Last-level cache side channel attacks are practical

Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser and Ruby B. Lee
Motivation
Cross-VM side channel attacks

- L1 attacks require being on the same core
  - Exploit hyperthreading or scheduler weaknesses
- Existing last-level cache attacks like Flush+Reload require page deduplication
- Our attack does not have these restrictions
  - No synchronisation with victim needed
Cache Structure

- Stores fixed-size *lines*
- Arranged as multiple *sets*, each consisting of multiple *ways*.
- Each memory line maps to a single cache set
  - Can be cached in any of the ways in the set
Prime+Probe [OST05, Per05]

- Attacker chooses a cache-sized memory buffer
- Attacker accesses all the lines in the buffer, filling the cache with its data
- Victim executes, evicting some of the attackers lines from the cache
- Attacker measures the time to access the buffer
  - Accesses to cached lines is faster than to evicted lines
Challenges to last-level cache attacks

• Difficulty in finding memory lines that map to a given cache set
  – Virtual memory
    • Use large pages
  – LLC slices
    • Empirically determine the mapping to cache slices

• Large cache size and longer cache access times mean LLC Prime+Probe is very slow
  – Only probe selected cache sets at a time

• Visibility of the victim memory access at the LLC
  – Intel inclusive cache takes care of the issue
Virtual vs. Physical addresses

• In L1, the cache stride is the same as the page size (4KiB)
  – The cache set is completely determined by the page offset
In larger caches, the stride is bigger. Pages only contain lines of some of the cache sets.

Parts of the mapping of memory to cache sets is masked by the virtual address.
Addressing uncertainty

The attacker cannot guarantee an even cover of the cache.
A feature of the MMU. Large pages (2MiB or bigger) reduce the overhead of address translation.
Mapping the LLC slices

- The last-level cache is divided into *slices*
- One slice per core
- The slice for a memory line is chosen using an unpublished hash function
- We need to find which memory lines map to each cache set, in each slice
Finding memory lines that map to a cache set without requiring the Intel slice mapping

- Create a pool of memory lines that exactly covers a given cache set in all slices
  - Start from an empty pool
  - Iteratively add lines to the pool as long as there is no self-eviction
  - Self-eviction is detected by priming a potential new member, accessing the current pool and timing another access to the potential new member

- Partition the pool to slices
Finding memory lines that map to a cache set without requiring the Intel slice mapping

- Create a pool of memory lines that exactly covers a given cache set in all slices
  - Start from an empty pool
  - Iteratively add lines to the pool as long as there is no self-eviction
  - Self-eviction is detected by priming a potential new member, accessing the current pool and timing another access to the new member

- Partition the pool to slices
Virtual address hides the victim cache set

• Past solution – generate a victim’s spatial footprint
  – Represent cache state as a vector
  – Classify vector to find state
  – Filter the classification results
  – Example [ABG10] uses vector quantisation followed by a hidden Markov model

• Does not work for the LLC
Slow LLC Prime+Probe times

- **L1 (32KB) probe:**
  - 64 sets * 8 ways * 4 cycles = 2,048 cycles

- **Small last-level cache (6MB):**
  - 8,192 sets * 12 ways * ~30 cycles = ~3,000,000 cycles

- **We cannot probe the entire LLC in a reasonable time, but probing one cache set is fast**
  - We show a LLC covert channel bandwidth of 1.2 Mbps
    - 6 times faster than previous LLC-based channel

- **Our solution:**
  - Probe one or a few cache sets at a time
  - Look for temporal patterns rather than spatial footprints
Attack Strategy

• Find an eviction set for every cache set in every slice

• Search for temporal pattern of victim operations:
  – For each cache set in each cache slice
  – Example: search for the SQUARE code to attack the Square and Multiply exponentiation in GnuPG 1.4.13
ElGamal Encryption

• The ElGamal encryption is a public key cryptographic scheme
• The main operation is modular exponentiation, i.e. calculating

\[ b^e \mod n \]

• The exponent \( e \) used for decryption is the private key
Square-and-Multiply Exponentiation (GnuPG 1.4.13)

\[
x \leftarrow 1 \\
\text{for } i \leftarrow |e|-1 \text{ downto } 0 \text{ do} \\
\quad x \leftarrow x^2 \mod n \\
\quad \text{if } (e_i = 1) \text{ then} \\
\quad \quad x \leftarrow xb \mod n \\
\quad \text{endif} \\
\text{done} \\
\text{return } x
\]

Example:
\[
11^5 \mod 100 = 161,051 \mod 100 = 51
\]

The secret exponent is encoded in the spacing between squaring operations !!!
Attack

• Find eviction sets for cache sets
• Divide time to slots of 5,000 cycles
• For each cache set:
  – Prime set at the beginning of a time slot
  – Wait to the end of the time slot
  – Probe the cache set and record data
  – Repeat for 7,000 time slots
• Scan recorded data, looking for the expected use of the squaring code.
• Parse data
Searching for the Square code

- Expected pattern: bursts of cache activity separated by shorter or longer intervals for the modular reduce and multiplication operations.
Reliability

Number of errors

Desktop
Server
Sliding Window Exponentiation (GnuPG 1.4.18)

Divide exponent into windows

- Zero windows of any length
- Non-zero windows
  - Up to \( w \) bits long
  - Start and end with 1

Precompute \( b^1, b^3, \ldots, b^{2^{w-1}} \)

\[
e = \begin{array}{cccccccccccccccc}
1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
\end{array}
\]

\[
r = 1 \\
\text{for } i = n \text{ downto } 1 \text{ do} \\
\quad \text{for } j = 1 \text{ to } L(w_i) \text{ do} \\
\quad \quad r = \text{SQR}(r) \mod m \\
\quad \text{end} \\
\text{MUL}(r;r) \\
\text{if } w_i \neq 0 \text{ then} \\
\quad r = \text{MUL}(r,b^{w_i}) \mod m \\
\text{end} \\
\text{return } r
The Sliding Window exponentiation

• Differences from Square and Multiply:
  – Square is implemented using the multiply code
  – Significantly less “real” multiplications
  – 8 possible multipliers used in the multiplications

• Challenges:
  – The multipliers table is dynamically allocated
  – Multipliers use is sparse and irregular
  – Cannot distinguish squares from multiplications
Attacking the Sliding Window exponentiation

- Find the multiplication code.
  - We know how to do that

- We now know when exponentiation starts and the timing of each multiplication
Finding the multipliers used in multiplications

- Multiplier access pattern repeats across exponentiations
  - A temporal access pattern, while unknown, does exist
- We probe the cache set of the multiplication code and an arbitrary cache set
  - Cluster activity during multiplications sequence to find repeating patterns
Summary of results

• Cross-core, cross-VM asynchronous attack on square-and-multiply exponentiation
  – Under one minute on-line attack
    • Compared to 6 hours [ZJR+12]
    • Slower than Flush+Reload, but does not require deduplication

• Cross-core, cross-VM asynchronous attack on sliding-window exponentiation achieving full key recovery
  – 23 minutes
    • 12 minutes on-line
    • 1 minute automated off-line processing
    • 10 minutes manual processing
  – Not feasible with Flush+Reload
Conclusions

• A practical side-channel attack on the last-level cache
  – asynchronous, cross-core and cross-VM
• A novel attack strategy based on finding temporal patterns of victim activity
• Use large pages to find the mapping of memory lines to cache sets
• A new algorithm for finding eviction sets for every cache set in every cache slice
  – without knowing the memory to slice mapping
• A fast LLC covert channel of 1.2 Mbps
• Identify and help fix a vulnerability in the then latest version of a popular crypto library
  – CVE-2015-0837