CACHE MISSING FOR FUN AND PROFIT

COLIN PERCIVAL

ABSTRACT. Simultaneous multithreading — put simply, the sharing of 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, the sharing of processor resources between threads extends beyond the execution units; of particular concern is that the threads share access to the memory caches.

We demonstrate that this shared access to memory 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 many cases for theft of cryptographic keys.

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 [12]. Where out-of-order execution allows instructions to be reordered (subject to maintaining architectural semantics) within a narrow window of perhaps a hundred instructions,

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

[1]Simultaneous multithreading had existed since at least 1974 in theory [10], even if it had not yet been shown to be practically feasible.

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 2.8 GHz Intel Pentium 4 with Hyper-Threading processor, 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 [8]. Caches have already been demonstrated to be cryptographically dangerous: Many implementations of AES [9] 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 [7] — 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 multilevel secure system, but both with access to some large reference file (naturally, on a multilevel secure system this access would necessarily be read-only). The Trojan process now reads a subset of pages in this reference file, resulting in page faults which load the selected pages from disk into memory. Once this is complete (or even in the middle of this operation) the Spy process reads *every* 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 the 2.8 GHz Intel Pentium 4 with Hyper-Threading processor for reasons of availability, but expect that the results in this paper will apply equally to all processors with the same simultaneous multithreading and memory cache design.

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 far lower 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 forcing each other's data *out* of 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 of the word 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 $0 \leq i < 32$. Each memory access performed by the Trojan will evict a cache line owned by the Spy,

[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 can already communicate.

```
        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.

resulting in 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 2.8 GHz 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 (512 kB divided by 4 kB 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 4 MB 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 [11] 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

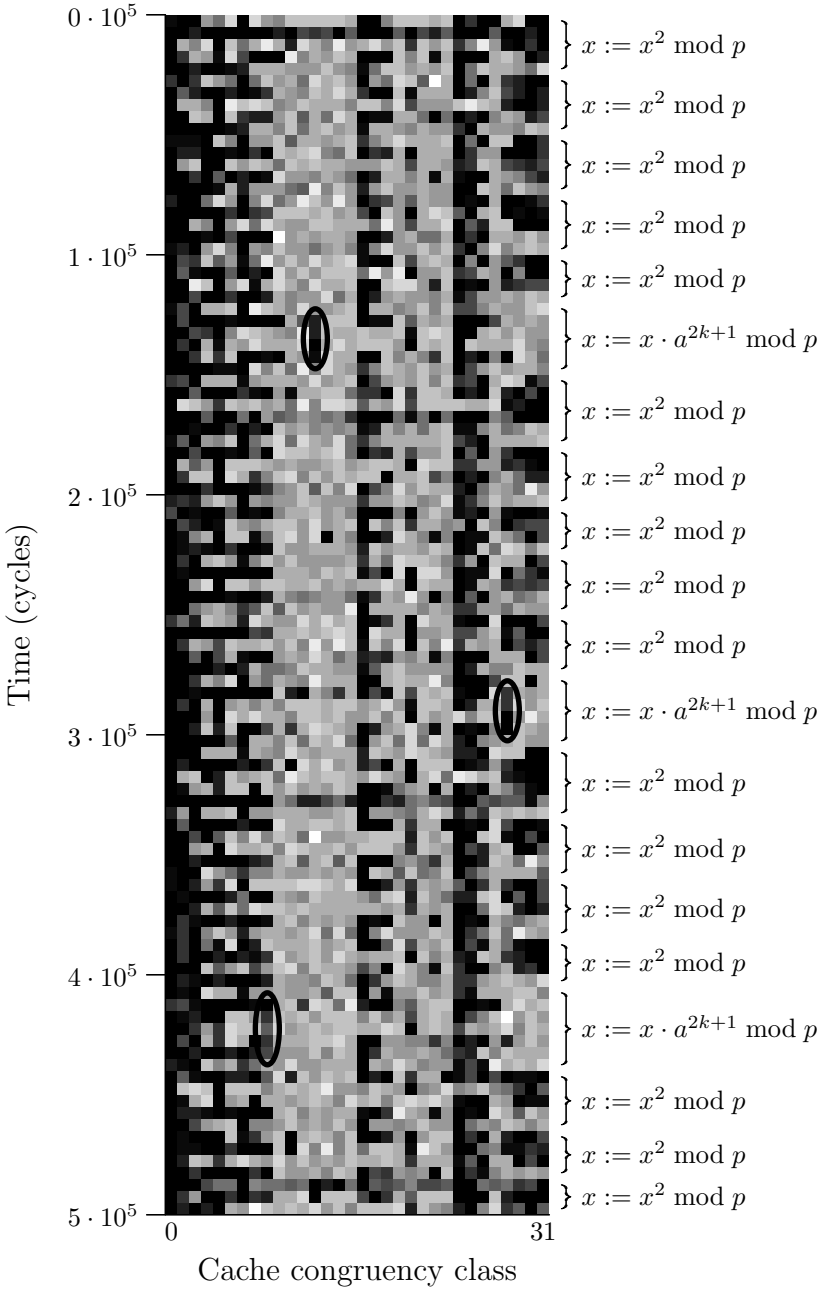

FIGURE 2. Part of a 512-bit modular exponentiation in OpenSSL 0.9.7c. The shading of each block indicates the number of cycles needed to access all the lines in a cache set, ranging from 120 cycles (white) to over 170 (black). The circled regions reveal information about the multipliers a^{2k+1} being used.

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 subsequent multiplication 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 pairs (p, q) satisfying $pq \equiv N \pmod{2^j}$ as j increases — given more than half of the bits of p and q, this set can be repeatedly 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 can typically 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 easily factored.

6. Solutions and workarounds

Both the simultaneous multithreading covert channel and the associated side channel can be easily blocked by processor designers. In addition to the trivial option of not utilizing simultaneous multithreading, if the data caches are split into separate parts, with no sharing of resources between threads, then the channel would vanish. More interestingly, the covert channel can be almost completely removed, and the side channel can be made small enough as to be cryptologically insignificant, if the cache eviction logic is changed: Rather than using a single pseudo-LRU cache eviction strategy, the cache eviction logic could be made thread-aware and only allow thread A to evict a cache line "owned" by thread B if thread B currently "owns" more than its "fair share"(i.e., one half on current processors) of the cache lines in the set[7]. As we lack expertise in the field of microarchitecture, we cannot comment upon whether such a strategy would be practical in such a performance-critical path as the L1 data cache.

These channels can also be closed by the operating system. Again, simultaneous multithreading could be disabled; but a better solution can be provided by the kernel scheduler. Recognizing that a side channel between threads is only dangerous if the threads are operating at different privileges — or, put another way, if the threads are not permitted to debug each other — the scheduler could be written in such a way as to use the credentials of threads in the process of determining which threads should be scheduled on which (virtual) processors. There are some potential dangers in this approach, however: Since the credentials of a thread can be changed during a system call, the kernel would have to re-evaluate whether a set of threads are allowed to share a processor core at several different points, which could lead to a loss of performance, the introduction of bugs, and quite possibly difficulties involving the locking of kernel data.

In some cases, this side channel can also be closed at the application level. If applications and libraries are written in such a manner that the

[7]Naturally, we would also require that cache lines are marked as "not owned" upon a context switch.

code path and sequence of memory accesses are oblivious to the data and key being used, then all timing side channels are immediately and trivially closed[8]. This would be a dramatic divergence from existing practice[9], and would require that some existing algorithms be thrown out or reworked considerably (e.g., the "sliding window" method of modular exponentiation), which could impact performance; however, in the specific case of RSA private key operations this loss of performance can be as little as 10%. This approach has the disadvantage of requiring a vast amount of code to be audited if it is to be carried out comprehensively (since any application which could conceivably be used to process senstive data would need to be inspected), but a good first step would be to rewrite the most obvious target for such attacks, namely cryptographic libraries.

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. 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, 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. A somewhat more tolerable approach would limit the frequency with which the time stamp counter could be read — say, to a maximum of four times within any 10000 cycle window — which would be very unlikely to affect any "real" software; but this could only be performed via modifications in the microcode, and it is not clear if the necessary modifications would even be possible given existing architectural limitations.

7. Recommendations

Our recommendations are as follows:

1. **CPU designers** should, on all future processors which implement simultaneous multithreading, use cache eviction strategies which respect threading and minimize the extent to which one thread can evict data used by another thread. Similarly, "multi-core" processors should

[8]We assume that the hardware used does not exhibit data-dependent instruction timings; without this assumption, it is impossible to offer any such guarantees.

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

either avoid sharing caches between the processor cores or use thread-aware cache eviction strategies.

We further recommend that "x86" processors implementing Hyper-Threading should use a bit in the processor feature flags register to indicate whether the caches have been designed to close these covert and side channels, in order that operating systems can determine whether countermeasures are necessary.

2. **Operating systems** should first determine if countermeasures need to be taken. If the processor reports that the cache-eviction channels have been closed (as described above) or if the system is known to have no untrusted users, then no action needs to be taken. Otherwise, action must be taken to ensure that no pair of threads execute simultaneously on the same processor core if they have different privileges.

Due to the complexities of performing such privilege checks correctly and based on the principle that security fixes should be chosen in such a way as to minimize the potential for new bugs to be introduced, we recommend that existing operating systems provide the necessary avoidance of inappropriate co-scheduling by never scheduling *any* two threads on the same core, i.e., by only scheduling threads on the first thread associated with each processor core[10]. The more complex solution of allowing certain "blessed" pairs of threads to be scheduled on the same processor core is best delayed until future operating systems where it can be extensively tested.

In light of the potential for information to be leaked across context switches, especially via the L2 and larger cache(s), we also recommend that operating systems provide some mechanism for processes to request special "secure" treatment, which would include flushing all caches upon a context switch. It is not immediately clear whether it is possible to use the occupancy of the cache across context switches as a side channel, but if an unprivileged user can cause his code to pre-empt a cryptographic operation (e.g., by operating with a higher scheduling priority and being repeatedly woken up by another process), then there is certainly a strong possibility of a side channel existing even in the absence of Hyper-Threading.

3. **Cryptographic libraries** should be rewritten to avoid any data-dependent or key-dependent memory access or code path patterns. As noted in the preceeding section, this is important both for the issues

[10]It is possible to disable Hyper-Threading, but it has been reported that some systems have problems with interrupt routing, attempting to send interrupts to a "virtual processor" which doesn't exist when Hyper-Threading is disabled. Consequently, it is safer to leave Hyper-Threading enabled, with all of the threads able to respond to interrupts, and close the security problems via the kernel scheduler.

discussed in this paper and also to eliminate the potential for new timing side channels.

We note, however, that due to the large number of cryptographic libraries in use, and the vastly larger number of applications which use (and distribute) them, it is impractical to attempt to fix all cryptographic libraries and applications in a narrow time frame, so the first line of defence against the particular issues discussed here must come from the operating system.

8. Acknowledgements

The author wishes to thank Jacques Vidrine and Mike O'Connor for their invaluable aid in facilitating communications prior to the public disclosure of this security issue, and the many people who provided helpful feedback during that period.

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

[8] 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/`.

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

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

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

[12] D.M. Tullsen, 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`

Appendix A. Vendor statements

CVE: The Common Vulnerabilities and Exposures (CVE) project has assigned the name CAN-2005-0109 to the problem of information disclosure resulting from cache evictions in simultaneous multithreading processors. This is a candidate for inclusion in the CVE list (`http://cve.mitre.org`), which stadardizes names for security problems.

FreeBSD: This issue affects FreeBSD/i386 and FreeBSD/amd64, and is address in advisory FreeBSD-SA-05:09.htt.

NetBSD: The NetBSD Security-Officer Team believes that workarounds will be suitable for the majority of our users. Since this issue is a complex one, the 'right' solution will require a larger discussion which is only possible once this issue is public. This issue will be addressed in advisory NetBSD-SA2005-001, which will provide a list of workarounds for use until the 'final' conclusion is reached.

OpenBSD: OpenBSD does not directly support hyperthreading at this time, therefore no patch is available. Affected users may disable hyperthreading in their system BIOS. We will revisit this issue when hyperthreading support is improved.

SCO: This affects OpenServer 5.0.7 if an update pack is applied and SMP is installed; if also affects UnixWare 7.1.4 and 7.1.3 with hyperthreading enabled, but hyperthreading is disabled in UnixWare by default. This is covered by advisory SCOSA-2005.24.