The impact of side-channel attacks on the design of cryptosystems

D. J. Bernstein
University of Illinois at Chicago
The standard model of senders, receivers, attackers, etc.: 

1. Parties perform computation, send messages to other parties. 
2. Parties receive all messages, perform more computation, send messages to other parties. 
3. Etc. 

Party’s local computation is not visible to other parties except via message contents. Each party’s behavior is completely determined by message contents and local data.
e.g. Diffie-Hellman using public elliptic-curve point $P$:

1. Alice generates random $a$, computes $aP$, sends $aP$.
Bob generates random $b$, computes $bP$, sends $bP$.

2. Alice and attacker receive $bP$. Bob and attacker receive $aP$. Alice computes $a(bP) = abP$. Bob computes $b(aP) = abP$.

Attacker tries to compute the secret $abP$ from his own data and $aP$ and $bP$. Standard model: attacker has no other inputs.
Traditional cryptography: Design and implement cryptographic systems in the standard model.

Traditional cryptanalysis: Analyze security of cryptographic systems in the standard model.

Build confidence that contents of messages don’t reveal keys, don’t allow forgeries, etc. Also aim for high speed.
In the real world, extra information is visible through “side channels” outside the standard model.

e.g. *When* did program finish computing outgoing message? Visible to network sniffer.

e.g. When did program finish processing incoming message? Visible to the next program run by the same CPU.

e.g. How much power is program using right now? Visible to power source.
Often this information depends on cryptographic keys (or similarly critical secrets).

Can attacker now compute keys?

In many cases: Yes.

In many cases: Unclear.
Can be difficult to analyze.
Growing research area.

Have built confidence in standard-model security, but do we have confidence in real-world security? No!
Strategy to regain confidence:
Ensure that all information visible on side channels is independent of cryptographic keys etc.

Independence guarantees that real-world attacks are as hard as standard-model attacks.

“Side-channel-immune cryptographic implementation.”

More difficult when there are more side channels. Extreme case: Hard to keep secrets in Pay-TV smartcards given to attackers.
This talk focuses on *software* side channels: load timing, branch timing, etc.

Happy fact: Every cipher, every signature system, etc. can be implemented to keep keys safely away from all known software side channels.

Replace “all known” with “all”? I think so, but can’t verify.

Intel and AMD are hiding security-critical information. “Proprietary speed data.” Idiots. But let’s assume “all.”
Unhappy fact:
For many cryptographic systems, side-channel-immune software is surprisingly slow.

Happy fact:
Some cryptographic systems achieve very high software speeds despite side-channel immunity.

“Side-channel-immune cryptographic design”:
In designing cryptosystems, aim for security and high speed of side-channel-immune software, not standard-model software.
Case study: string comparison

1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at first difference.

AAAAAA vs. SECRET: stop at 1.
SAAAAAA vs. SECRET: stop at 2.
SEAAAAA vs. SECRET: stop at 3.

Attackers watch comparison time, deduce position of difference. A few hundred tries reveal secret password.
Objection: “Timings are noisy!”

Answer #1: Even if noise stops simplest attack, does it stop all attacks? Need side-channel immunity to regain confidence.
Objection: “Timings are noisy!”

Answer #1: Even if noise stops simplest attack, does it stop all attacks? Need side-channel immunity to regain confidence.

Answer #2: Eliminate noise using statistics of many timings.
Objection: “Timings are noisy!”

Answer #1: Even if noise stops simplest attack, does it stop all attacks? Need side-channel immunity to regain confidence.

Answer #2: Eliminate noise using statistics of many timings.

Answer #3, what the 1970s attackers actually did: Increase timing signal by crossing page boundary, inducing page faults.
2007: IPsec software uses `memcmp` to check authenticators!

TENEX disaster redivivus.

Exercise: Forge IPsec packets.

Typical `memcmp`-style comparison:
```c
for (i = 0; i < 16; ++i)
    if (x[i] != y[i])
        return 0;
return 1;
```

Fix, side-channel immune:
```c
diff = 0;
for (i = 0; i < 16; ++i)
    diff |= x[i] ^ y[i];
return !diff;
```
Case study: Montgomery ladder

Montgomery’s computation of \(((2n + b)P, (2n + b + 1)P)\) from \((P, nP, (n + 1)P, b)\) inside fast scalar multiplication:

```plaintext
if (secretbit == 1) {
    newnp = g(p, np, n1p);
    newn1p = f(n1p);
} else {
    newn1p = g(p, np, n1p);
    newnp = f(np);
}
```
Naive view: “This doesn’t leak secretbit. It takes the same time whether secretbit is 0 or 1.”

Reality: Branch time is an extremely complicated function of secretbit, previous bits, branch bits elsewhere in program, branch bits in other processes, et al.

secretbit influences time for this computation, time for other computations, and time in other processes.
The obvious way to achieve branch-side-channel immunity: Never branch on secret bits.

This is easily doable, and for Montgomery ladder it adds very little overhead.

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.
How can we do

```c
if (secretbit == 1) {
    foutput = f(n1p);
} else {
    foutput = f(np);
}
```

without secret branches
(or secret indices)? Answer:

```c
finput =
    ((secretbit-1)&np )
|(( -secretbit)&n1p);
foutput = f(finput);
```
Full Montgomery, branchless:

\[ g_{output} = g(p,np,n1p); \]
\[ f_{input} = ((\text{secretbit}-1)\&np) \]
\[ \mid((\neg\text{secretbit})\&n1p); \]
\[ f_{output} = f(f_{input}); \]
\[ \text{newnp} = g_{output} ^\sim \]
\[ (((f_{output} ^\sim g_{output}) \]
\[ \& (\text{secretbit}-1)); \]
\[ \text{newn1p} = \text{newnp} ^\sim \]
\[ (f_{output} ^\sim g_{output}); \]

Faster alternative:

Merge flips across loops.
Case study: modular arithmetic

General-purpose libraries for high-precision arithmetic often have data-dependent “skip leading 0” loops.

Integers modulo $n$ are occasionally shorter than $n$. Length leaked by timings.

Fix: Always process integers to the same length as $n$. Eliminating leading-0 tests normally saves time.
More branches in arithmetic?

Kocher’s original target for timing attacks:

```c
    a += b;
    if (a >= n) a -= n;
```

Fix, side-channel immune:

```c
    a += b;
    c = a - n;
    secretbit = signbit(c);
    c ^= a;
    a ^= ((secretbit-1)&c);
```

Often acceptable alternative:

```c
    a += b; /* that’s it! */
```
Exercise: Starting from extended-Euclid algorithm or simpler binary variant, build side-channel-immune modular inversion. Replace branches by arithmetic, use known loop-count limit.

Much simpler, often fast enough: compute \( a^{−1} \mod p \) as \( a^{p−2} \mod p \).

One modular inversion, after elliptic-curve scalar mult using inversion-free coordinates, is not a big bottleneck.
Case study: AES

AES code: $y_0 = T_0 [x_0 \& 255]$

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 T0 etc. 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 reason to think that these are the only effects.
The obvious way to achieve address-side-channel immunity: Never use secret addresses.

To compute \( T0[\text{secret}] \):
Load \( T0[0], T0[1], T0[2], \ldots \) and do appropriate arithmetic.

Takes time to load all of \( T0 \).
Parallel lookups in one table can save some time (asymptotic cost \( O(n \lg n) \) for \( n \) lookups in \( n \)-entry table), but still quite expensive.

Side-channel-immune AES is disappointingly slow.
Do cipher designers need secret table indices?

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

We can design ciphers with many parallel arithmetic operations, allowing implementors to exploit CPU capabilities.

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

But it is slower and has a much smaller input.

Side-channel-immune Salsa20 software is faster than side-channel-vulnerable AES software despite having a much larger security margin.
What’s safe?

No known side channels in some operations on current CPUs:

Loads from constant addresses.
Stores to constant addresses.

Constant-distance shifts.
Constant-distance rotations.

Logical operations: xor, or, etc.

Can build all computations from these operations, just like building a circuit from individual gates.
And some more arithmetic:
Additions, subtractions.
No common CPU is believed to abort carry chains early.

Integer multiplication, except that large inputs slow down some CPUs—CPU implicitly does, e.g.,
if (input < 65536) . . .

Particularly useful for RSA etc.: floating-point multiplication in normal exponent ranges, except that an input of 0 slows down a few CPUs.
Summary of the impact

Maintaining confidence in security of cryptographic software, in a world of side-channel attacks, takes some implementation effort.

Can avoid big slowdowns for typical public-key systems. Small effect on designer.

Much larger slowdown for typical secret-key ciphers, except parallel-arithmetic ciphers. Much larger effect on designer.