fraction of bus bandwidth used for initialization. However, in both the previously mentioned work, writes to the memory cells still occur. Lewis et al. [22] propose reducing the memory access required to fetch uninitialized blocks on a store miss by using a specialized cache to keep track of uninitialized regions of memory. While their work and Silent Shredder have similar advantages, our design's main goal is to efficiently protect applications' data. Speeding up applications and reducing memory traffic are additional advantages of the way we shred data. Yang et al. [42] study the performance implications when using temporal versus non-temporal stores for zeroing data. Sartor et al. [33] propose new ISA instructions to speed up zeroing in managed languages. Their solution, however, requires that the kernel zeros out processes' new physical pages in main

Table 2. Comparison between Silent Shredder and other initialization techniques.

Mechanism	Features					
	No Cache Pollution	Low Processor Time	Fast to Read/Write	No Memory Writes	Persistent	No Memory Bus Writes
Non-temporal stores	✓	✗	✗	✗	✓	✗
Temporal stores	✗	✗	✓	✗(indirectly)	✗	✗(indirectly)
DMA bulk zeroing engine	✓	✓	✗	✗	✓	✗
RowClone (DRAM-specific)	✓	✓	✗	✗	✓	✓
Silent Shredder	✓	✓	✓	✓	✓	✓

memory; hence Silent Shredder can be of great importance when using such a scheme. Table 2 presents a brief summary of the features of the most related initialization techniques. Note that for both temporal and non-temporal stores, the processor needs to execute a large `for` loop of `movq/movntq` instructions to zero out the whole page. In the case of temporal stores, it can also pollute caches and indirectly increase the main memory writes, as shown in Section 3. Furthermore, Silent Shredder enables fast read/write operations for the initialized blocks; shredded lines can be quickly recognized from their IV values. Silent Shredder also achieves persistent shredding, because the IV values must be backed up in case of power failure, otherwise, recovering the data will be impossible.

Secure Non-Volatile Memory Controllers: Chhabra and Solihin [16] propose encrypting cache lines when writing to the main memory. Their design aims to efficiently avoid revealing data when NVMM chips get stolen. However, their implementation does not protect from bus-snoop, dictionary-based and replay attacks. Later, Young et al. [43] propose a design that uses counter-mode based encryption that eliminates bus-snoop attacks, replay attacks, dictionary-based attacks, and physical NVM access attacks. Their design, DEUCE, uses AES counter mode for encrypting cache lines before writing back to the main memory. Our secure NVMM controller is based on a design similar to the one in DEUCE. Accordingly, it prevents all the previously mentioned attacks.

Write-aware Non-Volatile Memory Controllers: Writes to NVM-based main memory are expensive in terms of latency, energy requirements, and lifetime reduction. Accordingly, many previous work have targeted reducing the actual number of writes to NVM-based main memory [30, 45]. Qureshi et al. [30] propose to dynamically change the bound of the main memory using start and gap registers to uniformly distribute writes across memory cells. Later, Zhou et al. [45] suggest using differential writes, i.e., first read the previous data, compare it with the new data, and then only write the cells whose values have changed. Cho and Lee [17] propose Flip-N-Write technique, their technique suggests that memory words can be written in a flipped manner to reduce the number of bits having their value changed and hence reducing the number of actual writes. In the context of secure NVMM controllers, Young et al. [43] observe that techniques such as Flip-N-Write and differential writes are inefficient in presence of encryption. They propose to reduce the number of writes by avoiding re-encrypting the unmodified partitions of a cache line. Our work is orthogonal and can be easily integrated with their design, DEUCE. DEUCE targets reducing the need for changing cells' values when a cache line write is certain to occur. However, Silent Shredder eliminates the cache line writes completely when shredding new pages.

9. Conclusion

NVM technologies are serious contenders for replacing DRAM as main memory in the near future. However, they face two key challenges for widespread adoption: limited write endurance and data remanence vulnerability. NVM encryption alleviates the data remanence vulnerability, but exacerbates the endurance challenge by increasing the number of main memory writes. In this paper, we propose an approach to reduce the number of writes to encrypted NVM by completely eliminating the writes occurring due to OS data shredding. Our approach, Silent Shredder, manipulates counter mode encryption IV values to render memory pages unintelligible and hence obviates the writing of zeros to memory pages. For software compatibility, we encode IV values in a manner so that Silent Shredder quickly identifies shredded blocks from their IV values and then returns a zero block instead of returning unintelligible data stored in shredded pages. As a byproduct, this process also speeds up reading shredded cache lines. Hence Silent Shredder reduces the number of writes and speeds up reading in encrypted NVM, and hence reduces power consumption and improves overall system performance. We believe that encryption will be commonly used in NVMM for data protection; hence our Silent Shredder approach can contribute significantly to NVMM adoption by increasing NVMM lifetime and improving overall system performance.

Acknowledgments

A large portion of this work was done when Amro Awad was an intern at HP Labs and in collaboration with his Ph.D. advisor Yan Solihin. We would like to thank the anonymous reviewers for their generous feedback.

References

[1] Intel 3D XPoint. URL http://newsroom.intel.com/docs/DOC-6713.

[2] FreeBSD. URL http://www.freebsd.org/doc/en_US.ISO8859-1/articles/vm-design/prefault-optimizations.html.

[3] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3a, Chapter 11, Page 12. April 2012.

[4] The Machine: A new kind of computer. URL `http://www.hpl.hp.com/research/systems-research/themachine/`.

[5] SPEC, SPEC CPU2000 and CPU2006, http://www.spec.org/.

[6] Huai, yiming, et al. "observation of spin-transfer switching in deep submicron-sized and low-resistance magnetic tunnel junctions." Applied Physics Letters 84.16: 3118-3120, 2004.

[7] *Understanding Memory Resource Management in VMware vSphere® 5.0*.

[8] Exploring high-performance and energy proportional interface for phase change memory systems. *the proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA-20), 2013*, 0:210–221, 2013. ISSN 1530-0897. .

[9] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.80 edition, May 2014.

[10] J. Bennett and S. L. and. The Netflix Prize. In *In KDD Cup and Workshop in conjunction with KDD*, 2007.

[11] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Implications of CPU caching on byte-addressable non-volatile memory programming. Technical report, 2012.

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011. ISSN 0163-5964. . URL `http://doi.acm.org/10.1145/2024716.2024718`.

[13] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. ISBN 0596005652.

[14] M. Calhoun, S. Rixner, and A. L. Cox. Optimizing kernel block memory operations. In *IEEE 4th Workshop on Memory Performance Issues, Feb. 2006.*

[15] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *the proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 433–452. ACM, 2014.

[16] S. Chhabra and Y. Solihin. i-NVMM: A Secure Non-volatile Main Memory System with Incremental Encryption. In *"the proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA-38), 2011*, ISCA '11, pages 177–188, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0472-6. . URL `http://doi.acm.org/10.1145/2000064.2000086`.

[17] S. Cho and H. Lee. Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance. In *the proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009. MICRO-42*, pages 347–357, Dec 2009.

[18] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *the proceedings of the 14th Conference on USENIX Security Symposium (SSYM-14), 2005*, SSYM'05, pages 22–22, Berkeley, CA, USA, 2005. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251398.1251420`.

[19] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *the proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA), 2007*, ISCA '07, pages 13–23, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. . URL `http://doi.acm.org/10.1145/1250662.1250665`.

[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *the proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI-10), 2012*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL `http://dl.acm.org/citation.cfm?id=2387880.2387883`.

[21] X. Jiang, Y. Solihin, L. Zhao, and R. Iyer. Architecture support for improving bulk memory copying and initialization performance. In *the proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT-18), 2009*, PACT '09, pages 169–180, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3771-9. . URL `http://dx.doi.org/10.1109/PACT.2009.31`.

[22] J. Lewis, B. Black, and M. Lipasti. Avoiding initialization misses to the heap. In *the proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA-29), 2002*, pages 183–194, 2002. .

[23] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang. NVM Duet: Unified working memory and persistent store architecture. In *ACM SIGPLAN Notices*, volume 49, pages 455–470. ACM, 2014.

[24] I. Moraru, D. G. Andersen, M. Kaminsky, N. Binkert, N. Tolia, R. Munz, and P. Ranganathan. Persistent, protected and cached: Building blocks for main memory data stores. *Technical Report CMU-PDL-11-114*, 2012.

[25] N. Muralimanohar and R. Balasubramonian. Cacti 6.0: A Tool to Model Large Caches.

[26] O. Mutlu and L. Subramanian. Research Problems and Opportunities in Memory Systems. *Invited Article in Supercomputing Frontiers and Innovations (SUPERFRI), 2015.*, pages 32–34.

[27] P. J. Nair, D.-H. Kim, and M. K. Qureshi. Archshield: Architectural framework for assisting dram scaling by tolerating high error rates. In *the proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA-40), 2013*, ISCA '13, pages 72–83, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. . URL `http://doi.acm.org/10.1145/2485922.2485929`.

[28] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *the proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2007*. Press, 2007.

[29] B. Pham, A. Bhattacharjee, Y. Eckert, and G. Loh. Increasing tlb reach by exploiting clustering in page translations. In *the proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), 2014*, pages 558–567, Feb 2014. .

[30] M. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-based Main Memory with Start-Gap Wear Leveling. In *the proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42), 2009*, pages 14–23, Dec 2009.

[31] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly. In *the proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40), 2007*, MICRO 40, pages 183–196, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3047-8. . URL `http://dx.doi.org/10.1109/MICRO.2007.44`.

[32] M. Russinovich and D. A. Solomon. *Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition*. Microsoft Press, 5th edition, 2009. ISBN 0735625301, 9780735625303.

[33] J. B. Sartor, W. Heirman, S. M. Blackburn, L. Eeckhout, and K. S. McKinley. Cooperative cache scrubbing. In *the proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT-23), 2014*, pages 15–26. ACM, 2014.

[34] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization. In *the proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46), 2013*, MICRO-46, pages 185–197, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2638-4. . URL `http://doi.acm.org/10.1145/2540708.2540725`.

[35] A. Singh. *Mac OS X Internals*. Addison-Wesley Professional, 2006. ISBN 0321278542.

[36] W. Stallings. *Cryptography and Network Security (6th ed.)*. 2014.

[37] J. Stuecheli. Power8. In *Hot Chips. Vol. 25. 2013.*, 2013.

[38] S. Valat, M. Pérache, and W. Jalby. Introducing kernel-level page reuse for high performance computing. In *the proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, page 3. ACM, 2013.

[39] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGPLAN Notices*, 46 (3):91–104, 2011.

[40] C. Yan, B. Rogers, D. Englender, D. Solihin, and M. Prvulovic. Improving cost, performance, and security of memory encryption and authentication. In *the proceedings of the 33rd International Symposium on Computer Architecture (ISCA-33), 2006*, pages 179–190, 2006. .

[41] J. J. Yang, D. B. Strukov, and D. R. Stewart. Memristive devices for computing. *Nature nanotechnology*, 8(1):13–24, 2013.

[42] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why nothing matters: The impact of zeroing. In *the proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), 2011*, OOPSLA '11, pages 307–324, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. . URL `http://doi.acm.org/10.1145/2048066.2048092`.

[43] V. Young, P. J. Nair, and M. K. Qureshi. DEUCE: Write-Efficient Encryption for Non-Volatile Memories. In *the proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-20), 2015*, ASPLOS '15, pages 33–44, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. . URL `http://doi.acm.org/10.1145/2694344.2694387`.

[44] R. Zafarani and H. Liu. Social computing data repository at ASU, 2009. URL `http://socialcomputing.asu.edu`.

[45] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *the proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA-36), 2009*, ISCA '09, pages 14–23, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. . URL `http://doi.acm.org/10.1145/1555754.1555759`.