CPU traps and pitfalls

D. J. Bernstein

University of Illinois at Chicago
Situation: You’ve written software for a cipher/hash/etc. It computes correct output.

But it has speed problems: it’s awfully slow.

It also has security problems: it leaks secret data through side channels.

This talk: some CPU behaviors that cause slowdowns, leaks; lessons for the implementor; lessons for the designer.
Load throughput

Oversimplification: CPU has many megabytes of “RAM,” instantaneously accessible.

Reality: A few “registers” are instantaneously accessible. Must copy (“load”) data from RAM into registers before performing computations.

Imagine 256 bytes of registers. Exact number depends on CPU.
\( n \) loads take
\[ \geq n \text{ cycles on a Pentium III}, \]
\[ \geq n \text{ cycles on a Pentium 4}, \]
\[ \geq n \text{ cycles on an UltraSPARC}, \]
\[ \geq \frac{n}{2} \text{ cycles on an Athlon}, \]
\[ \geq n \text{ cycles on a Pentium M}, \]
\[ \geq n \text{ cycles on a Core 2}, \]

etc.

“Load throughput: 1 load/cycle but 2 loads/cycle on Athlon.”

“Cycles” = microseconds/MHz; e.g., on a 1600MHz Pentium M, 1600 cycles every microsecond.
Typical line of AES code:
y0 = T0[x0&255]

T0 is a 1024-byte array with 4-byte elements
T0[0], T0[1], ..., T0[255].

Array is stored in RAM:
it won’t fit into registers!
T0[...] is a load.

Each AES round involves 16 of these “S-box loads” and 4 “expanded-key loads”:
≥ 20 cycles on typical CPU.
10 rounds: ≥ 200 cycles.
When loads are a bottleneck, try to eliminate loads by keeping data in registers; recomputing instead of loading; merging adjacent loads; etc.

e.g. In AES, can replace 44 expanded-key loads with 14 loads, 30 xors. “Partially expanded keys.”

For subsequent AES blocks, can eliminate these 14 loads at expense of 14 regs, if CPU has 14 spare regs.
Skipping operations

Oversimplification: CPU takes time $n$ for an $n$-iteration loop.

Reality: CPU stops early if asked.

1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at first difference. Attackers watch comparison time, deduce position of difference. A few hundred tries reveal secret password.
1996: Kocher points out timing attacks on cryptographic key bits. Example: key-dependent branch in modular reduction, performing big-integer subtraction for some inputs and not others, leaking key via timings.

1999: Koeune Quisquater publish fast timing attack on a “careless implementation” of AES that used input-dependent branches.
Koeune-Quisquater attack target:

```
byte c = S(b);
if (c<128) return c+c;
return (c+c)^283;
```

Faster if \( c < 128 \).

Fix, eliminating leak:
replace branch by arithmetic.

```
byte c = S(b);
X = c>>(7);
X |= (X<<1);
X |= (X<<3);
return (c<<1)^X;
```
2007: IPsec software uses memcmp to check authenticators! TENEX disaster redivivus.

How memcmp works:

   for (i = 0; i < n; ++i)
       if (x[i] != y[i])
           return 0;

   return 1;

Fix, eliminating leak:

   diff = 0;
   for (i = 0; i < n; ++i)
       diff |= x[i] ^ y[i];
   return !diff;
Arithmetic latency

Oversimplification:
Pentium III performs
2 arithmetic ops every cycle;
Pentium 4 performs
4 arithmetic ops every cycle;
UltraSPARC performs
2 arithmetic ops every cycle;
Athlon performs
3 arithmetic ops every cycle;
Pentium M performs
2 arithmetic ops every cycle;
Core 2 performs
3 arithmetic ops every cycle; etc.
Typical lines of MD5 code:
\[
w += (z \^\left(x \& (y \^ z)\right)) + \text{block}[i] + c;
\]
\[
w \llll s; w += x
\]
8 arithmetic operations.
4 cycles on Pentium M?
2.66666... cycles on Core 2?

Reality: 6 cycles!
Reason: Result of arithmetic isn’t usable until next cycle.
“Arithmetic latency: 1 cycle.”

\[
\^ \rightarrow \& \rightarrow \^ \rightarrow + \rightarrow \llll \rightarrow +.
\]

MD5 takes \( > 5 \) cycles/byte.
Each Salsa20 round involves 16 adds of 32-bit words, 16 rotations, 16 xors.

One Salsa20 implementation works with 4-word vectors using “XMM” instructions.

Salsa20 state is 4 vectors. Each Salsa20 round involves 4 vector adds, 8 vector copies, 8 vector shifts, 8 vector xors, 3 vector shuffles.
Core 2: 3 vector ops/cycle, but > 20-cycle latency for this sequence of ops.

Better software by Wei Dai: Take advantage of counter mode; handle 4 blocks in parallel. No more latency concerns. Throughput is bottleneck.

< 14 cycles/round, despite extra load time.
4.9 cycles/byte for Salsa20/20.
3.1 cycles/byte for Salsa20/12.
2.3 cycles/byte for Salsa20/8.
Branch-prediction channels

Oversimplification:
If $x()$ and $y()$ have same speed then

```python
    if (secretbit)
        outbit = x();
    else
        outbit = y();
```

doesn’t leak secretbit.

Reality: CPU’s branch predictor remembers secretbit—and doesn’t hide it from other programs on computer.
Complete fix:

\[
\text{outbit} = (x() \& \text{secretbit}) + (y() \& (1-\text{secretbit}));
\]

Takes twice as much time. Look for ways to merge work between \(x()\) and \(y()\).

2005: Bernstein “Curve25519” elliptic-curve Diffie-Hellman “avoids all input-dependent branches, all input-dependent array indices”; overhead is “about 6% of the total” time.
Load latency

Typical memory hierarchy:

3-cycle latency to load from 32768-byte “L1 cache.”

10-cycle latency to load from 2097152-byte “L2 cache.”

250-cycle latency to load from 1073741824-byte “DRAM.”

The numbers here depend on CPU, motherboard, etc.
Pentium III, Pentium 4, Athlon, Pentium M, Core 2 (but not UltraSPARC) are “out-of-order” CPUs: they’ll look ahead in program to find load instructions (and various other instructions) that can be performed now.

Latency is still a problem: many loads aren’t ready yet; lookahead distance is limited; loads are performed greedily.
Warning: Most benchmarks load all data into cache, hiding the cache-miss latency.

In (e.g.) busy network server, data is often out of cache, and cache misses are critical.

Try to reduce latency by compressing data in RAM; reusing recently used data; prefetching data; etc.

e.g. 8192-byte AES tables can be compressed to 2048 bytes.
Load-address channels

Time for array lookup depends on array index, leaking information to attacker.

Variability mentioned by 1996 Kocher, 2000 Kelsey Schneier Wagner Hall ("We believe attacks based on cache hit ratio in large S-box ciphers like Blowfish, CAST and Khufu are possible"), 2003 Ferguson Schneier.
In AES, $y_0 = T_0[x_0 \& 255]$ time depends on $x_0 \& 255$, a byte of plaintext $\oplus$ key.

Attacker can force selected table entries out of L2 cache, observe encryption time. Each cache miss creates timing signal, easily visible despite noise from other AES cache misses, other software, etc. Repeat for many plaintexts, easily deduce key.
Partial fix:
Eliminate all cache misses.

Put AES software into operating-system kernel.
Disable interrupts.
Disable hyperthreading etc.
Read all S-boxes into cache.
Wait for reads to complete.
Encrypt some blocks of data.

The bad news: Stopping cache misses isn’t enough.
There are timing leaks in cache *hits*. 
Load-after-store conflicts:

On (e.g.) Pentium III, load from L1 cache is slightly slower if it involves same cache line modulo 4096 as a recent store.

This timing variation happens even if all loads are from L1 cache!
Cache-bank throughput limits:

On (e.g.) Athlon, can perform two loads from L1 cache every cycle.

Exception: Second load waits for a cycle if loads are from same cache “bank.”

Time for cache hit again depends on array index.

No guarantee that these are the only effects.
Complete fix: Never use secret data as load address.

Every cipher can be simulated with simple arithmetic: e.g., add, xor, rotation.

To compute $T[\text{secret}]$: load $T[0]$, $T[1]$, $T[2]$, ... and do appropriate arithmetic.

Takes time to load all of $T$. Look for ways to reduce time: e.g., bitslicing.
Design thoughts

Modern CPUs offer considerable parallelism: can compute several independent adds, xors, etc. each cycle.

We can design ciphers with many parallel adds, xors, etc., allowing implementors to exploit CPU capabilities.

Do other operations achieve the same level of security at higher speed?
An AES S-box lookup mangles its input more thoroughly than an addition or xor.

But it is slower and has a much smaller input.

Challenge: Can anyone build an unbroken stream cipher using AES S-box lookups as fast as Salsa20/12? as fast as Salsa20/8? as fast as Salsa20/8, without side-channel leaks?
Advertisement

“SPEED: Software Performance Enhancement for Encryption and Decryption”

A workshop on software speeds for secret-key cryptography and public-key cryptography.

Amsterdam, June 11–12, 2007

http://

www.hyperelliptic.org/SPEED