StealthMem: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud

Taesoo Kim — *MIT CSAIL*
Marcus Peinado — *Microsoft Research*
Gloria Mainar-Ruiz — *Microsoft Research*

Abstract

Cloud services are rapidly gaining adoption due to the promises of cost efficiency, availability, and on-demand scaling. To achieve these promises, cloud providers share physical resources to support multi-tenancy of cloud platforms. However, the possibility of sharing the same hardware with potential attackers makes users reluctant to off-load sensitive data into the cloud. Worse yet, researchers have demonstrated side channel attacks via shared memory caches to break full encryption keys of AES, DES, and RSA.

We present StealthMem, a system-level protection mechanism against cache-based side channel attacks in the cloud. StealthMem manages a set of locked cache lines per core, which are never evicted from the cache, and efficiently multiplexes them so that each VM can load its own sensitive data into the locked cache lines. Thus, any VM can hide memory access patterns on confidential data from other VMs. Unlike existing state-of-the-art mitigation methods, StealthMem works with existing commodity hardware and does not require profound changes to application software. We also present a novel idea and prototype for isolating cache lines while fully utilizing memory by exploiting architectural properties of set-associative caches. StealthMem imposes 5.9% of performance overhead on the SPEC 2006 CPU benchmark, and between 2% and 5% overhead on secured AES, DES and Blowfish, requiring only between 3 and 34 lines of code changes from the original implementations.

1 Introduction

Cloud services like Amazon's Elastic Compute Cloud (EC2) [5] and Microsoft's Azure Service Platform (Azure) [26] are rapidly gaining adoption because they offer cost-efficient, scalable and highly available computing services to their users. These benefits are made possible by sharing large-scale computing resources among a large number of users. However, security and privacy concerns over off-loading sensitive data make many end-users, enterprises and government organizations reluctant to adopt cloud services [18, 20, 25].

To offer cost reductions and efficiencies, cloud providers multiplex physical resources among multiple tenants of their cloud platforms. However, such sharing exposes multiple side channels that exist in commodity hardware and that may enable attacks even in the absence of software vulnerabilities. By exploiting side channels that arise from shared CPU caches, researchers have demonstrated attacks extracting encryption keys of popular cryptographic algorithms such as AES, DES, and RSA. Table 1 summarizes some of these attacks.

Unfortunately, the problem is not limited to cryptography. Any algorithm whose memory access pattern depends on confidential information is at risk of leaking this information through cache-based side channels. For example, attackers can detect the existence of `sshd` and `apache2` via a side channel that results from memory deduplication in the cloud [38].

There is a large body of work on countermeasures against cache-based side channel attacks. The main directions include the design of new hardware [12, 23, 24, 41–43], application specific defense mechanisms [17, 28, 30, 39] and compiler-based techniques [11]. Unfortunately, we see little evidence of general hardware-based defenses being adopted in mainstream processors. The remaining proposals often lack generality or have poor performance.

We solve the problem by designing and implementing a system-level defense mechanism, called StealthMem, against cache-based side channel attacks. The system (hypervisor or operating system) provides each user (virtual machine or application) with small amounts of memory that is largely free from cache-based side channels. We first design an efficient software method for locking the pages of a virtual machine (VM) into the shared cache, thus guaranteeing that they cannot be evicted by other VMs. Since different processor cores might be running

Type	Enc.	Year	Attack description	Victim machine		Samples	Crypt. key
Active Time-driven [9]	AES	2006	Final Round Analysis	UP	Pentium III	$2^{13.0}$	Full 128-bit key
Active Time-driven [30]	AES	2005	Prime+Evict (Synchronous Attack)	SMP	Athlon 64	$2^{18.9}$	Full 128-bit key
Active Time-driven [40]	DES	2003	Prime+Evict (Synchronous Attack)	UP	Pentium III	$2^{26.0}$	Full 56-bit key
Passive Time-driven [4]	AES	2007	Statistical Timing Attack (Remote)	SMT	Pentium 4 with HT	$2^{20.0}$	Full 128-bit key
Passive Time-driven [8]	AES	2005	Statistical Timing Attack (Remote)	UP	Pentium III	$2^{27.5}$	Full 128-bit key
Trace-driven [14]	AES	2011	Asynchronous Probe	UP	Pentium 4 M	$2^{6.6}$	Full 128-bit key
Trace-driven [29]	AES	2007	Final Round Analysis	UP	Pentium III	$2^{4.3}$	Full 128-bit key
Trace-driven [3]	AES	2006	First/Second Round Analysis	-	-	$2^{3.9}$	Full 128-bit key
Trace-driven [30]	AES	2005	Prime+Probe (Synchronous Attack)	SMP	Pentium 4 with HT	$2^{13.0}$	Full 128-bit key
Trace-driven [32]	RSA	2005	Asynchronous Probe	SMT	Xeon with HT	-	310-bit of 512-bit key

Table 1: Overview of cache-based side channel attacks: UP, SMT and SMP stand for uniprocessor, simultaneous multithreading and symmetric multiprocessing, respectively.

different VMs at the same time, we assign a set of locked cache lines to each core, and keep the pages of the currently running VMs on those cache lines. Therefore each VM can use its own special pages to store sensitive data without revealing its usage patterns. Whenever a VM is scheduled, StealthMem ensures the VM's special pages are loaded into the locked cache lines of the current core. Furthermore, we describe a method for locking pages without sacrificing utilization of cache and memory by exploiting an architectural property of caches (set associativity) and the cache replacement policy (pseudo-LRU) in commodity hardware.

We apply this locking technique to the last level caches (LLC) of modern x64-based processors (usually the L2 or L3 cache). These caches are particularly critical as they are typically shared among several cores, enabling one core to monitor the memory accesses of other cores. StealthMem prevents this for the locked pages. The LLC is typically so large that the fraction of addresses that maps to a single cache line is very small, making it possible to set aside cache lines without introducing much overhead. In contrast, the L1 cache of a typical x64 processor is not shared and spans only a single 4 kB page. Thus, we do not attempt to lock it.

We use the term "locking" in a conceptual sense. We have no hardware mechanism for locking cache lines on mass market x64 processors. Instead, we use a hypervisor to control memory mappings such that the protected memory addresses are guaranteed to stay in the cache, irrespective of the sequence of memory accesses made by software. While the cloud was our main motivation, our techniques are not limited to the cloud and can be used to defend against cache-based side channel attacks in a general setting.

Our experiments show that our prototype of the idea on Windows Hyper-V efficiently mitigates cache-based side channel attacks. It imposes a 5.9% performance overhead on the SPEC 2006 CPU benchmark running with 6 VMs. We also adapted standard implementations of three common block ciphers to take advantage of StealthMem. The code changes amounted to 3 lines for Blowfish, 5 lines for DES and 34 lines for AES. The overheads of the secured versions were 3% for DES, 2% for Blowfish and 5% for AES.

Level	Shared	Type	Line size	Assoc.	Size
L1	No	Inst./Data	64 Bytes	4/8	32 kB/32 kB
L2	No	Unified	64 Bytes	8	256 kB
L3	Yes	Unified	64 Bytes	16	8 MB

Table 2: Caches in a Xeon W3520 processor

2 Background

This section provides background on the systems StealthMem is intended to protect, focusing on CPU caches and the channels through which cache information can be leaked. It also provides an overview of known cache-based side channel attacks.

2.1 System Model

We target modern virtualized server systems. The hardware is a shared memory multiprocessor whose processing cores share a cache (usually the last level cache). The CPUs may support simultaneous multi-threading (Hyper-Threading). The system software includes a hypervisor that partitions the hardware resources among multiple tenants, running in separate virtual machines (VMs). The tenants are not trusted and may not trust each other.

2.1.1 Cache Structure

The following short summary of caches is specific to typical x64-based CPUs, which are the target of our work. The CPU maps physical memory addresses to cache addresses (called *cache indices*) in n-byte aligned units. These units are called *cache lines*, and mapped physical addresses are called *pre-image sets* of each cache line as in Figure 1. A typical value of n is 64. We call the number of possible cache indices the *index range*. We call the index range times the line size, the *address range of the cache*.

On x64 systems, caches are typically *set associative*. Every cache index is backed by cache storage for some number $w > 1$ of cache lines. Thus, up to w different lines of memory that map to the same cache index can

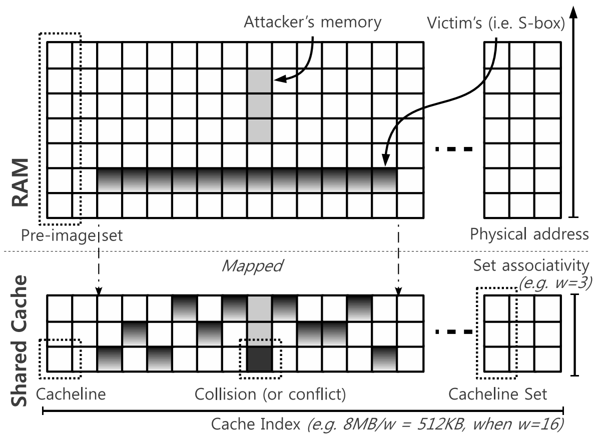


Figure 1: Cache structure and terminology

be retained in the cache simultaneously (see Figure 1). The number *w* is called the *wayness* or *set associativity*, and typical values are 8 and 16, as in Table 2. Since *w* cache lines have the same *pre-image sets* (correspondingly mapped physical memory), we refer to all *w* cache lines as a *cache line set*.

CPUs typically implement a logical hierarchy of caches, called L1, L2 and L3 depending on where they are located. L1 is physically closest to CPU, so it is the fastest (about 4 cycles), but has the smallest capacity (e.g., 32 kB). In multi-core architectures (e.g., Xeon), each core has its own L1 and backed L2 cache. The L3 cache, usually the last level cache, is the slowest (about 40 cycles) and largest cache (e.g., 8 MB). It is shared by all cores of a processor. The L3 is particularly interesting because it can be shared among virtual machines running concurrently on different cores.

2.1.2 Cache Properties

This section lists two well-known properties of caches that our algorithms rely on. The first condition is the basis for our main algorithm. We will also describe an optimization that is possible if the cache has the second property.

Inertia No cache line of a cache line set will be evicted unless there is an attempt to add another item to the cache line set. In other words, the current contents of each cache line set stay in the cache until an address is accessed that is not in the cache and that maps to the same cache line set. That is, cache lines are not spontaneously forgotten. The only exceptions are CPU instructions to flush the cache such as `invd` or `wbinvd` on x64 CPUs. However, such instructions are privileged and can be controlled by a trusted hypervisor.

***k*-LRU** Cache lines are typically evicted according to a pseudo-LRU cache replacement policy. Under an LRU replacement policy, the least recently used cache line is evicted, assuming that cache line is not likely to be utilized in the near future. Pseudo-LRU is an approximation to LRU which is cheaper to implement in hardware. We say that an associative cache has the *k-LRU property* if the replacement algorithm will never evict the *k* most recently used copies. The *k* is not officially documented by major CPU vendors and may also differ by micro architectures and their implementations. We will perform an experiment to find the proper *k* for our Xeon W3520 in Section 5.

2.1.3 Leakage Channels

This section summarizes the different ways in which information can leak through caches (see Figure 2). These leakage channels form the basis for *active time-driven attacks* and *trace-driven attacks* that we will define in the next section.

Preemptive scheduling An attacker's VM and a victim's VM may share a single CPU core (and its cache). The system uses preemptive scheduling to switch the CPU between the different VMs. Upon each context switch from the victim to the attacker, the attacker can observe the cache state as the victim had left it.

Hyper-Threading Hyper-Threading is a hardware technology that allows multiple (typically two) hardware threads to run on a single CPU core. The threads share a number of CPU resources, including the ALU and all of the core's caches. This gives rise to a number of side channels, and scheduling potentially adversarial VMs on Hyper-Threading of the same core is generally considered to be unsafe.

Multicore The attacker and the victim may be running concurrently on separate CPU cores with a shared L3 cache. In this case, the attacker can try to probe the L3 cache for accesses by the victim while the victim is running.

2.2 Cache-based Side Channel Attacks

In this section, we summarize and classify well-known cache-based side channel attacks. Following Page [31], we distinguish between time-driven and trace-driven cache attacks, based on the information that is leaked in the attacks. Furthermore, we classify time-driven attacks as *passive* or *active*, depending on the scope of the attacks.

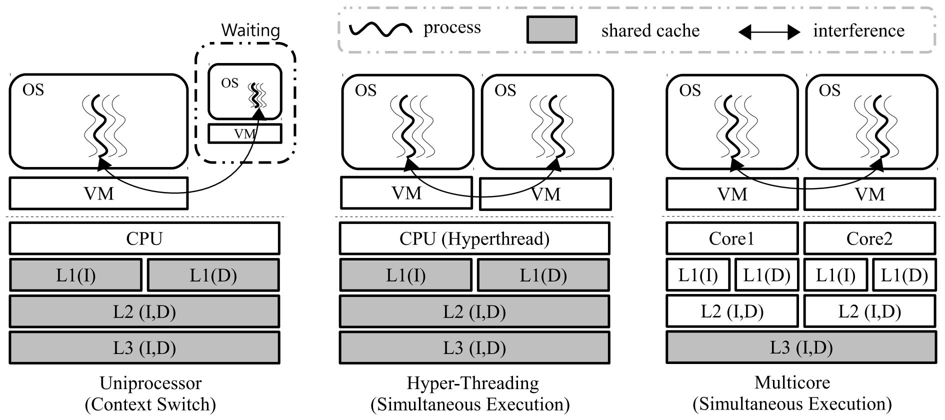

Waiting
process
shared cache
interference
OS
VM
CPU
L1(I)
L1(D)
L2 (I,D)
L3 (I,D)
Uniprocessor (Context Switch)
CPU (Hyperthread)
Hyper-Threading (Simultaneous Execution)
Core1
Core2
Multicore (Simultaneous Execution)

Figure 2: Leakage channels in three VM settings—uniprocessor, Hyper-Threading and multicore architectures. Modern commodity multicore machines suffer from all of three types of cache-based side channels. The letters (I) and (D) indicate instruction-cache and data-cache, respectively.

2.2.1 Time-driven Cache Attacks

The first class of attacks are time-driven cache attacks, also known as timing attacks. Memory access times depend on the state of the cache. This can result in measurable differences in execution times for different inputs. Such timing differences could be converted into meaningful attacks such as inferring cryptographic keys. For example, the number of cache lines accessed by a block cipher during encryption may depend on the key and on the plaintext, resulting in differences in execution times. Such differences may allow an attacker to derive the key directly or to reduce the possible key space, making it possible to extract the complete key within a feasible amount of time by brute force search.

Depending on the location of the attacker, the time-driven cache attacks fall into two categories: passive and active attacks. A passive attacker has no direct access to the victim's machine. Thus the attacker cannot manipulate or probe the victim's cache directly. Furthermore, he does not have access to precise timers on the victim's machine. An active attacker, on the other hand, can run code on the same machine as the victim. Thus, the attacker can directly manipulate the cache on the victim's machine. He can also access precise timers on that machine.

Passive time-driven cache attacks The time measurements in passive attacks are subject to two sources of noise. The initial state of the cache, which passive attackers cannot directly manipulate or observe, may influence the running time. Furthermore, since the victim's running time cannot be measured locally with a high precision timer, the measurement itself is subject to noise (e.g. due to network delays). Passive attacks, therefore, generally require more samples and try to reduce the noise by means of statistical methods.

For example, Bernstein's AES attack [8] exploits the fact that the execution time of AES encryption varies with the number of cache misses caused by S-box table lookups during encryption. The indices of the S-box lookups depend on the cryptographic key and the plaintext chosen by the attacker. After measuring the execution times for a sufficiently large number of carefully chosen plaintexts, the attacker can infer the key after performing further offline analysis.

Active time-driven cache attacks Active attackers can directly manipulate the cache state, and thus can induce collisions with the victim's cache lines. They can also measure the victim's running time directly using a high precision timer of the victim. This eliminates much of the noise faced by passive attackers, and makes active attacks more efficient. For example, Osvik et al. [30] describe an active timing attack on AES which can recover the complete 128-bit AES key from only 500,000 measurements. In contrast, Bernstein's passive timing attack required $2^{27.5}$ measurements.

2.2.2 Trace-driven Cache Attacks

The second type of cache-based side channel attacks are trace-driven attacks. These attacks try to observe which cache lines the victim has accessed by probing and manipulating the cache. Thus, like active timing attacks, trace-driven attacks require attackers to access the same machine as the victim. Given the additional information about access patterns of cache lines, trace-driven attacks have the potential of being more efficient and sophisticate than time-driven attacks.

A typical attack strategy (Prime+Probe) is for the attacker to access certain memory addresses, thus filling the cache with its own memory contents (Prime). Later, the attacker measures the time required to access the same memory addresses again (Probe). A large access time indicates a cache miss which, in turn, may indicate that the victim accessed a pre-image of the same cache line.

Trace-driven attacks were considered harmful especially with simultaneous multi-threading technologies, such as Hyper-Threading, that enable one CPU to execute multiple hardware threads at the same time without a context switch. By exploiting the fact that both threads share the same processor resources, such as caches, Percival [32] experimentally demonstrated a trace-driven cache attack against RSA. The attacker's process monitoring L1 activity of RSA encryption can easily distinguish the footprints of modular squaring and modular multiplications based on the Chinese Remainder Theorem, which is used by various RSA implementations to compute modular operations on the private key of RSA [32].

More severely, Neve [29] introduced another trace-driven attack even without requiring multi-threading technologies. Within a single-threaded processor, Neve analyzed the last round of AES encryption with multiple footprints of the AES process. To gain a footprint, Neve's attack exploits the preemptive scheduling policy of commodity operating systems. Gullasch et al. similarly used the Completely Fair Scheduler of Linux to extract full AES encryption keys. This is the first fully functional asynchronous attack in a real-world setting.

More quantitative research on trace-driven cache-based side channel attacks was conducted by Osvik, Shamir and Tromer [30, 39]. They demonstrated two interesting AES attacks by analyzing the first and second round of AES. The first attack (Prime+Probe) was able to recover a complete 128-bit AES key after only 8,000 encryptions. The second attack is asynchronous and allows an attacker to recover parts of an AES key when the victim is running concurrently on the same machine. The attack was applied to a Hyper-Threading processor. However, it is in principle also applicable to modern multicore CPUs with a shared last level cache.

3 Threat Model and Goals

With the move from private computing hardware toward cloud computing, the dangers of cache-based side channels become more acute. The sharing of hardware resources, especially CPU caches, exposes cloud tenants to both *active* time-driven and trace-driven cache attacks by co-located attackers. Neither of these attack types is typically a concern in a private computing environment which does not admit arbitrary code of unknown origin.

In contrast, *passive* time-driven attacks do not require the adversary to execute code on the victim's machine and thus apply equally to both environments. This class of attacks depends on the design, implementation, and behavior of the victim's algorithms.

The goal of this paper is to reduce the exposure of cloud systems to cache-based side channels to that of private computing environments. This requires defenses against *active* time-driven and trace-driven attacks.

We aim to design a practical system-level mechanism that provides such defenses. The design should be practical in the sense that it is compatible with existing commodity server hardware. Furthermore, its impact on system performance should be minimal, and it should not require significant changes to tenant software.

4 Design

We have designed the STEALTHMEM system to meet the aforementioned goals. The high-level idea is to provide users with a limited amount of private memory that can be accessed as if caches were not shared with other tenants. We call this abstraction *stealth memory* [13]. Tenants can use stealth memory to store data, such as the S-boxes of block ciphers, that are known to be the target of cache-based side channel attacks.

We describe our design and implementation for virtualized systems that are commonly used in public clouds. However, our design could also be applied to regular operating systems running directly on a physical machine. STEALTHMEM extends a hypervisor, such that each VM can access small amounts of memory whose cache lines are not shared.

Let p be the maximum number of CPU cores that can share a cache. This number depends on the CPU model. However, it is generally a small constant, such as $p = 4$ or $p = 6$. In particular, systems with larger numbers of processors typically consist of independent CPUs without shared caches among them.

The hypervisor selects p pre-image sets arbitrarily and assigns one page (or a few pages) from each set to one of the cores such that any two cores that share a cache are assigned pages from different pre-image sets and such that no page is assigned to more than one core. These pages are the cores' *stealth pages*, and they will be exposed to virtual machines running on the cores. At boot or initialization time, the hypervisor sets up the page tables for each core, such that each stealth page is mapped only to the core to which it was assigned. We will call the p pre-image sets from which the stealth pages were chosen the *collision sets* of the stealth pages.

Figure 3 shows an example of a CPU with four cores sharing an L3 cache. Thus, $p = 4$. STEALTHMEM would pick four pages from four different pre-image sets and set the page tables such that the i-th core has exclusive access to the i-th page.

In the rest of this section, we will refine the design and describe how STEALTHMEM disables the three leakage channels of Section 2.

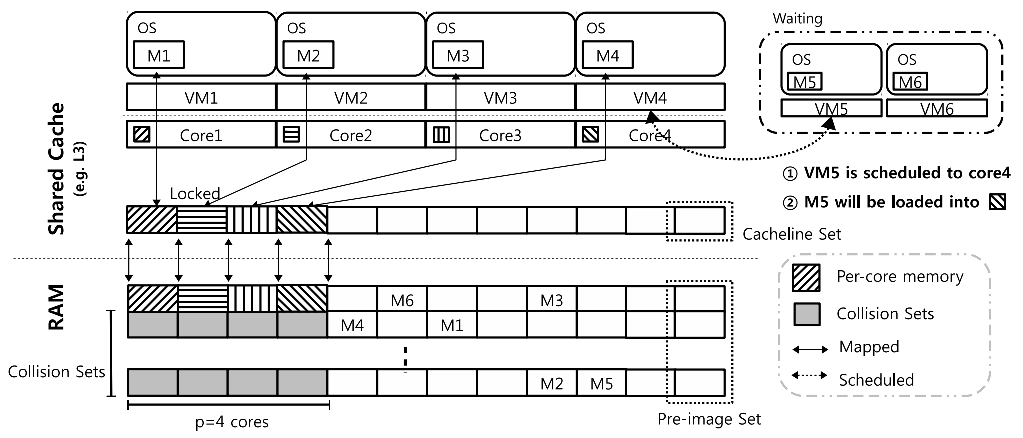


Figure 3: STEALTHMEM on a typical multicore machine: Each VM has its own stealth page. When a VM is scheduled on a core, the core will lock the VM's stealth page into the shared cache. In one version, the hypervisor will not use the collision sets in order to avoid cache collisions.

4.1 Context Switching

In general, cores are not assigned exclusively to a single VM, but are time-shared among multiple VMs. STEALTHMEM will save and restore stealth pages of VMs during context switches. In the notation of Figure 3, when VM5 is scheduled to a core currently executing VM4, the STEALTHMEM hypervisor will save the stealth pages of the core into VM4's context, and restore them from VM5's context. STEALTHMEM will thus ensure that all of VM4's stealth pages are removed from the cache and all of VM5's stealth pages are loaded into the cache. STEALTHMEM performs this step at the very end of the context switch—right before control is transferred from VM4 to VM5. This way, all of VM5's stealth pages will be in the L1 cache (in addition to being in L2 and L3) when VM5 starts executing.

Guest operating systems can use the same technique to multiplex their stealth memory to an arbitrary number of applications.

4.2 Hyper-Threading

In order to avoid asynchronous cache side channels between hyperthreads on the same CPU core, STEALTHMEM gang schedules them. In other words, the hyperthreads of a core are never simultaneously assigned to different VMs. Some widely used hypervisors such as Hyper-V already implement this policy. Given the tight coupling of hyperthreads through shared CPU components, it is hard to envision how the hyperthreads of a core could be simultaneously assigned to multiple VMs without giving rise to a multitude of side channels. Another option is to disable Hyper-Threading.

4.3 Multicore

STEALTHMEM has to prevent an attacker running on one core from using the shared cache to gain information about the stealth memory accesses of a victim running concurrently on another core. For this purpose, STEALTHMEM has to remove or tightly control access to any page that maps to the same cache lines as the stealth pages; i.e., to the p pre-image sets from which the stealth pages were originally chosen. We consider two options: a) STEALTHMEM makes these pages inaccessible and b) STEALTHMEM makes the pages available to VMs, but mediates access to them carefully.

Under the first option, STEALTHMEM ensures at the hypervisor level that, beyond the stealth pages, no pages from the p pre-image sets from which the stealth pages were taken are mapped in the hardware page tables. Thus, these pages are not used and are physically inaccessible to any VM. There is no accessible page in the system that maps to the same cache lines as the stealth pages. Code running on one core cannot probe or manipulate the cache lines of another core's stealth page because it cannot access any page that maps to the same cache lines.

The total amount of memory that is sacrificed in this way depends on the shared cache configuration of the processor. It is about 3% for all CPU models we have examined. For example, the Xeon W3520 of Table 2 has an 8 MB 16-way set associative L3 cache that is shared among 4 cores ($p = 4$). Dividing 8 MB by the wayness (16) and the page size (4096 bytes), yields 128 page-granular pre-image sets. Removing $p = 4$ of them corresponds to a memory overhead of $4/128 = 3.125\%$. The available shared cache is reduced by the same amount.

One could consider the option of reducing the overhead by letting trusted system software (e.g. the hypervisor, or root partition) use the reserved pages, rather than not assigning them to guest VMs. However, this would make it hard to argue about the security of the resulting system. For example, if the pages were used to store system code, one would have to ensure that attackers could not access the cache lines of stealth pages indirectly by causing the execution of certain system functions.

4.4 Page Table Alerts

The second option is to use the memory from the p pre-image sets, but to carefully mediate access to them. This option eliminates the memory and cache overhead at the expense of maintenance cost.

StealthMem maintains the invariant that the stealth pages never leave the shared cache. The shared cache is w-way set associative. Intuitively, StealthMem tries to reserve one of the w slots for the stealth cache line, while the remaining $w-1$ slots can be used by other pages. StealthMem interposes itself on accesses that might cause stealth cache lines to be evicted by setting up the hardware page mappings for most of the colliding pages, such that attempts to access them result in page faults and, thus, invocation of the hypervisor. We call this mechanism a page table alert (PTA).

Rather than simply not using the pre-image sets, the hypervisor maps all their pages to VMs like regular pages. However, the hypervisor sets up PTAs in the hardware page mappings for most of these pages.

More precisely, the hypervisor ensures that there will never be more than $w-1$ pages (other than one stealth page) from any of the p pre-image sets without a PTA. The $w-1$ pages without PTAs are effectively a cache of pages that can be accessed directly without incurring the overhead of a PTA.

At initialization, the hypervisor places a PTA on every page of each of the p pre-image sets. Upon a page fault, the handler in the hypervisor will determine if the page fault was caused by a PTA. If so, it will determine the pre-image set of the page that triggered the page fault and perform the following steps: (a) If the pre-image set already contains $w-1$ pages without a PTA then one of these pages is chosen (according to some replacement strategy), and a PTA is placed on it. (b) The hypervisor ensures that all cache lines of the stealth page and of the up to $w-1$ colliding pages without PTAs are in the cache. This can be done by accessing these cache lines—possibly repeatedly. On most modern processors, the hypervisor can verify that the lines are indeed in the cache by querying the CPU performance counters for the number of L3 cache misses that occurred while accessing the w pages. If this number is zero then all required lines are in the cache. (c) The hypervisor removes the PTA from the page that caused the page fault. (d) The hypervisor resumes execution of the virtual processor that caused the page fault. The hypervisor executes steps (b) and (c) atomically—preemption is disabled.

The critical property of these steps is that all accesses to the w pages without PTAs will always hit the cache and, by the inertia property, not cause any cache evictions. Any accesses to other pages from the same pre-image set are guarded by PTAs and will be mediated by StealthMem.

In order to improve scalability, we maintain a separate set of PTAs for each group of p processors that share the cache. Steps (a) to (d) are performed only locally for the set of PTAs of the processor group that contains the processor on which the page fault occurred. Thus, only the local group of p processors needs to be involved in the TLB shootdown, and different processor groups can have different sets of pages on which the PTAs are disabled. This comes at the expense of additional memory for page tables.

***k*-LRU** If the CPU's cache replacement algorithm has the k-LRU property (see Section 2) for some $k>1$, the following simplification is possible in step (b). Rather than loading the cache lines from all pages without PTAs from the pre-image set, StealthMem only needs to access once each cache line of the stealth page. This reduces the overhead per PTA.

Furthermore, the maximum number of pages without PTAs must now be set to $k-1$, which may be smaller than $w-1$. This may lead to more PTAs in this variant of the algorithm.

The critical property of this variant of the algorithm is that, at any time, the only pages in the stealth page's pre-image set that could have been accessed more recently than the stealth page are the $k-1$ pages without PTAs. Thus, by the k-LRU property, the stealth page will never be evicted from the cache. Figure 4 illustrates this for $k=4$.

4.5 Optimizations

Our design to expose stealth pages to arbitrary numbers of VMs adds work to context switches. Early experiments showed that this overhead can be significant. We use the following optimizations to minimize this cost.

We associate physical stealth pages with cores, rather than VMs, in order to minimize the need for shared data structures and the resulting lock contention. StealthMem virtualizes these physical stealth pages and exposes a (virtual) stealth page associated with each virtual processor of a guest. This requires copying the contents of a virtual processor's stealth page and acquiring inter-processor locks whenever the hypervisor's scheduler decides to move a virtual processor to a different core. This event, however, is relatively rare and costly in itself. Thus, the work we add is only a small fraction of the total cost.

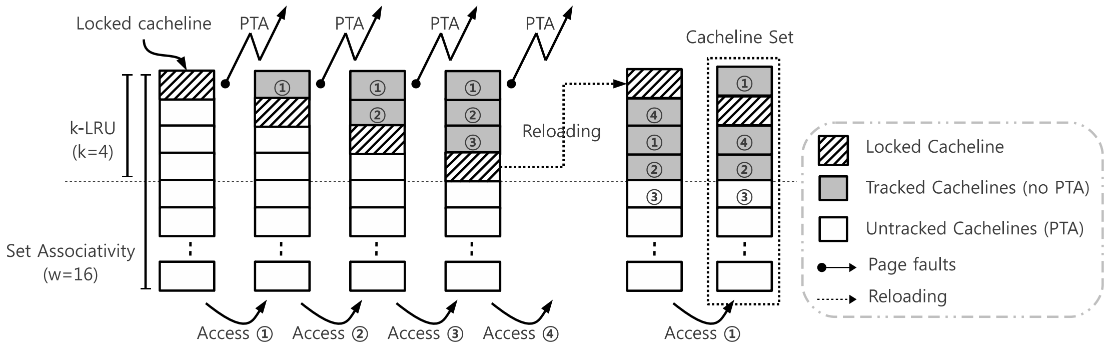


Figure 4: Page table alerts on accessing pages 1, 2, 3, 4 and 1, which are the pre-images of the same cache line set. When getting a page fault on accessing page 4, STEALTHMEM$_{\text{PTA}}$ reloads the stealth page to lock its cache lines. The k-LRU policy ($k = 4$) guarantees that the stealth page will not be evicted from the cache. Extra page faults come from accessing PTA-guarded pages. Accessing the tracked cache lines (pages without PTAs) will not generate extra page faults and, thus, no extra performance penalty.

With this optimization, each guest still has its own private stealth pages (one per virtual processor). A potential difficulty of this approach is that guest code sees different stealth pages, depending on which virtual processor it runs on. However, this problem is immaterial for the standard application of STEALTHMEM, in which the stealth pages store S-box tables that never change.

Furthermore, we use several optimizations to minimize the cost of copying stealth pages and flushing their cache lines during context switches. Rather than backing the contents of a core's stealth page to a regular VM context page, we give each VM a separate set of stealth pages. Each VM has its own stealth page from pre-image set i for core i. Thus, if a VM is preempted and later resumes execution on the same set of cores, it is only necessary to refresh the cache lines of its stealth pages. The contents of a stealth page only have to be saved and restored if a virtual processor moves to a different core.

A frequent special case are transitions between a VM and the root partition. When a VM requires a service, such as access to the disk or the network, the root partition needs to be invoked. After the requested service is complete, control is returned to the VM—typically on the same cores on which it was originally running. Thus, it is not necessary to copy the stealth page contents on either transition. Furthermore, since we do not assign stealth pages to the root partition, it is not even necessary to flush caches.

4.6 Extensions

As long as the machine has sufficient memory, we do not use the pages from the collision sets. This will help STEALTHMEM to avoid the performance overhead of maintaining PTAs. If, at some point, the machine is short of memory, STEALTHMEM can start assigning PTA-guarded pages to VMs, making all memory accessible.

STEALTHMEM can, in principle, provide more than one page of stealth memory per core. In order to ensure that stealth pages are not evicted from the cache, the number of stealth pages per core can be at most $k-1$ for variants that rely on the k-LRU property and at most $w-1$ for other variants, where w is the wayness of the cache.

4.7 API

VM level STEALTHMEM exposes stealth pages as architectural features of virtual processors. The guest operating system can find out the physical address of a virtual processor's stealth page by making a hypercall, which is a common interface to communicate with the hypervisor.

Application level Application code has to be modified in order to place critical data on stealth pages. STEALTHMEM provides programmers with two simple APIs for requesting and releasing stealth memory as shown in Table 3: *sm_alloc*() and *sm_free*(). Programmers can protect important data structures, such as the S-boxes of encryption algorithms, by requesting stealth memory and then copying the S-boxes to the allocated space. In Section 6, we will evaluate the API design by modifying popular cryptographic algorithms, such as DES, AES and Blowfish, in order to protect their S-boxes with STEALTHMEM.

5 Implementation

We have implemented the STEALTHMEM design on Windows Server 2008 R2 using Hyper-V for virtualization. The STEALTHMEM implementation consists of 5,000 lines of C code that we added to the Hyper-V hypervisor. We also added 500 lines of C code to the Windows boot loader modules (bootmgr and winloader).

API	Description
*void *sm_alloc(size_t size)*	Allocate dynamic memory of size bytes and return a corresponding pointer
*void sm_free(void *ptr)*	Free allocated memory pointed to by the given pointer, *ptr*

Table 3: APIs to allocate and free stealth memory

STEALTHMEM exposes stealth pages to applications through a driver that runs in the VMs and that produces the user mode mappings necessary for *sm_alloc*() and *sm_free*(). We did not have to modify the guest operating system to use STEALTHMEM.

We implemented two versions of STEALTHMEM. In the first implementation, Hyper-V makes the unused pages from the p pre-image sets inaccessible. We will refer to this implementation as STEALTHMEM. The second implementation maps those pages to VMs, but guards them with PTAs. We will explicitly call this version STEALTHMEM$_{\text{PTA}}$.

Hyper-V configures the hardware virtualization extensions to trap into the hypervisor when VM code executes `invd` instructions. We extended the handler to reload the stealth cache lines immediately after executing `invd`. We proceeded similarly with `wbinvd`.

5.1 Root Partition Isolation

Hyper-V relies on Windows to boot the machine. First, Windows boots on the physical machine. Hyper-V is launched only after that. The Windows instance that booted the machine becomes the root partition (equivalent to dom0 in Xen). In general, by the time Hyper-V is launched, the root partition will be using physical pages from all pre-image sets. It would be hard or impossible to free up complete pre-image sets by evicting the root partition from selected physical pages. The reasons include the use of large pages which span all pre-image sets or the use of pages by hardware devices that operate on physical addresses.

We obtain pre-image sets that are not used by the system by marking all pages in these sets as *bad pages* in the boot configuration data using `bcdedit`. This causes the system to ignore these pages and cuts physical memory into many small chunks. We had to adapt the Windows boot loader to enable Windows to boot under this unusual memory configuration.

As a result of this change there are no contiguous large (2 MB or 4 MB) pages on the machine. Both the Windows kernel and Hyper-V attempt to use large pages to improve performance. Large page mappings reduce the translation depth from virtual to physical addresses. Furthermore, they reduce pressure on the TLB. We will evaluate the impact of not using large pages on the performance of STEALTHMEM in Section 6).

5.2 *k*-LRU

Major CPU vendors implement pseudo-LRU replacement policies as an approximation of the LRU policy [14]. However, this is neither officially documented nor explicitly stated in CPU developer manuals [6, 16]. We conducted the following experiment to find a k value for which our target Xeon W3520 CPU has the k-LRU property.

We selected a set of pages that mapped to the same cache lines. Then, we loaded one page into the L3 cache by reading the contents of the page. After that, we loaded k' other pages of the same pre-image set. Then, we turned on the performance counter and checked L3 cache misses after reading the first page again. We ran this experiment in a device driver (ring0) on one core, while the other cores were spinning on a shared lock. Interrupts were disabled. We varied k' from 1 to 16 (set associativity). We started seeing L3 misses at $k' = 15$ and concluded that our CPU has the 14-LRU property.

6 Evaluation

We ask three questions to evaluate STEALTHMEM. First, how effective is STEALTHMEM against cache-based side channel attacks? Second, what is the performance overhead of STEALTHMEM and its characteristics? And finally, how easy is it to adopt STEALTHMEM in existing applications?

6.1 Security

6.1.1 Basic Algorithm

We consider the basic algorithm (without the optimizations of Section 4.5) first. STEALTHMEM guarantees that all cache lines of stealth pages are always in the shared (L3) cache. In the version that makes colliding pages inaccessible, this is the case simply because on each group of cores that share a cache, the only accessible pages from the collision sets of the stealth pages are the stealth pages themselves. We load all stealth pages into the shared cache at initialization. Since Section 4.6 limits the number of stealth pages per collision set to $w-1$, this will result in all stealth pages being in the cache simultaneously. It is impossible to generate collisions. Thus, by the inertia property, these cache lines will never be evicted.

In the PTA version, it is theoretically possible for stealth cache lines to be evicted very briefly from the cache during PTA handling while the $w-1$ colliding pages without PTAs are loaded into the cache. The stealth cache line would be reloaded immediately as part of the same operation, and the time outside the shared cache could be limited to one instruction by accessing the stealth cache line immediately after accessing a colliding line.

Leakage channels This property together with other properties of STEALTHMEM prevents trace-driven and active time-driven attacks on stealth pages. We consider each of the three leakage channels in turn:

Multicore: Attackers running concurrently on other cores cannot directly manipulate (prime) or probe stealth cache lines of the victim's core. This holds for the shared cache because, as observed above, all stealth lines always remain in the shared (L3) cache irrespective of the actions of victims or attackers. It also holds for the other caches (L1 and L2) because they are not shared.

Time sharing: Attackers who time-share a core with a victim cannot directly manipulate or probe stealth cache lines either because we load all stealth cache lines into the cache (including L1 and L2) at the very end of a context switch. Thus, no matter what the adversary or the victim did before the context switch, all stealth lines will be in all caches after a context switch. Thus, direct priming and probing the cache should yield no information.

Hyper-Threading: STEALTHMEM gang schedules hyperthreads to prevent side channels across them.

Limitations While STEALTHMEM locks stealth lines into the last level shared (L3) cache, it has no such control over the upper level caches (L1 and L2) other than reloading stealth pages while context switching. Accordingly, STEALTHMEM cannot hide the timing differences coming out of L1 and L2 cache. Passive timing attacks may arise by exploiting the timing differences between L1 and L3 from a different VM. As stated earlier, passive timing attacks are not our focus since they are not a new threat that results from hardware sharing in the cloud.

6.1.2 Extensions and Optimizations

Per-VM stealth pages Section 4.5 describes an optimization that maintains a separate set of per-core stealth pages for each VM. With this optimization, stealth cache lines are not guaranteed to stay in the shared cache permanently. However, by loading the stealth page contents into the cache at the end of context switches, STEALTHMEM guarantees that the contents of a VM's per-core stealth pages are reloaded in the shared cache, *whenever the core executes the VM*. Thus, the situation for attackers running concurrently on different cores is the same as for the basic algorithm. Our observations regarding context switches and Hyper-Threading also carry over directly.

***k*-LRU** In the PTA variant that relies on the k-LRU property, the stealth page is kept in the cache because at most $k-1$ colliding pages can be accessed without PTAs. Since STEALTHMEM accesses the stealth page at the end of every page fault that results in a PTA update, the stealth cache lines are always at least the k-least recently used lines in their associative set. Thus, on a CPU with the k-LRU property, they will not be evicted.

6.1.3 Denial of Service

VMs do not have to (and cannot) request or release stealth pages. Instead, STEALTHMEM provides every VM with its own set of stealth pages as part of the virtual machine interface. This set is fixed from the point of view of the VM. Accesses by a VM to its stealth pages do not affect other VMs. Thus, there should be no denial of service attacks involving stealth pages at the VM interface level.

Guest operating systems running inside VMs may have to provide stealth pages to multiple processes. The details of this lie outside the scope of this paper. As noted above, the techniques used in STEALTHMEM can also be applied to operating systems. Operating systems that choose to follow the STEALTHMEM approach virtualize their VM-level stealth pages and provide a fixed independent set of stealth pages to each process. Again, this type of stealth memory should not give rise to denial of service attacks. The APIs of Table 3 would be merely convenient syntax for a process to obtain a pointer to its stealth pages.

6.2 Performance

We have measured the performance of our STEALTHMEM implementation to assess the efficiency and practicality of STEALTHMEM. The experiments ran on an HP Z400 workstation with a 2.67 GHz 4 core Intel Xeon W3520 CPU with 16 GB of DDR3 RAM. The cores were running at 2.8 GHz. Each CPU core has a 32 kB 8-way L1 D-cache, a 32 kB 4-way L1 I-cache and a 256 kB 8-way L2 cache. In addition, the four cores share an 8 MB 16-way L3 cache. The machine ran a 64-bit version of Windows Server 2008 R2 HPC Edition (no service pack). We configured the power settings to run the CPU always at full speed in order to reduce measurement noise. The virtual machines used in the experiments ran the 64-bit version of Windows 7 Enterprise Edition and had 2 GB of RAM. This was the recommended minimum amount of memory for running the SPEC 2006 CPU benchmark [37].

6.2.1 Performance Overhead

Our first goal was to estimate the overhead of STEALTHMEM and STEALTHMEM$_{\text{PTA}}$. We have measured execution times for three configurations: Baseline—an un-

Benchmark	**Baseline**		**Stealth**			**Stealth PTA**			**BaselineNLP**		
	time	st.dev.	time	st.dev.	overhead	time	st.dev.	overhead	time	st.dev.	overhead
perlbench	508	0.1%	537	0.3%	5.7%	538	0.5%	5.9%	532	0.5%	4.7%
bzip2	610	2.0%	618	0.2%	1.3%	624	1.8%	2.3%	617	2.0%	1.1%
gcc	430	0.1%	466	0.3%	8.4%	476	0.2%	10.7%	462	0.3%	7.4%
milc	257	0.1%	289	0.7%	12.5%	298	0.5%	16.0%	284	1.6%	10.5%
namd	498	0.0%	500	0.1%	0.4%	500	0.1%	0.4%	499	0.1%	0.2%
dealII	478	0.1%	492	0.3%	2.9%	495	0.2%	3.6%	490	0.1%	2.5%
soplex	361	1.9%	401	0.4%	11.1%	412	0.3%	14.1%	394	0.2%	9.1%
povray	228	0.1%	229	0.6%	0.4%	229	0.1%	0.4%	228	0.2%	0.0%
calculix	360	0.2%	366	0.3%	1.7%	366	0.3%	1.7%	363	0.8%	0.8%
astar	454	0.1%	501	0.3%	10.4%	508	1.3%	11.9%	495	0.2%	9.0%
wrf	307	1.9%	331	0.8%	7.8%	336	1.2%	9.4%	329	0.6%	7.2%
sphinx3	602	0.1%	654	0.4%	8.6%	662	0.7%	10.0%	639	0.2%	6.1%
xalancbmk	307	0.2%	324	0.2%	5.5%	329	0.3%	7.2%	321	0.0%	4.6%
average					5.9%			7.2%			4.9%

Table 4: Running time in seconds (time), error bound (st.dev.) and overhead on 13 SPEC2006 CPU benchmarks for Baseline, StealthMem, StealthMem$_{\text{PTA}}$ and BaselineNLP.

modified version of Windows with an unmodified version of Hyper-V—and our respective implementations of StealthMem and StealthMem$_{\text{PTA}}$.

In the first experiment, we ran each configuration with two VMs. One VM ran the SPEC 2006 CPU benchmark [37]. Another VM was idle. Table 4 displays the execution times for 13 applications from the SPEC benchmark suite. We repeated each run ten times, obtaining ten samples for each time measurement. The running times in the table are the sample medians. The table also displays the sample standard deviation as a percentage of the sample average as an indication of the noise in the sample. The sample standard deviation is typically less than one percent of the sample average.

The overhead of StealthMem varies between close to zero for about one third of the SPEC applications and 12.5% for *milc*. The average overhead is 5.9%. As expected, the overhead of StealthMem$_{\text{PTA}}$ (7.2%) is larger than that of StealthMem because of the extra cost of handling PTA page faults. Server operators can choose either variant, depending on the memory usage of their servers.

We also attempted to find the source of the overhead of StealthMem. Possible sources are the cost of virtualizing stealth pages, the 3% reduction in the size of the available cache and the cost of not being able to use large pages. We repeated the experiment with a configuration that is identical to the Baseline configuration, except that it does not use large pages. It is labeled BaselineNLP (for 'no large pages') in Table 4. The overheads for BaselineNLP across the different SPEC applications correlate with the overheads of StealthMem. The overhead due to not using large pages (4.9% on average) accounts for more than 80% of the overhead of StealthMem.

We constructed BaselineNLP using the same binaries as Baseline. However, at hypervisor startup, we disabled one Hyper-V function by using the debugger to overwrite its first instruction with a *ret*. This function is responsible for replacing regular mappings by large mappings in the extended page tables. Without it, Hyper-V will not use large page mappings irrespective of the actions of the root partition or other guests.

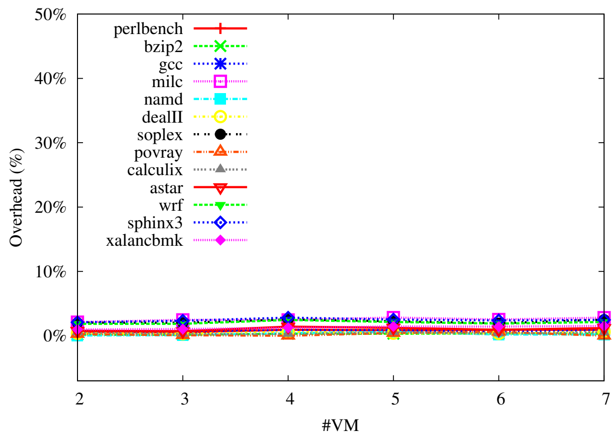
6.2.2 Comparison with Page Coloring

Page coloring [33] isolates VMs from cache-related dependencies by partitioning physical memory pages among VMs such that no VM shares cache lines with any other VM. We modified one of the Hyper-V support drivers in the root partition (vid.sys) to assign physical memory to VMs accordingly.

In this simple implementation of Page Coloring, the VMs still share cache lines with the root partition. The same holds for the system in [33]. In contrast, our StealthMem implementation isolates stealth pages also from the root partition. While this difference makes the Page Coloring configuration less secure, it should work to its advantage in the performance comparison.

The next experiment compares the overheads of StealthMem and Page Coloring as the number of VMs increases. We ran BaselineNLP, StealthMem and Page Coloring with between 2 and 7 VMs, running the SPEC workload in one VM and leaving the remaining VMs idle. The root partition is not included in the VM count. Again, each time measurement is the median of ten SPEC runs. The sample standard deviation was typically less than1% and in no case more than 2.5% of the sample mean.

Figure 5 displays the overheads over BaselineNLP of StealthMem (left) and Page Coloring (right) as a function of the number of VMs. We chose to display the overhead over BaselineNLP, rather than Baseline, in order to eliminate the constant cost of not using large pages,



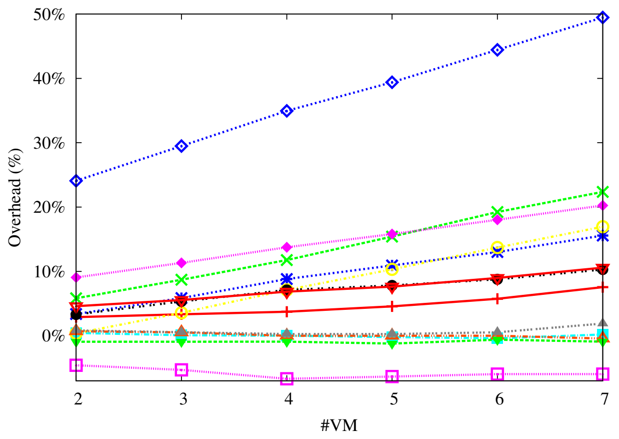


Figure 5: Overhead of STEALTHMEM (left) and Page Coloring (right) over BaselineNLP. The *x*-axis is the number of VMs.

which affects STEALTHMEM and Page Coloring similarly. Using Baseline adds an application dependent constant to each curve.

Overall, the overhead of STEALTHMEM is significantly smaller than the overhead of Page Coloring. The latter grows with the number of VMs, as each VM gets a smaller fraction of the cache. In contrast, the overhead of STEALTHMEM remains largely constant as the number of VMs increases.

Figure 5 also shows significant differences between the individual benchmarks. For eight benchmarks, Page Coloring shows a large and rising overhead. The most extreme case of this is *sphinx3* with a maximum overhead of almost 50%. For four benchmarks, the overhead of Page Coloring is close to zero. Finally, the *milc* benchmark stands out, as Page Coloring runs it consistently faster than BaselineNLP and STEALTHMEM.

These observations are roughly consistent with the cache sensitivity analysis of Jaleel [19]. The applications with low overhead (*namd*, *povray* and *calculix*) appear to have very small working sets that fit into the L3 cache of all configurations we used in the experiment (including Page Coloring with 7 VMs). For the eight benchmarks with higher overhead, the number of cache misses appears to be sensitive to lower cache sizes in the range covered by our Page Coloring experiment (8/7 MB to 8 MB). For the *milc* application, the data reported by Jaleel indicate a working set size of more than 64 MB. This suggests that *milc* may be thrashing the L3 cache as well as the TLB even when given the entire cache of the machine under BaselineNLP. The performance improvement under Page Coloring may be the result of the CPU being able to resolve certain events (such as page table walks) faster when a large part of the cache is not being thrashed by *milc*.

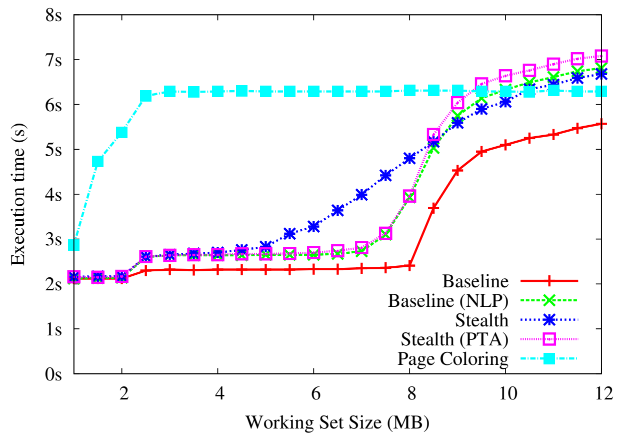


Figure 6: Running times of a micro-benchmark as a function of its working set size.

6.2.3 Overhead With Various Working Set Sizes

The following experiment shows overhead as a function of working set size. Given the working set of an application, developers can estimate the expected performance overhead when they modify an application to use STEALTHMEM.

In the experiment, we used a synthetic application that makes a large number of accesses to an array whose size we varied (the working set size). The working set size is the input to the application. It allocates an array of that size and reads memory from the array in a tight loop. The memory accesses start at offset zero and move up the array in a quasi-linear pattern of increasing the offset for the next read operation by 192 bytes (three cache line sizes) and reducing the following offset by 64 bytes (one cache line size). This is followed by another 192 byte increase and another 64 byte reduction etc. When the end of the array is reached, the process is repeated, starting

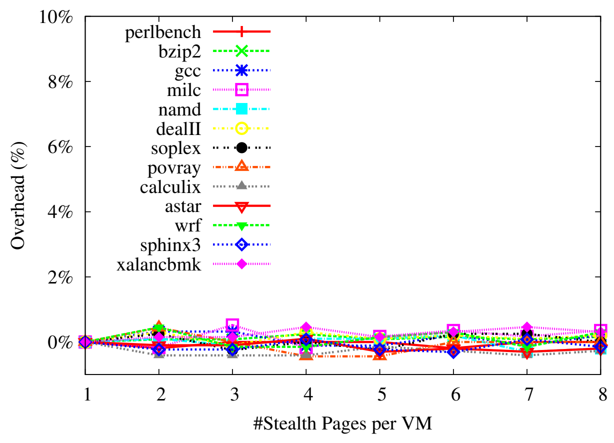


Figure 7: Overhead of STEALTHMEM as a function of the number of stealth pages

again at offset zero.

We ran the application for several configurations. In each case, we ran seven VMs. One VM was running our application. The remaining six VMs were idle. We varied the working set sizes from 100 kB to 12.5 MB and measured for each run the time needed by the application to make three billion memory accesses. The results are displayed in Figure 6. The time measurements in the figure are the medians over five runs. The sample standard deviations were less than 0.5% of the sample means for most working set sizes. However, where the slope of a curve was very steep, the sample standard deviations could be up to 5% of the sample means.

Most configurations show a sharp rise in the running times as the working set size increases past the size of the L3 cache (8 MB). For Page Coloring, this jump occurs for much smaller working sets since the VM can access only one seventh of the CPU's cache. Most configurations also display a second, smaller increase around 2 MB. This may be the result of TLB misses. The processor's L2 TLB has 512 entries which can address up to 2 MB based on regular 4 kB page mappings.

For very large workload sizes, BaselineNLP and STEALTHMEM become slower than Page Coloring. This appears to be the same phenomenon that caused Page Coloring to outperform BaselineNLP and STEALTHMEM on the *milc* benchmark.

6.2.4 Overhead With Various Stealth Pages

This experiment attempts to estimate how the overhead of STEALTHMEM depends on the number of stealth pages that the hypervisor provides to each VM. We ran STEALTHMEM with one VM running the SPEC benchmarks and varied the number of stealth pages per VM. As before, the times we report are the medians over ten runs. The sample standard deviations were less than 0.4% of the sample means in all cases.

Figure 7 displays the overhead with respect to STEALTHMEM with one stealth page per VM. There is no noticeable increase in the running time as the number of stealth pages increases. This is the result of the optimizations described earlier that eliminate the need to copy the contents of stealth pages or to load them into the cache frequently.

6.3 Block Ciphers

The goal of this experiment is to evaluate performance for real-world applications that heavily use stealth pages. We choose three popular block ciphers: AES [2], DES [1] and Blowfish [35]. Efficient implementations of each of these ciphers perform a number of lookups in a table during encryption and decryption. We picked Bruce Schneier's implementation of Blowfish [36], and standard commercial implementations of AES and DES and adapted them to use stealth pages (as described in Section 6.4).

We measured the encryption speeds of each of the ciphers for (a) the baseline configuration (unmodified Windows 7, Hyper-V and cipher implementation), (b) our STEALTHMEM configuration using the modified versions of the cipher implementations just described and (c) an uncached configuration, which places the S-box tables on a page that is not cached. Configuration (c) runs the modified version of the block cipher implementations on an unmodified version of Windows and an essentially unmodified version of the hypervisor. We added a driver in the Windows 7 guest that creates an uncached user mode mapping to a page. We also had to add one hypercall to Hyper-V to ensure that this page was indeed mapped as uncached in the physical page tables. We included this configuration in our experiments since using an uncached page is the simplest way to eliminate cache side channels.

We measured the time required to encrypt 5 million bytes for each configuration. In order to reduce measurement noise, we raised the scheduling priority of the encryption process to the HIGH_PRIORITY_CLASS of the Windows scheduler. We ran the experiment in a small buffer configuration (50,000 byte buffer encrypted 1,000 times) and a large buffer configuration (5 million byte buffer encrypted once) to show performance overheads with different workloads.

The numbers in Table 5 are averaged over 1,000 runs. The sample standard deviation lies between 1 and 4 percent of the sample averages. The overhead of using a stealth page with respect to baseline performance lies between 2% and 5%, while the overhead of the uncached version lies between 97.9% and 99.9%.

	A small buffer (50,000 bytes)					A large buffer (5,000,000 bytes)				
Cipher	**Baseline**	**Stealth**		**Uncached**		**Baseline**	**Stealth**		**Uncached**	
DES	60	58	-3%	0.83	-99%	59	57	-3%	0.83	-99%
AES	150	143	-5%	1.33	-99%	142	135	-5%	1.32	-99%
Blowfish	77	75	-2%	1.65	-98%	75	74	-2%	1.64	-98%

Table 5: Block cipher encryption speeds in MB/s for small and large buffers. We mapped the S-box of each encryption algorithm to cached, stealth and uncached pages.

	Source code
Original	static unsigned long S[4][256];

Modified	typedef unsigned long UlongArray[256];
	static UlongArray *S;
	// in the initialization function
	S = sm_alloc(4*256);

Table 6: Modified Blowfish to use STEALTHMEM

Encryption	**Size of S-box**	**LoC Changes**
DES	256 B * 8 = 2 kB	5 lines
AES	1024 B * 4 = 4 kB	34 lines
Blowfish	1024 B * 4 = 4 kB	3 lines

Table 7: Size of S-box in various encryption algorithms, and corresponding changes to use STEALTHMEM

6.4 Ease-of-use

We had to make only minor changes to the block cipher implementations to adapt them to STEALTHMEM. These changes amounted to replacing the global array variables that hold the encryption tables by pointers to the stealth page. In the case of Blowfish, this change required only 3 lines. We replaced the global array declaration by a pointer and assigned the base of the stealth page to it in the initialization function (see Table 6).

Adapting DES required us to change a total of 5 lines. In addition to a change of the form just described, we had to copy the table contents (constants in the source code) to the stealth page. This was not necessary for Blowfish which read these data from a file. Adapting AES required a total of 34 lines. This large number is the result of the fact that our AES implementation declares its table as 8 different variables, which forced us to repeat 8 times the simple adaptation we did for DES. Table 7 summarizes the S-box layouts and the required code changes for the three ciphers.

7 Related Work

Kocher [22] presented the initial idea of exploiting timing differences to break popular cryptosystems. Even though Kocher speculated about the possibility of exploiting cache side channels, the first theoretical model of cache attacks was described by Page [31] in 2002. Around that time, researchers started investigating cache-based side channels against actual cryptosystems and broke popular cryptosystems such as AES [4, 8, 9, 30], and DES [40]. With the emergence of simultaneous multithreading, researchers discovered a new type of cache attacks, classified as trace-driven attacks in our paper, against AES [3, 30] and RSA [32] by exploiting the new architectural feature of an L1 cache that is shared by two hyperthreads. Recently, Osvik et al. [30, 39] executed more quantitative research on cache attacks and classified possible attack methods. The new cloud computing environments have also gained the attention of researchers who have explored the possibility of cache-based side channel attacks in the cloud [7, 34, 44], or inversely their use in verifying co-residency of VMs [45].

Mitigation methods against cache attacks have been studied in three directions: suggesting new cache hardware with security in mind, designing software-only defense mechanism, and developing application specific mitigation methods.

Hardware-based mitigation methods focus on reducing or obfuscating cache accesses [23, 24, 41–43] by designing new caches, or partitioning caches with dynamic or other efficient methods [12, 21, 27, 42, 43]. Wang and Lee [42, 43] proposed PLcache to hide cache access patterns by locking cache lines, and RPcache to obfuscate patterns by randomizing cache mappings. These hardware-based approaches, however, will not provide practical defenses until CPU makers integrate them into mainstream CPUs and cloud providers purchase them. Our defense mechanism not only provides similar security guarantee as these methods, but also allows cloud providers to utilize existing commodity hardware.

Software-only defenses [7, 11, 13, 15, 33] also have been actively proposed. Against time-drive attacks, Coppens et al. [11] demonstrated a mitigation method by modifying a compiler to remove control-flow dependencies on confidential data, such as secret keys. This compiler technique, however, leaves applications still vulnerable to trace-driven cache attacks in the cloud. Against trace-driven attacks, static partitioning techniques, such as page coloring [33], provide a general mitigation solution by partitioning pre-image sets among VMs. Since static partitioning divides the cache by the number of VMs, its performance overhead becomes significantly larger when cloud providers run more VMs, as we demonstrated in

Section 6. Our solution, however, assigns unique cache line sets to virtual processors and flexibly loads stealth pages of each VM if necessary, and thus demonstrates better performance.

Erlingsson and Abadi [13] proposed the abstraction of "stealth memory" and sketched techniques for implementing it. We have realized the abstraction in a virtualized multiprocessor environment by designing and implementing a complete defense system against cache side channel attacks and evaluating it across system layers (from the hypervisor to cryptographic applications) in a concrete security model.

Since existing hardware-based and software-only defenses are not practical because they require new CPU hardware or because of their performance overhead, researchers have been exploring mitigation methods for particular algorithms or applications. The design and implementation of AES has been actively revisited by [8–10, 14, 30, 39], focusing on eliminating or controlling access patterns on S-Boxes, or not placing S-Boxes into memory [28], but into registers of x64 CPUs. Recently, Intel [17] introduced a special instruction for AES encryption and decryption. These approaches may secure AES from cache side channels, but it is not realistic to introduce new CPU instructions for every software algorithm that might be subject to leaking information via cache side channels. In contrast, STEALTHMEM provides a general system-level protection solution that every application can take advantage of if it wants to protect its confidential data in the cloud.

8 Conclusion

We design and implement STEALTHMEM, a system-level protection mechanism against cache-based side channel attacks, specifically against active time-driven and trace-driven cache attacks, which cloud platforms suffer from. STEALTHMEM helps cloud service providers offer better security against cache attacks, without requiring any hardware modifications.

With only a few lines of code changes, we can modify popular encryption schemes such as AES, DES and Blowfish to use STEALTHMEM. Running the SPEC 2006 CPU benchmark shows an overhead of 5.9%, and our micro-benchmark shows that the secured AES, DES, and Blowfish have between 2% and 5% performance overhead, while making extensive use of STEALTHMEM.

Acknowledgments

We thank the anonymous reviewers, and our shepherd, David Lie, for their feedback. We would also like to thank Úfar Erlingsson and Martín Abadi for several valuable conversations. Taesoo Kim is partially supported by the Samsung Scholarship Foundation.

References

[1] Data Encryption Standard (DES). In *FIPS PUB 46, Federal Information Processing Standards Publication* (1977).

[2] Advanced Encryption Standard (AES). In *FIPS PUB 197, Federal Information Processing Standards Publication* (2001).

[3] ACIIÇMEZ, O., AND ÇETIN KAYA KOÇ. Trace-driven cache attacks on AES. Cryptology ePrint Archive, Report 2006/138, 2006.

[4] ACIIÇMEZ, O., SCHINDLER, W., AND ÇETIN K. KOÇ. Cache based remote timing attack on the AES. In *Topics in Cryptology – CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007* (2007), Springer-Verlag, pp. 271–286.

[5] AMAZON, INC. Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2`, 2012.

[6] AMD, INC. *AMD64 Architecture Programmer's Manual*. No. 24594. December 2011.

[7] AVIRAM, A., HU, S., FORD, B., AND GUMMADI, R. Determinating timing channels in compute clouds. In *Proceedings of the 2010 ACM Cloud Computing Security Workshop* (2010), pp. 103–108.

[8] BERNSTEIN, D. J. Cache-timing attacks on AES. Available at: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, 2005.

[9] BONNEAU, J., AND MIRONOV, I. Cache-collision timing attacks against AES. In *Proceedings of the 8th International Workshop on Cryptographic Hardware and Embedded Systems* (2006), pp. 201–215.

[10] BRICKELL, E., GRAUNKE, G., NEVE, M., AND SEIFERT, J.-P. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. IACR ePrint Archive, Report 2006/052, 2006.

[11] COPPENS, B., VERBAUWHEDE, I., BOSSCHERE, K. D., AND SUTTER, B. D. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy* (2009), pp. 45–60.

[12] DOMNITSER, L., JALEEL, A., LOEW, J., ABU-GHAZALEH, N., AND PONOMAREV, D. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization 8*, 4 (Jan. 2012), 35:1–35:21.

[13] ERLINGSSON, Ú., AND ABADI, M. Operating system protection against side-channel attacks that exploit memory latency. Tech. Rep. MSR-TR-2007-117, Microsoft Research, August 2007.

[14] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache Games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (May 2011), pp. 490 –505.

[15] HU, W. M. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy* (1991), pp. 8–20.

[16] INTEL, INC. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. No. 253669-033US. December 2009.

[17] INTEL, INC. Advanced Encryption Standard (AES) Instructions Set. `http://software.intel.com/file/24917`, 2010.

[18] ION, I., SACHDEVA, N., KUMARAGURU, P., AND ČAPKUN, S. Home is safer than the cloud! Privacy concerns for consumer cloud storage. In *Proceedings of the Seventh Symposium on Usable Privacy and Security* (2011), pp. 13:1–13:20.

[19] JALEEL, A. Memory characterization of workloads using instrumentation-driven simulation – a pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites. Tech. rep., VSSAD, 2007.

[20] JANSEN, W., AND GRANCE, T. Guidelines on security and privacy in public cloud computing. NIST Special Publication 800-144, December 2011.

[21] KIM, S., CHANDRA, D., AND SOLIHIN, Y. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques* (2004), pp. 111–122.

[22] KOCHER, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology* (1996), pp. 104–113.

[23] KONG, J., ACIIÇMEZ, O., SEIFERT, J.-P., AND ZHOU, H. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM Workshop on Computer Security Architectures* (2008), pp. 25–34.

[24] KONG, J., ACIIÇMEZ, O., SEIFERT, J.-P., AND ZHOU, H. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proceedings of the 15th International Conference on High Performance Computer Architecture* (2009), pp. 393–404.

[25] MANGALINDAN, J. Is user data safe in the cloud? `http://tech.fortune.cnn.com/2010/09/24/is-user-data-safe-in-the-cloud`, September 2010.

[26] MICROSOFT, INC. Microsoft Azure Services Platform. `http://www.microsoft.com/azure/`.

[27] MOSCIBRODA, T., AND MUTLU, O. Memory performance attacks: denial of memory service in multi-core systems. In *Proceedings of the 16th USENIX Security Symposium* (2007), pp. 257–274.

[28] MÜLLER, T., DEWALD, A., AND FREILING, F. C. AESSE: a cold-boot resistant implementation of AES. In *Proceedings of the Third European Workshop on System Security* (2010), pp. 42–47.

[29] NEVE, M., AND SEIFERT, J.-P. Advances on access-driven cache attacks on AES. In *Selected Areas in Cryptography*, vol. 4356. 2007, pp. 147–162.

[30] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers Track at the RSA Conference 2006* (2005), pp. 1–20.

[31] PAGE, D. Theoretical use of cache memory as a cryptanalytic side-channel. Tech. Rep. CSTR-02-003, Department of Computer Science, University of Bristol, June 2002.

[32] PERCIVAL, C. Cache missing for fun and profit. In *BSDCan 2005* (Ottawa, 2005).

[33] RAJ, H., NATHUJI, R., SINGH, A., AND ENGLAND, P. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM Cloud Computing Security Workshop* (2009), pp. 77–84.

[34] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), pp. 199–212.

[35] SCHNEIER, B. The Blowfish encryption algorithm. `http://www.schneier.com/blowfish.html`.

[36] SCHNEIER, B. The Blowfish source code. `http://www.schneier.com/blowfish-download.html`.

[37] (SPEC), S. P. E. C. The SPEC CPU 2006 Benchmark Suite. `http://www.specbench.org`.

[38] SUZAKI, K., IIJIMA, K., YAGI, T., AND ARTHO, C. Memory deduplication as a threat to the guest OS. In *Proceedings of the Fourth European Workshop on System Security (EUROSEC '11)* (2011), pp. 1:1–1:6.

[39] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology 23*, 2 (2010), 37–71.

[40] TSUNOO, Y., SAITO, T., SUZAKI, T., AND SHIGERI, M. Cryptanalysis of DES implemented on computers with cache. In *Proceedings of the 2003 Cryptographic Hardware and Embedded Systems* (2003), pp. 62–76.

[41] WANG, Z., AND LEE, R. B. Covert and side channels due to processor architecture. In *Proceedings of the 22nd Annual Computer Security Applications Conference* (December 2006), pp. 473 –482.

[42] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th International Symposium on Computer Architecture* (2007), pp. 494–505.

[43] WANG, Z., AND LEE, R. B. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture* (2008), pp. 83–93.

[44] XU, Y., BAILEY, M., JAHANIAN, F., JOSHI, K., HILTUNEN, M., AND SCHLICHTING, R. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 2011 ACM Cloud Computing Security Workshop* (2011), pp. 29–40.

[45] ZHANG, Y., JUELS, A., OPREA, A., AND REITER, M. K. Home-Alone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (2011), pp. 313–328.