23 years of software side channel attacks

Colin Percival
Tarsnap Backup Inc.
cperciva@tarsnap.com

September 22, 2019

Who am I?

- FreeBSD developer since 2004.
  - Author of FreeBSD Update and Portsnap.
  - Maintainer of the FreeBSD/EC2 platform.
- FreeBSD Security Officer 2005–2012.
- Occasional cryptographer.
  - Best known for a side channel attack on shared L1 caches (2005) and scrypt (2009).
- Author of Tarsnap.
  - Online backups for the truly paranoid.
  - This is my day job, and it's paying for me to be here.

Software side channel attacks

- Black boxes tend to leak information in many ways.
  - Electromagnetic radiation.
  - Power consumption.
  - Sound.
  - Time before the output is produced.
  - Internal state which can be retrieved later.
- If you leak information deliberately, it's a *covert channel*.
- If you leak information accidentally, it's a *side channel*.
- *Software* side channels are those which can be exploited without needing special hardware or physical access.
- If you can obtain secrets via a side channel, you have a *side channel attack*.
  - Typically the secrets we're concerned with are cryptographic.

Early modern cryptography

- 1977: Rivest, Shamir, and Adleman publish RSA.
  - Mostly a mathematical curiosity given computers of the era.
- June 1991: Phil Zimmermann releases PGP.
  - RSA is suddenly available to the general public!
  - The US Government is NOT happy.
  - Very hard to target with side channel attacks due to offline usage.
- February 1995: SSL 2.0 is released.
  - RSA is now being used *interactively*.
  - Web servers are connected to the internet and respond promptly to incoming packets.
  - This creates an opening for timing attacks.

Kocher 1996

- "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems"
  - Straightforward implementations of these used non-constant-time modular multiplication routines.
  - If you can predict which multiplications will complete faster than others, you can time operations on chosen inputs to gain information about the private key being used.
  - The private key can be extracted one or two bits at a time based on which inputs yield the fastest operations.
  - Requires timing $\approx 10^3$ RSA operations.
- At the time, one RSA private key operation typically took 400 ms, while a "fast" modular multiplication was ≈ 20 μs faster than a "slow" multiplication.
- IEEE 802.3u "Fast Ethernet" was introduced in 1995; a 1500 byte packet took 120 μs to transmit.

Boneh / Brumley 2003

- "Remote timing attacks are practical".
  - Perform a binary search for one of the factors of an RSA modulus, relying on a timing channel in Montgomery reduction with the Chinese Remainder Theorem.
  - Rather than measuring how long *one* cryptographic operation takes, measure how long *many* cryptographic operations take.
  - Averaging the times taken by N operations increases the signal:noise ratio by a factor of $\sqrt{N}$.
  - Rather than timing $\approx 10^3$ RSA operations, we now time a total of $\approx 2 \times 10^6$ operations.
  - "a typical attack takes approximately 2 hours".
- That attack which was "purely theoretical"? It's real. Fix your side channels!

Defense: Blinding

- The Kocher and Boneh / Brumley attacks make use of *chosen inputs* in order to find the secret exponent or prime.
- Rather than calculating

$$x^d \mod N$$

pick a random value r and calculate

$$(xr^e)^d r^{-1} \mod N$$

- Since $e \ll d$, calculating r^e and r^{-1} is fast compared to calculating x^d.
- As long as a new random value r is chosen for each exponentiation, the inputs are unpredictable and cannot reveal information to the attacker.

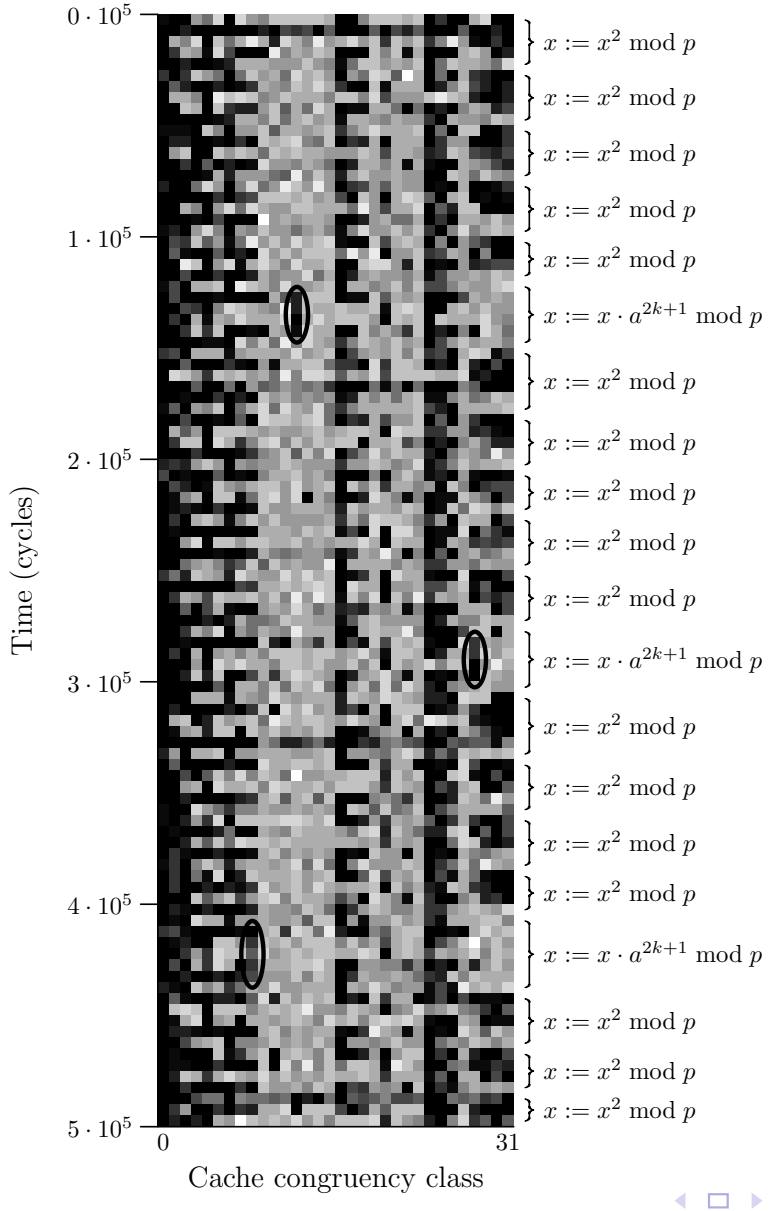
Bernstein 2004

- "Cache-timing attacks on AES".
- Straightforward implementations on AES perform "S-box" table lookups.
- Table lookups are performed using the bytes in $key \oplus input$ as indices.
- If certain table offsets take longer to access than others, you can try many different inputs and find the key which correlates best with the observed timings.
- Cache occupancy, load/store conflicts, cache-bank conflicts...
- Attack typically requires timing $\approx 10^9$ random inputs to AES.
- Defense: Use hardware AES circuits rather than software AES whenever possible!

Percival 2005

- "Cache missing for fun and profit".
- Attack on Symmetric Multi-Threading (e.g., Intel Hyperthreading):
  1. Pull data into the L1 cache.
  2. A moment later, measure how long it takes to re-access the same data.
  3. Time taken for memory access reveals whether it was evicted from the L1 cache by the other hyperthread.
- We never measure how long a cryptographic operation takes — this is not a timing attack!
- New family of attacks: Microarchitectural side channels.
  - Microarchitectural side channels can be much higher bandwidth since they can reveal information *while* an operation is being performed.
  - An RSA private key can be stolen by observing a single operation.

Percival 2005



Osvik / Shamir / Tromer 2005

- Uses the same approach of timing data re-accesses to determine the "cache footprint" of an AES operation.
- As before, a hyperthread can monitor an operation sharing the L1 cache.
- Also demonstrated stealing AES keys used by Linux `dm-crypt` after kernel returns to userland — having *simultaneous* access to the cache is not necessary.
- Attack takes between 10^2 and 10^6 AES operations depending on the CPU and method of attack.

Defense: Oblivious code + data accesses

- No secret-dependent conditional branches (`if`, `?:`, or `for`/`while` conditions).
- No secret-dependent array indexing.
- This may require extra operations; e.g., replacing

  ```
  x = condition ? foo() : bar();
  ```

  with

  ```
  x = foo() * condition + bar() * (1 - condition);
  ```

  and executing "both sides" of the conditional.
- Side benefit: In addition to preventing microarchitectural side channels, this protects against timing side channels.

More attacks followed...

- Over the years more attacks targetting shared CPU resources piled up.
  - Intel, 2005: L2 cache (unpublished).
  - Aciiçmez / Koç / Seifert, 2006: CPU branch predictors.
  - Aciiçmez, 2007: L1 instruction cache.
  - Liu / Yarom / Ge / Heiser / Lee, 2015: L3 cache.
  - Gras / Razavi / Bos / Giuffrida, 2018: TLB.
  - Aldaya / Brumley / Hassan / Garca / Tuveri, 2018: CPU execution ports.
  - ... probably many more that I've forgotten.
- Code which follows guidelines from 2005 is also immune to all of these attacks.

CPU architecture

- CPU Pipelining has been used since the IBM Stretch (1961).
  - Improves performance by allowing the CPU to start processing the next instruction before it finishes the previous one.
  - Classic RISC pipeline: Instruction fetch, Instruction decode, Execute, Memory access, Commit.
  - Modern x86 pipelines typically have ≈ 15 stages.
- Out-of-order execution became common starting with IBM POWER1 (1990).
  - The start (instruction fetch/decode) and end (commit) of the pipeline remains in order.
  - Particularly important on x86 due to small number of registers.
- The instructions must flow!

Speculative execution

- All modern CPUs start handling instruction $\#N + 1$ before instruction $\#N$ has completed.
  - Unless you insert a *serializing* instruction.
- Pipeline flushes can happen for many reasons.
  - Branch misprediction.
  - Indirect branch *target* misprediction.
  - Exceptions.
  - Data hazards.
  - Self-modifying code.
- When a pipeline flush occurs, the speculatively executed instructions are not committed — the architectural state of the CPU is unchanged.
- Unfortunately the *micro*architectural state might be changed.

Meltdown

- Meltdown attack:
  1. Try to read from an unreadable address.
  2. Use the value read as an index for an array access.
  3. Intel handles traps at time of instruction *commit*.
  4. Pipeline is flushed and memory access in step 2 "never happened".
  5. ... but you can measure its effects on the cache anyway.
- Rogue System Register Read: Same as Meltdown except using RDMSR.
- Lazy FPU state switching attack: Same as Meltdown except reading the SSE registers.
- SWAPGS attack is also similar.
- Delayed exception handling makes it possible to speculate through faulting instructions.
- AMD and other non-Intel CPUs are (mostly?) not affected since they identify faults earlier and do not speculate through them.

More CPU design issues

- Speculative Store Bypass:
  1. Affects ≈all modern CPUs.
  2. Write to a memory location.
  3. Read from that same memory location.
  4. Do something using the value you read.
  5. If the CPU realizes "too late" that it's the same memory location, the pipeline will be flushed.
  6. But you can measure side effects from the old value in memory anyway.
- Microarchitectural buffer sampling:
  1. Several of these vulnerabilities on Intel CPUs.
  2. Data is forwarded from internal temporary buffers to upcoming instructions.
  3. Processor realizes "too late" that the data should not have been forwarded, and the pipeline is flushed after the data has been leaked.
  4. In some cases leak occurs between hyperthreads.

Spectre

- Bounds check bypass: CPU mispredicts branch; data is speculatively read (and used) from beyond the end of a buffer.
- Branch target injection: CPU mispredicts the target of an indirect branch; code (of your choice!) is speculatively executed.
- General issue with speculative execution: If the processor mis-speculates, you might speculatively run code you didn't expect to run.
  - Affects all modern CPUs.
  - Branches mispredictions happen even in good times.
  - Will not bypass OS-level privilege boundaries — sandboxes are your friend!
- Many possible exploit paths; e.g., switch(opcode) in p-code machines might mispredict with dangerous results.

Many ways of leaking state!

- Attacks to date have leaked speculated information by leaving footprints in the L1 data cache.
- You can also leak information via the L1 instruction cache, via branch predictors, via the TLB, via CPU execution port contention...
- You can even leak information without executing instructions:
  - Instruction pre-decode loads data from the L1 code cache, leaving a measurable footprint behind.
  - Instruction pre-decode performance varies depending on the bytes being decoded.
  - How much code gets loaded into the L1 code cache before the CPU pipeline is flushed reveals information about the code...

Questions?