CACHE MISSING FOR FUN AND PROFIT

COLIN PERCIVAL

Abstract. Simultaneous multithreading — put simply, the idea of sharing the execution resources of a superscalar processor between multiple execution threads — has recently become widespread via its introduction (under the name "hyper-threading") into Intel Pentium 4 processors. In this implementation, for reasons of efficiency and economy of processor area, while some resources (e.g., architectural registers) are duplicated for each thread, some resources are divided between the threads (e.g., instruction TLB entries), and yet others — including memory caches — are shared.

We demonstrate that this last characteristic — the sharing of caches — provides not only an easily used high bandwidth covert channel between threads, but also permits a malicious thread — operating, in theory, with limited privileges — to monitor the execution of another thread, allowing in some cases for theft of cryptographic key data.

Finally, we provide some suggestions to processor designers, operating system vendors, and the authors of cryptographic software, of how this attack could be mitigated or eliminated entirely.

1. Introduction

As integrated circuit fabrication technologies have improved, providing not only faster transistors, but smaller transistors, processor designers have been met with two critical challenges. First, memory latencies have increased dramatically in relative terms; and second, while it is easy to spend extra transistors on building additional execution units, many programs have fairly limited instruction-level parallelism, which limits the extent to which additional execution resources can be utilized. Caches provide a partial solution to the first problem, while out-of-order execution provides a partial solution to the second.

In 1995, simultaneous multithreading was revived[1] in order to combat these two difficulties [13]. Where out-of-order execution allows

Key words and phrases. Side channels, Simultaneous multithreading, caching.

[1]Simultaneous multithreading had existed since at least 1974 in theory [11], even if it had not yet been shown to be practically feasible.

instructions to be reordered (subject to maintaining architectural semantics) within a narrow window of perhaps a hundred instructions, simultaneous multithreading allows instructions to be reordered across threads; that is, rather than having the operating system perform context switches between two threads, it can schedule both threads simultaneously on the same processor, and instructions will be interleaved, dramatically increasing the utilization of existing execution resources.

On the 2.8GHz Intel Pentium 4 processor with hyperthreading [9], with which the remainder of this paper is concerned[2], the two threads being executed on each processor share more than merely the execution units; of particular concern to us, they share access to the memory caches. Caches have already been demonstrated to be cryptographically dangerous: Many implementations of AES [10] are subject to timing attacks arising from the non-constancy of S-box lookup timings [1]. However, having caches shared between threads provides a vastly more dangerous avenue of attack.

2. Covert communication via paging

To see how shared caches can create a cryptographic side-channel, we first step back for a moment to a simpler problem — covert channels [8] — and one of the classic examples of such a channel: virtual memory paging.

Consider two processes, known as the Trojan process and the Spy process, operating at different privilege levels on a system with mandatory access control, but both with access to some large reference file (naturally, mandatory access control implies that this access would be read-only). The Trojan process now reads a set of pages from this reference file, resulting in page faults which load the pages from disk into memory. Once this is complete (or even in the midst of this operation) the Spy process reads each page of the reference file and measures the time taken for each memory access. Attempts to read pages which have been previously read by the Trojan process will complete very quickly, while those pages which have not already been read will incur the (easily measurable) cost of a disk access. In this manner, the Trojan process can repeatedly communicate one bit of information to the Spy process in the time it takes for a page to be loaded from disk into memory, up to a total number of bits equal to the size (in pages) of the shared reference file.

[2]We examine this particular processor for reasons of availability, but expect the results to apply equally to all processors from the "Pentium 4 with hyperthreading" line.

If the two processes do not share any reference file, this approach will not work, but instead an opposite approach may be taken: Instead of faulting pages *into* memory, the Trojan process can fault pages *out* of memory. Assume that the Trojan and Spy processes each have an address space of more than half of the available system memory and the operating system uses a least-recently-used page eviction strategy. To transmit a "one" bit, the Trojan process reads its entire address space; to transmit a "zero" bit, the Trojan process spins for the same amount of time while only accessing a single page of memory. The Spy process now repeatedly measures the amount of time needed to read its entire address space. If the Trojan process was sending a "one" bit, then the operating system will have evicted pages owned by the Spy process from memory, and the necessary disk activity when those pages are accessed will provide an easily measurable time difference. While this covert channel has a far more limited bandwidth than the previous channel — it operates at a fraction of a bit per second, compared to a few hundred bits per second — it demonstrates how a shared cache can be used as a covert channel, even if the two communicating processes do not have shared access to any potentially cached data.

3. L1 cache missing

The L1 data cache in the Pentium 4 consists of 128 cache lines of 64 bytes each, organized into 32 4-way associative sets. This cache is completely shared between the two execution threads; as such, each of the 32 cache sets behaves in the same manner as the paging system discussed in the previous section: The threads cannot communicate by loading data *into* the cache, since no data is shared between the two threads[3], but they can communicate via a timing channel by evicting each other's data from the cache.

A covert channel can therefore be constructed as follows: The Trojan process allocates an array of 2048 bytes, and for each 32-bit word it wishes to transmit, it accesses byte $64i$ of the array iff bit i is set. The Spy process allocates an array of 8192 bytes, and repeatedly measures the amount of time needed to read bytes $64i$, $64i + 2048$, $64i + 4096$, and $64i + 6144$ for each $0 \le i < 32$. Each memory access performed by the Trojan will evict a cache line owned by the Spy, resulting in

[3]By default, cache lines are tagged according to which thread "owns" them and cannot be accessed by the other thread; this behaviour may be modified by the operating system, but only to the extent of allowing cache line sharing between threads with the same paging tables, and such threads already have much easier means of communication.

```
        mov ecx, start_of_buffer
        sub length_of_buffer, 0x2000
        rdtsc
        mov esi, eax
        xor edi, edi

loop:
        prefetcht2 [ecx + edi + 0x2800]

        add cx, [ecx + edi + 0x0000]
        imul ecx, 1
        add cx, [ecx + edi + 0x0800]
        imul ecx, 1
        add cx, [ecx + edi + 0x1000]
        imul ecx, 1
        add cx, [ecx + edi + 0x1800]
        imul ecx, 1

        rdtsc
        sub eax, esi
        mov [ecx + edi], ax
        add esi, eax
        imul ecx, 1

        add edi, 0x40
        test edi, 0x7C0
        jnz loop

        sub edi, 0x7FE
        test edi, 0x3E
        jnz loop

        add edi, 0x7C0
        sub length_of_buffer, 0x800
        jge loop
```

FIGURE 1. Example code for a Spy process monitoring the L1 cache.

lines being reloaded from the L2 cache[4], which adds an additional latency of approximately 30 cycles if the memory accesses are dependent. This alone would not be measurable, thanks to the long latency of the RDTSC (read time stamp counter) instruction, but this problem is resolved by adding some high-latency instructions – for example, integer multiplications – into the critical path. In Figure 1 we show an example of how the Spy process could measure and record the amount of time required to access all the cache lines of each set.

Using this code, 32 bits can be reliably transmitted from the Trojan to the Spy in roughly 5000 cycles with a bit error rate of under 25%; using an appropriate error correcting code, this provides a covert channel of 400 kilobytes per second on a 2.8GHz processor.

4. L2 cache missing

The same general approach is effective in respect of the L2 cache, with a few minor complications. The Pentium 4 L2 cache (on the particular model which we are examining) consists of 4096 cache lines of 128 bytes each, organized into 512 8-way associative sets. However, the data TLB holds only 64 entries — only enough to provide address mappings for half of the cached data[5]. As a result, a Spy process operating in the same manner as described in the previous section will incur the cost of TLB misses on at least some of its memory accesses. To avoid allowing this to add noise to the measurements, we can resort to ensuring that *every* memory access incurs the cost of a TLB miss, by accessing each of the 128 pages (512kB divided by 4kB per page) before returning to the first page and accessing the second cache line it contains. (Another option would use a buffer of 16 MB, placing each potentially cached line into a separate page, but accessing the lines in a suitable order is just as effective.) Since the TLB entries have to be repeatedly reloaded, however, we also experience some additional cache misses, as the memory holding the paging tables will be repeatedly reloaded into the cache. Fortunately, this will only affect a small number of cache lines, leaving the vast majority of the cache in full working covert-channel order.

Another complication is introduced by the design of the Pentium 4 as a streaming processor. The "Advanced Transfer Cache" includes a capability for hardware prefetching: If a series of cache misses occur, in arithmetic progression, within a single page, then the cache will

[4]In fact, thanks to the pseudo-LRU algorithm for cache line replacement, one memory access by the Trojan process causes four cache misses by the Spy.

[5]Unless 4MB pages are used; but these are often not available to user processes.

"recognize" this as a data stream and prefetch two additional cache lines. This is quite effective for reducing cache misses; but since we instead want to maximize cache misses, it becomes a disadvantage. Here we can simply trust to luck: If we access cache lines in an irregular manner (e.g., following a *de Bruijn* cycle rather than accessing the lines in increasing address order), then it is unlikely that we will exhibit three or more cache misses in arithmetic progression, and the hardware prefetcher will not activate.

Finally, since the L2 cache is used for both data and code, there will be some inevitable cache collisions (and line evictions) caused by the instruction fetching activity.

Due to the lower memory bandwidth, increased size of L2 cache sets (8 lines of 128 bytes, vs. 4 lines of 64 bytes in the L1 cache), and noise introduced by memory activity associated with TLB misses and instruction fetching, the L2 cache provides a significantly lower bandwidth covert channel than the L1 cache. In roughly 350000 cycles (on the same machine as previously used), 512 bits can be transmitted with an error rate of under 25%; this provides a channel of approximately 100 kilobytes per second.

Despite the reduced bandwidth, however, the L2-collision covert channel is potentially more interesting than the L1-collision channel: On systems without symmetric multithreading — i.e., where the Trojan and Spy processes are always separated by context switches — the contents of the L1 cache will tend to be fairly comprehensively replaced between schedulings of the Spy process; the L2 cache however, due to its larger size, is often not completely replaced, allowing it to be used as a covert channel with a bandwidth of a several bits per context switch. On an otherwise quiescent system this could easily provide a covert channel of a few kilobits per second, and several times that if the kernel makes the POSIX `sched_yield(2)` system call [4] available.

5. OpenSSL key theft

Having demonstrated the effectiveness of this cache-missing approach in the construction of a covert channel, we now examine it as a cryptanalytic side channel. Taking as a demonstration platform OpenSSL 0.9.7c [12] running on FreeBSD 5.2.1-RELEASE-p13 [3], we perform a 1024-bit private RSA operation in one process (via the command `openssl rsautl -inkey priv.key -sign`), while running the L1 Spy process described in Section 3. To ensure that the two processes run simultaneously, we start running the Spy process before we start OpenSSL, and stop it after OpenSSL has finished, while minimizing the

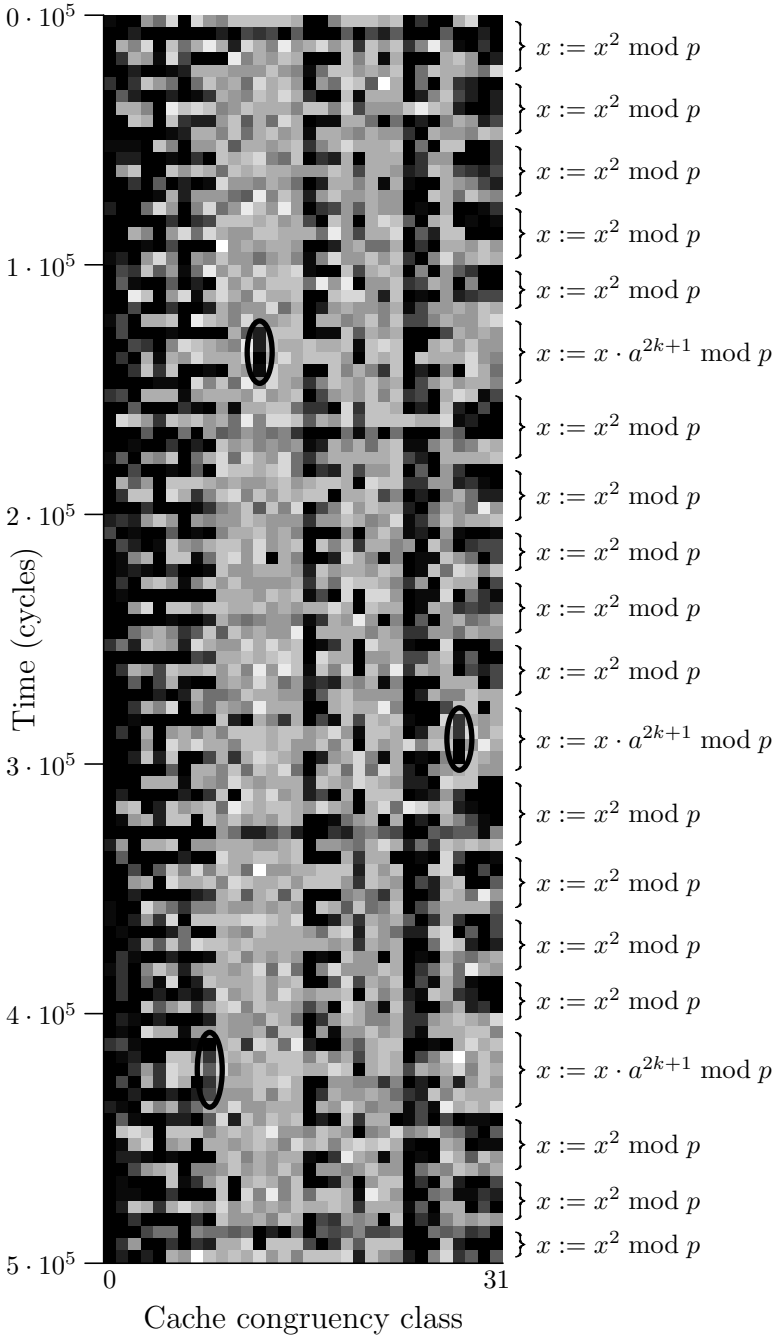

FIGURE 2. Part of a 512-bit modular exponentiation in OpenSSL 0.9.7c. The shading of each block indicates the number of cycles needed to access all the cache lines in a congruence class, ranging from 120 cycles (white) to over 170 (black).

number of other processes running; without these steps, an attacker might need to make several attempts before he successfully "spies" upon OpenSSL.

Since OpenSSL uses the Chinese Remainder Theorem [6] when performing private key operations, it computes a 1024-bit modular exponentiation over $\mathbb{Z}_{pq}$ using two 512-bit modular exponentiations over the rings $\mathbb{Z}_p$ and $\mathbb{Z}_q$. Further, OpenSSL utilizes a "sliding window" method of modular exponentiation, decomposing $x := a^d \bmod p$ into a series of squarings $x := x^2 \bmod p$ and multiplications $x := x \cdot a^{2k+1} \bmod p$, using a set of precomputed multipliers $\{a, a^3, a^5 \ldots a^{31}\} \bmod p$.

In Figure 2 we show a small portion of one such modular exponentiation, as seen from the perspective of the L1 Spy process. The modular squarings and modular multiplications are easily distinguishable here; this difference results from the use of the `BN_sqr` vs. `BN_mul` functions: `BN_sqr` is slightly faster, but uses a different temporary working space for performing Karatsuba multiplication [5] and consequentially leaves a different "footprint" behind in the cache.

From the sequence of multiplications and squarings, we can typically obtain about 200 bits out of each 512-bit exponent — for each multiplication we can infer a 1 bit, since the multipliers are all odd powers, and any time we have more than five squarings without an intervening multiplication, we can infer the presence of 0 bits, since the multipliers are of degree at most 31.

This alone is not quite enough to make factoring the RSA modulus $N = pq$ easy — we need the low 256 bits of either exponent [2] or more than half the bits randomly selected from both exponents[6] — but additional bits can be obtained by close inspection of the "footprint" left behind by the multiplications $x := x \cdot a^{2k+1} \bmod p$. These multipliers are precomputed at the start of the modular exponentiation, and we can easily identify the locations where they are stored by examining the footprints left in the cache during this process; each multiplication thereafter will then load the appropriate multiplier out of memory, indicating to us — if we are lucky — which multiplier is being used.

Obtaining these exponent bits is made non-trivial by the "noise" from the modular multiplications — we cannot distinguish between a cache eviction resulting from a multiplier a^{2k+1} being accessed and an eviction which results from the fixed memory-access pattern of the modular multiplication if they are mapped to the same cache set —

[6]To see why this is sufficient, consider the set of possible pairs $(p, q) \bmod 2^j$ as j increases — given more than half of the bits, this set can be pruned to avoid exponential growth.

and even once the correct cache set has been identified, the multiplier is often not uniquely determined; but in the case of OpenSSL we have typically been able to identify the multiplier to within two possibilities in 50% of the modular multiplications. This provides us with an additional 110 bits from each exponent on average, for a total of 310 out of 512 bits, allowing the RSA modulus N to be factored.

6. Solutions

A variety of methods are available to combat this attack. Most obviously, eliminating all instances of shared data caches will close both the side channel and the covert channel. This would require disabling the simultaneous multithreading capabilities of processors — which, for current processors is of little cost[7] — but also would require that the processor caches be flushed each time a different thread is scheduled (in order to avoid information leakage via the L2 cache), which could significantly impact performance. (Alternatively, future processors supporting simultaneous multithreading could be designed to split their caches rather than sharing them; but that does little to benefit users of existing hardware.)

Another option is simply to disallow the execution of untrusted code on any system which manipulates sensitive data (since login credentials usually constitute "sensitive data", this includes most multi-user systems). Since the repetitive timer measurements needed for this attack to be effective are highly unlikely to occur in "real" applications, this would provide almost certain protection against a side channel attack; it would however still be possible (albeit difficult) to construct a covert channel using applications chosen based on their memory access patterns. Systems with no local untrusted users are not necessarily immune to these attacks, but they are certainly at very low risk.

Perhaps the best option as far as cryptographic software is concerned is to rely exclusively upon oblivious algorithms — that is, algorithms which perform exactly the same series of operations and memory accesses, regardless of their input data. While this diverges quite significantly from the approaches taken by existing implementations[8], and would make certain algorithmic optimizations unavailable (e.g., the "sliding window" method of modular exponentiation, which typically

[7] While Intel's Hyperthreading has been shown to significantly improve performance on some benchmarks, it offers little, or even negative, performance benefit on many applications which have not been designed for the technology.

[8] In OpenSSL, the large integer arithmetic code alone contains over a thousand "if" statements.

reduces running time by roughly 40%), we believe that this is likely to be the most effective solution in the long run. As a side benefit, implementations utilizing oblivious algorithms are immediately immune to all timing attacks (providing that the processor does not have data-dependent instruction timings).

If none of these approaches can be used, some others can at least make an attack more difficult. While previous timing attacks against RSA [7] have been evaded by "blinding" of RSA inputs, this leaves the exponent unchanged and does nothing to defend against our attack; however, the computation of $a^d \bmod p$ could be replaced by the computation of $a^{d+k(p-1)} \bmod p$ for some random value k. Providing that k is sufficiently large as to make statistical attacks impossible (e.g., a random 40-bit value), this should be secure; however, we consider this to be a rather dangerous approach, given that the extent to which exponent bits can be obtained is not entirely clear. It should also be noted that this defence applies only to RSA, and it is entirely possible that this same side channel could be effectively applied to other applications.

Another form of randomization can be performed with memory addresses: Since this attack works by monitoring memory accesses, a program which randomizes the locations of structures — and repeatedly randomly rearranges them — will likely be immune to attack. However, such repetitive rearrangement would both be programmatically cumbersome and very slow.

Finally, the traditional method of closing covert timing channels is available: Access to the clock — in this case, the time stamp counter — can be removed[9]. However, this is only an option on single-processor systems: On multi-processor systems, a "virtual" time stamp counter with sufficient precision could be obtained by utilizing a second thread which repeatedly increments a memory location. Even on uniprocessor systems, this approach should not be taken lightly however, since many applications expect the time stamp counter to be available, either for profiling purposes, or to be used in combination with a random stream of events (e.g., key presses) as a source of entropy.

7. Notes to self

We've got a problem. We probably don't need to worry about information leakage via the L2 cache across context-switches, since the time resolution would be too low for an attack on cryptographic systems;

[9]Intel has helpfully provided a "time stamp disable" bit in all of its recent processors.

but hyperthreading is definitely in need of fixing, both in terms of the L1 cache and the L2 cache — I haven't shown this here, but with a different piece of RSA code I've extracted a key via eviction detection in a shared L2 cache.

It's important to note that while timing attacks require that a cryptographic operation is performed many times, this attack only requires that a single RSA private key operation be observed, followed by analysis of the "footprint" and the necessary computation to factor N (both of which can be done off-line).

Disabling hyperthreading entirely should be the best option, but there are apparently some interrupt routing issues with broken BIOSes resulting in interrupts being sent to processor #3 even when that processor doesn't exist. (Paul Saab says that this problem affects FreeBSD but would not affect Windows; I have no idea why there is any difference or whether Linux has this problem.)

Given the possibility of interrupt routing issues, the best option might be to tell the scheduler not to schedule any userland processes / threads on the the second of each pair of virtual processors. Allowing interrupts to run isn't a problem since they (hopefully!) won't be handling sensitive data in a non-oblivious manner and they (hopefully!) would execute in far less than the minimum number of cycles needed to make this attack practical (it's probably not possible to steal any useful data in less than $5 \cdot 10^4$ cycles $\approx 20\mu s$, and I think interrupts handlers tend to be faster than this).

I think some IA64 processors have HyperThreading support; if so, they're probably also affected.

The CVE name CAN-2005-0109 has been provisionally allocated for this issue.

References

[1] D.J. Bernstein. Cache-timing attacks on AES, 21 Nov 2004. Document ID: cd9faae9bd5308c440df50fc26a517b4.

[2] D. Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In U. Maurer, editor, *Advances in Cryptology - EUROCRYPT '96*, LNCS 1070, pages 178–189. Springer-Verlag, 1996.

[3] FreeBSD Project. The FreeBSD operating system. `http://www.freebsd.org/`.

[4] IEEE Std 1003.1. 2004 Edition.

[5] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics - Doklady*, 7:595–596, 1963.

[6] D.E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison Wesley, third edition, 1997.

[7] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO '96*, LNCS 1109, pages 104–113. Springer-Verlag, 1996.

[8] B.W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[9] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture: A hypertext history. *Intel Technology Journal*, February 2002. `http://developer.intel.com/technology/itj/2002/volume06issue01/`.

[10] National Institute of Standards and Technology. Announcing the Advanced Encryption Standard (AES). NIST FIPS PUB 197, U.S. Department of Commerce, 2001.

[11] L.E. Shar and E.S. Davidson. A multiminiprocessor system implemented through pipelining. *IEEE Computer*, 7(2):42–51, 1974.

[12] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. `http://www.openssl.org/`.

[13] D.M. Tullesen, S. Eggers, and H.M. Levy. Simultanous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

IRMACS Centre, Simon Fraser University, Burnaby, BC, Canada
E-mail address: `cperciva@freebsd.org`