Operating system protection against side-channel attacks that exploit memory latency

Úlfar Erlingsson
Microsoft Research Silicon Valley and
Reykjavík University

Martín Abadi
Microsoft Research Silicon Valley and
University of California, Santa Cruz

August, 2007

MSR-TR-2007-117

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

1. Preamble

The following notes were written in the first half of 2006. Although we have not developed them further into a polished paper, we suspect that, even in their current state, they may be of some interest to others. We resist the temptation to make small changes to these notes, except for the addition of a bibliographic section that maps URLs to more stable names.

2. Introduction

There is a new class of attacks (in particular against AES encryption) that allows a party that is able to execute code on a hardware system to learn critical data (such as AES encryption keys) used by other users of the system. The attacks come in several subclasses but they all depend on information being leaked through the timing behavior of memory systems that use caches.

We can add to operating systems and hypervisors (anything that manages physical memory and page tables) support for protection against these cache-effect side channel attacks, such as those used to attack AES encryption, etc. The following document describes ideas for how this can be done, both in the abstract, and also with some concrete implementation details.

See http://www.wisdom.weizmann.ac.il/~tromer/papers/cache.pdf , http://cr.yp.to/papers.html#cachetiming , and http://www.daemonology.net/hyperthreading-considered-harmful/ for details on these attacks. These papers also suggest some mitigations for these attacks. Many of those mitigations are either draconian, or require hardware changes. We haven't read these papers very closely, but to the extent we understand them, their proposed operating-system-based mitigations are both more simplistic and less practical than ours.

Throughout this document we talk about "encryption" as the activity to be protected—with the idea of preventing key information to be exposed through the ciphers use of "sbox" tables—but the ideas should apply to other activities and memory regions without change.

Our idea is that this type of protection might be applied in practice to the Windows cryptographic service providers, and possibly the CryptoAPI libraries. Also, it might be very useful for the Viridian to increase protection for certain partitions. The ideas may also apply to the Xbox360, although the attack model may differ there, and there may be other alternative implementations (e.g., placing the Sbox tables into a fixed cache partition—a novel hardware feature on the Xbox).

3. Abstraction

This support can be exposed through nice, clean programming interfaces to allocate "unobservable" or "stealth" memory regions. The support can add value to an operating system, or VMM, and may help support system abstractions (e.g., protect processes with code integrity) and satisfy some requirements (e.g., DRM or NSA). Using our ideas, this support would neither

entail a great increase in OS/VMM complexity, nor a great performance overhead (in fact, when no "encryption" is taking place, there should be no overhead).

We focus on providing protection at the granularity of processes or VMMs, but some of our schemes—in particular dynamic cache preloading, described below—also work in the case where a process/VMM is not uniformly trusted (as may be common in the case of virtual machines, or processes hosting many services).

The API could be something like void* p = AllocateStealthMemory(size_in_bytes). Alternatively, stealth memory might be requested for an identified, separate section of binary executable files, and be created at the time of loading those binaries. Thus, the static tables of AES might in fact be static, read-only "global data" in the source-code and resulting binary, and yet be placed in stealth memory when the binary is loaded. The stealth section of the binary could (at least typically) be backed by the same physical pages for all processes that load that binary. For support from a VMM, these APIs would have to go through the OS to the VMM.

The rest of this document is presented using just one stealth memory region, or one "color" of those regions. One could imagine that there are multiple such regions in use concurrently, and that a certain access control policy applied to them to allow for maximal sharing/performance as well as security. Thus, for instance, all the critical Windows security data could be stored in a stealth memory region of a single color. Alternatively, the granularity could even be that of processes/VMs, with all of their memory either being stealthy or not, etc.

A "colored" API could be void* p = AllocateStealthMemory(size_in_bytes, ID), where ID is the color, or name, of a distinct type of stealth memory. Implicitly, this API could take as extra arguments other information, such as the dynamic principal invoking the API. The policy for colors could then be implemented at run-time through an access control system. E.g., the OS or the VMM may keep an access control matrix that says which processes or VMs can request stealth regions of which colors, and what IDs have already been allocated to each.

4. Terminology

The insights we had are based on how caches relate to physical memory and paging. Some definitions first:

- **Cache line**: Storage that can contain data from memory at physical addresses whose middle bits are a certain value.
- **Pre-image of a cache line**: The set of all **physical** memory addresses, or rather physical memory pages, that can map to this cache line.
- **Cache lineset**: For a K-way associative cache, a set of K cache lines, all of which having the same pre-image.

- **Shadow cache lineset of a physical address**: The cache lineset that this physical address maps to.
- **Shadow cache linesets of physical memory pages**: The union of the shadows of the address in those physical pages.
- **Shadow of virtual memory pages**: The shadow cache linesets of the physical pages that those virtual memory pages map to.
- **Flush a cache lineset**: In sequence, access a set of K memory addresses with K distinct physical addresses that all map to the cache lineset, in effect evicting the previous contents of the cache lineset.
- **Cache preloading of a memory region**: For a memory region, loading the value at each of the region's physical memory address into the cache lineset for that address.

5. Schemes

The idea is to implement the abstraction of stealth memory, which is—either dynamically, or statically—prevented from exposing information about its usage pattern via the side channel of the state of the cache linesets in its shadow. The OS/VMM abstractions involved in doing this are (1) the physical memory page manager, (2) the context switcher, and (3) the page fault handler.

What follows is explained using only single-page stealth memory regions. The ideas apply equally well to smaller, sub-page regions (at least, as long as the page in question is only used for the stealth memory). On the other hand, multi-page regions may be supported up to a point (potentially with some restrictions, e.g., they may have to be contiguous in physical memory). However, for some of our schemes (and with respect to some attacks), stealth memory regions are difficult to support if they are too large: e.g., then they cannot be loaded in their entirety into the cache.

5.1. Static partitioning

One could simply partition the allocation of physical memory pages, e.g., using a physical policy, so that the single stealth memory page was the only one using its cache lineset. That is to say, that virtual memory allocated to processes/VMMs never cast a shadow onto the same cache lineset as that of the stealth memory.

This fixed partitioning is a bit wasteful, as it prevents certain cache linesets, and certain physical memory pages, from ever being used—as long as there is some stealth memory somewhere—even when there is nobody really using the stealth memory (i.e., no encryption is going on).

This implementation would really only need to involve the memory manager.

5.2. Dynamic schemes based on flushing

A more dynamic scheme would allow all physical memory and cache linesets to be used, as long as the process/VMM that allocated the stealth memory wasn't run. This implementation would involve the memory manager and the context-switch code. For each process/VMM, the shadow cache lineset of all of its memory would be maintained. Then, if ever another process/VMM was scheduled that cast a shadow onto the stealth memory's cache lineset, the stealth memory's cache lineset would be flushed.

Note that flushing clears all information from cache linesets, even when the cache values are useful, and not from stealth memory. Also, this scheme might do a lot of unnecessary flushing, even when nobody is using the stealth memory.

Therefore, it might be useful to keep track of whether somebody is using the stealth memory, and avoid flushes unless it had been used since the last flush. This refinement can be done by making use of page-table alerts to selectively choose when to flush the shadow cache lineset, as follows.

Page-table alerts specially mark the stealth memory page in the page-tables, so that its use can be detected. Whenever a context switch takes place to the "owner" process/VMM of a stealth memory page, that page is marked as invalid in the page tables. Then, when the stealth memory is used, a page-fault will happen, and in the page fault handler, a bit is set that marks that the stealth memory has been used recently, and the page-table entry is fixed to be a valid mapping. Then, flushes only need to take place whenever we are context switching away from the "owner" process/VMM, and stealth memory has actually been used.

A further refinement, to avoid flushes, is to not flush at the end of the "owner" process/VMM's context switch timeslice, but either statically or dynamically keep track of whether the processes in subsequent timeslices cast a shadow onto the stealth memory's cache lineset, and only in that case flush the cache lineset. Flushes do not need to happen if such a shadow is never cast, or not cast until the "owner" process/VMM is scheduled again. Also, flushes may not be necessary even when such a shadow is cast by trusted processes in the intermediary timeslices.

5.3. Dynamic schemes based on cache preloading

Perhaps the best schemes may be those that use page-table alerts (see above) to trigger cache preloading of the stealth memory. These schemes have the advantage of needing neither flushing nor fixed partitioning, as well as the extra performance advantage of allowing safe sharing of the cache linesets.

As before, a page-table alert is enabled per timeslice, and it rings out whenever a process/VMM touches its stealth memory. When this happens, the entire stealth memory is read, making it resident in cache lines of its shadow cache lineset.

This will prevent any observer from learning information from partial fills/retention in that cache lineset. Also, under some assumptions, this preloading will prevent certain timing attacks (such as that described for the 5th phase of AES). In particular, it may be necessary to determine that an encryption method using shadow memory doesn't simultaneously use other memory that causes contention for the stealth memory's shadow cache lineset.

5.4. Hybrid and variant schemes

For systems with multiple levels of memory hierarchy, different schemes could be used at different levels. For instance, stealth memory might never be paged out to disk. Alternatively, the top-level cache might always be flushed after a context slice that used stealth memory.

Of course, when a stealth memory region is requested, the operating system may want to move things around with respect to the existing virtual-to-physical mappings of processes etc. This may be useful for performance, and potentially necessary for security, if a fixed partitioning of cache linesets is being implemented.

Also, it will often be useful to avoid the potential of untrusted memory pages casting a shadow onto the stealth memory's cache lineset. This can be helped by partitioning non-stealth memory into the untrusted and the trusted, and allocating physical pages for trusted non-stealth memory out of the pre-image of the stealth memory's cache lineset. Trusted memory could be the memory of the process that created the stealth memory, or the memory of all processes running as the same principal (e.g., kernel or system in Windows, vs. a regular user).

These schemes may also apply to information leakage via the branch-prediction caches and ALU usage on certain systems, e.g. hyperthreaded ones. (Decompression is an example of an activity that could leak information about the data being used through the branch-prediction caches.)

The ideas of partitioning the shadow cache linesets, whether statically or dynamically, can also be useful to prevent the store/load and cache bank latency variations in L1 cache accesses described by Dan Bernstein in http://cr.yp.to/papers.html#cachetiming.

5.5. Concurrency considerations

The above is written for traditional computers: a single CPU with some caches, timesliced, by an OS or VMM. For multi-core and hyperthreaded systems, that use shared caches, the OS/VMM might look at what processes are running on each context slice, or whenever stealth memory use is signaled via a page-table alert. Then, an exclusion policy could be implemented using a "incompatibility" notion between processes/VMMs, that kicked certain activity off cores/SMTs when activity was occurring that used stealth memory. One can also try other means (e.g., static) of avoiding having any untrusted memory pages cast a shadow onto the stealth memory's cache lineset.

Partitioning can be done dynamically at the time of stealth memory page-table alert. Then the CPU getting the alert can ask other CPUs to relocate physical memory of their currently running process so that it doesn't cast a shadow onto the stealth memory's cache linesets. This could even be done partially, just on the working set, for performance.

More interestingly, such relocation could be done fully dynamically by using the TLB trick we used for SoftNX. By having the untrusted process's CPU flush its TLBs (which are per process, whether multicore or hyperthreaded), one could guarantee that a "possible conflict alert" would sound whenever the untrusted process might be observing cache effects related to the stealth memory. Then, upon such alerts, a range of options is possible: the OS could kick off the process, do the memory access for it in a constant-time, super-slow manner, or it could re-map to a different underlying physical page to enforce a partitioning.

Different strategies will most likely be best for different hardware systems: single core (no concurrent cache sharing), multi core (concurrently shared L2 caches), and hyperthreading (all caches concurrently shared). The solutions will range, e.g., from cache preloading, with its minimal performance effects, to static partitioning and its larger performance effect.

The net effect of using stealth memory page-table alerts—in combination with the above schemes for handling concurrency—is to make protection be zero cost whenever stealth memory is not being used, and only incur slowdown for the fraction of the systems activity that might possible leak information over the side channel. In particular, dynamic relocation using the above "TLB trick" can be done without touching the memory manager at all, avoiding the most complicated bit of any implementation.

References

http://cr.yp.to/papers.html#cachetiming
"Cache-timing attacks on AES", by Daniel J. Bernstein, 2005.

http://www.wisdom.weizmann.ac.il/~tromer/papers/cache.pdf
"Cache Attacks and Countermeasures: the Case of AES",
by Dag Arne Osvik, Adi Shamir, and Eran Tromer, 2005.

http://www.daemonology.net/hyperthreading-considered-harmful/
"Hyper-Threading Considered Harmful", by Colin Percival, 2005.