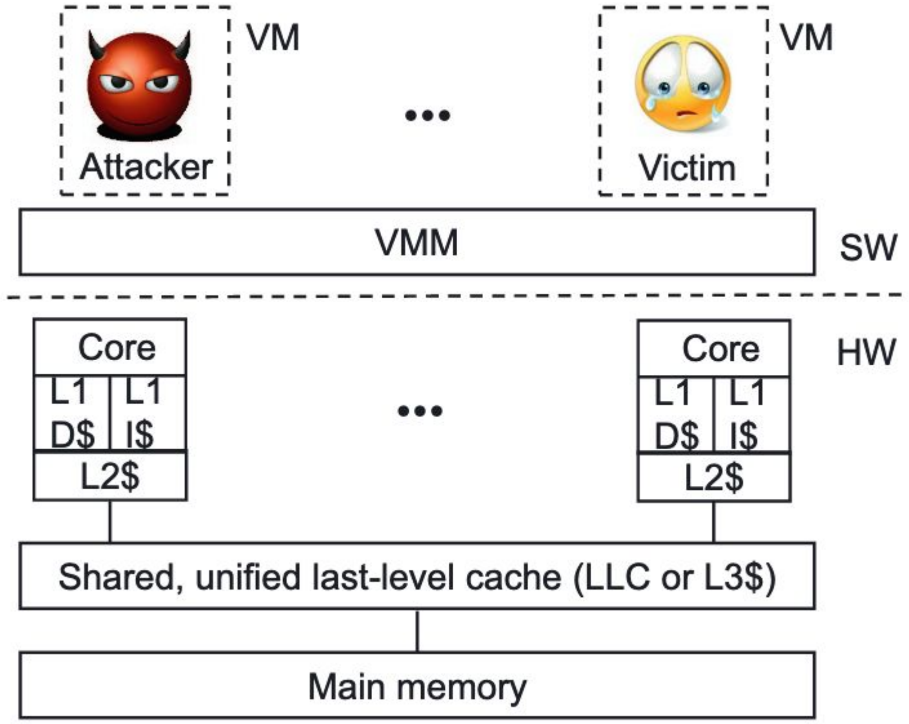
LLC Side-Channel Attacks are Practical

A Harder Version of Lab 2

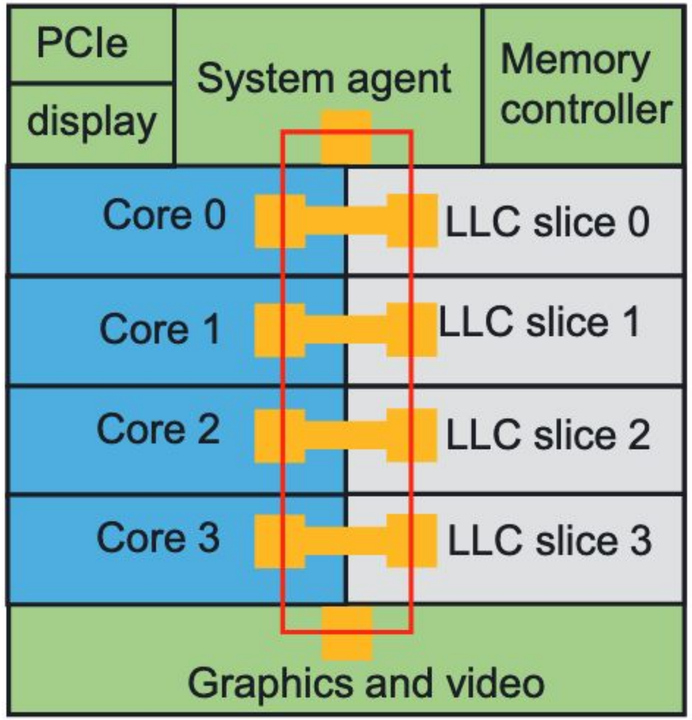
Scenario

- You’re a VM running on a cloud server.
- You want to figure out encryption keys of other VMs on the same machine?
  - We know from lab 2 that you can use the L1/L2 caches but this requires the VMs to be running on the same core
  - Are there other micro-architectural features that you can use that would let us avoid these limitations?
- How would you do it?

Introducing the Last Level Cache (LLC)



The architecture we're considering at a high level



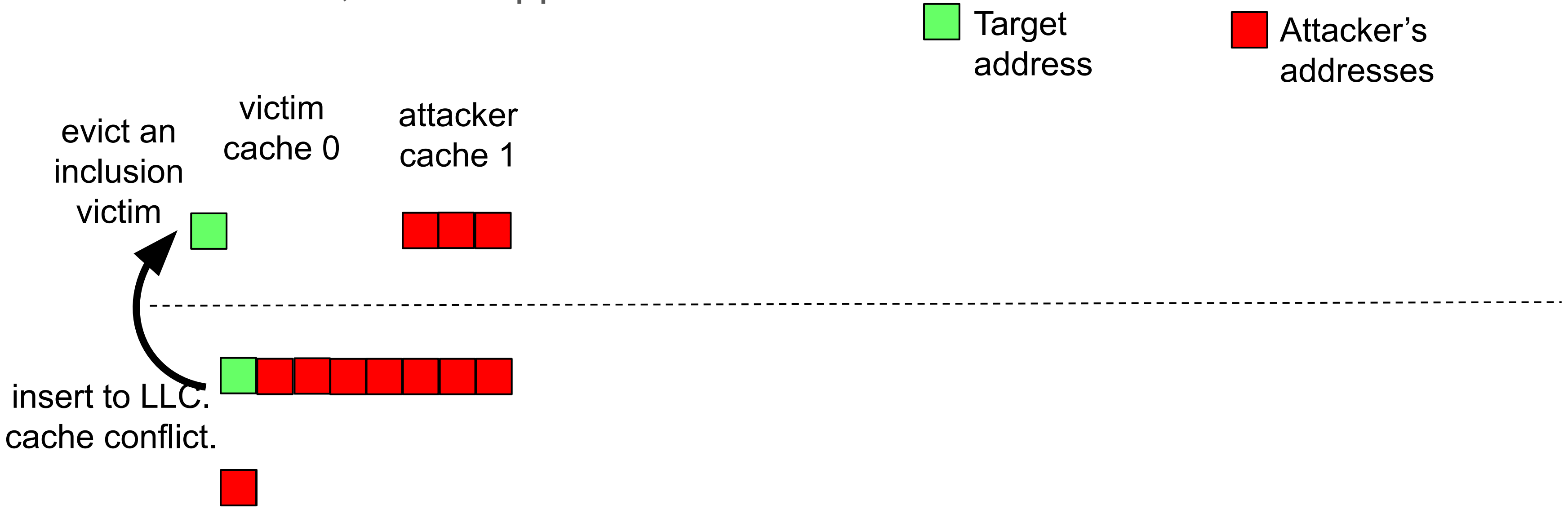
The LLC using what is called a ring-bus architecture

Using the LLC vs. L1/L2

1. Unable to see into one core's memory accesses from another core via LLC
   a. The attacker is unable to see the victim's memory accesses when they only affect L1/L2
2. It takes significantly longer time to probe the LLC
3. We want be able to identify the cache sets corresponding to security-critical accesses by the victim without probing the whole LLC
4. Need to construct an eviction set that can occupy exactly one cache set in the LLC, but we don't know the address mappings
5. Is our probing resolution fine-grained enough to be able to detect the victim's memory accesses?

Evicting Cache Lines

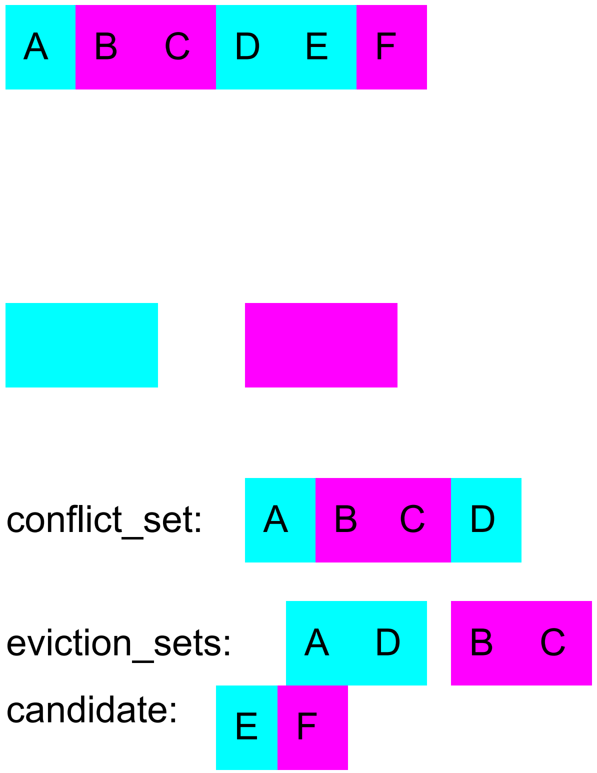
- Cache is inclusive
    - i.e. each line in L1 and L2 caches must be somewhere in LLC
- If a cache line is in the victim's L1 & L2 caches and we then evict it from the LLC, what happens?

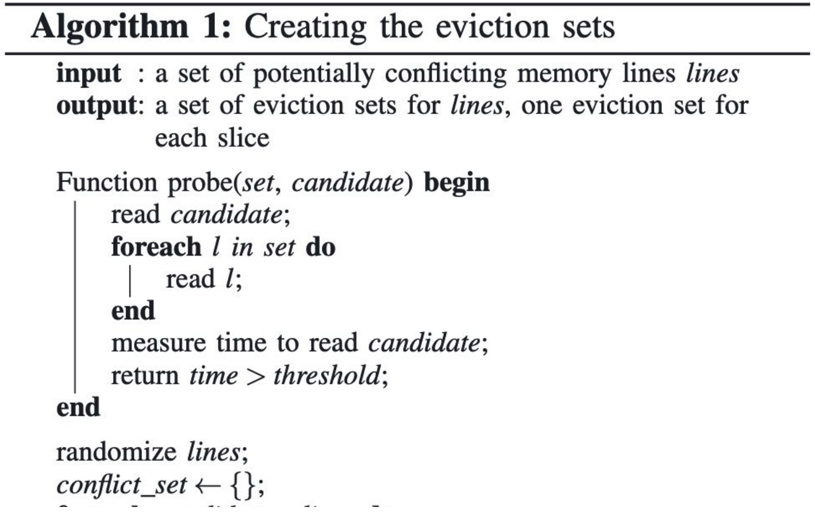


Creating Eviction Sets

- An eviction set fills up a certain set in the cache
- Nontrivial to create because LLC is physically addressed
- Algorithm:
    - For each line:
        - Build up a “conflict set”, which includes the minimum amount of conflicting lines required to evict the set, plus a bunch of other irrelevant lines (lines that had the same set bits but map to a different slice)
        - Create an eviction set by checking the lines not in the conflict set against the conflict set, if one conflicts then we can use it to figure out exactly what cache lines conflict with it

Example Eviction Set




Algorithm 1: Creating the eviction sets

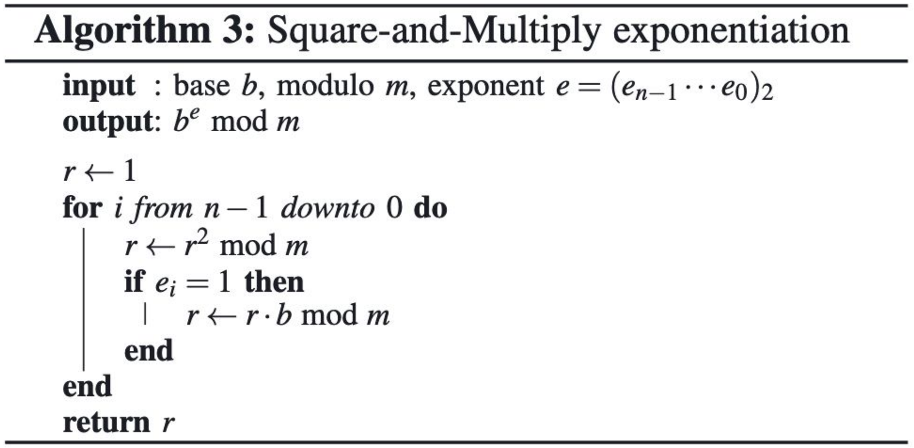
```
input  : a set of potentially conflicting memory lines *lines*
output: a set of eviction sets for *lines*, one eviction set for
        each slice

Function probe(set, candidate) begin
    read candidate;
    foreach l in set do
        read l;
    end
    measure time to read candidate;
    return $time > threshold$;
end

randomize lines;
conflict_set ← {};
```

Prime + Probe Attack

- Assume we know which line of cache relates to a sensitive function that exposes a secret key (we can figure this out based on access patterns)
- Insert an eviction set to force the sensitive line out of cache
- Repeatedly measure whether eviction set is still in cache (by measuring its access time) to detect when sensitive line is accessed

LLC Prime+Probe on Square-And-Multiply

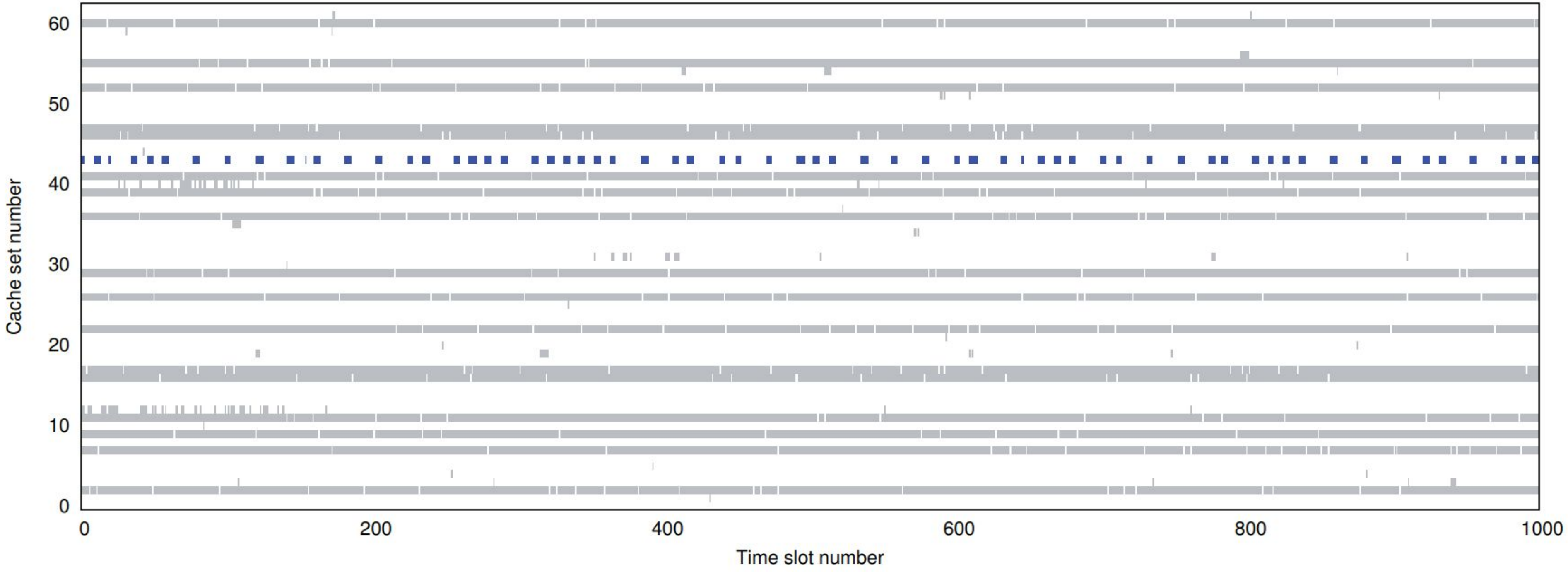
- Pay attention to the cache access patterns
  - When the exponent is a 1 then there is the squaring, a modulo, then a multiplication and an additional modulo
  - A 0 only has the squaring and reduction
- If we can find what cache set the square and multiply is grabbing the exponent from, then short periods between accesses will be a 0 and longer gaps will be a 1



Algorithm 3: Square-and-Multiply exponentiation

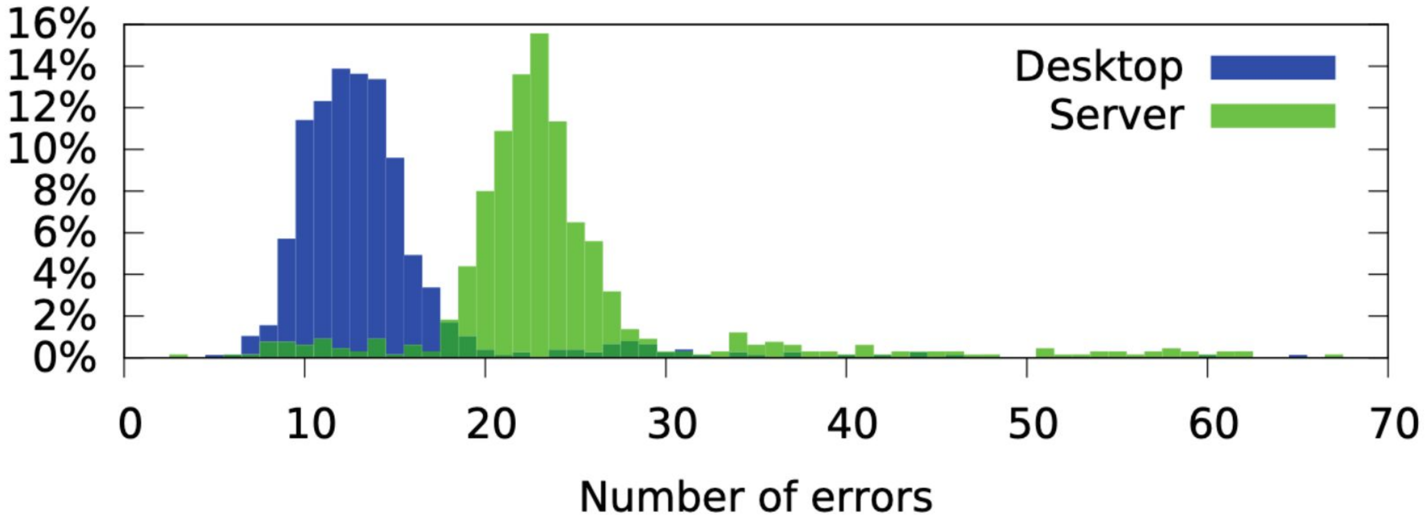
```
input  : base $b$, modulo $m$, exponent $e = (e_{n-1} \cdots e_0)_2$
output: $b^e$ mod $m$

$r \leftarrow 1$
for $i$ from $n-1$ downto $0$ do
    $r \leftarrow r^2$ mod $m$
    if $e_i = 1$ then
        $r \leftarrow r \cdot b$ mod $m$
    end
end
return $r$
```

LLC Prime+Probe on Square-And-Multiply



Square-And-Multiply Exponentiation Results

- 120/240 executions of the victim needed to locate the cache set of the squaring code on the desktop and server respectively

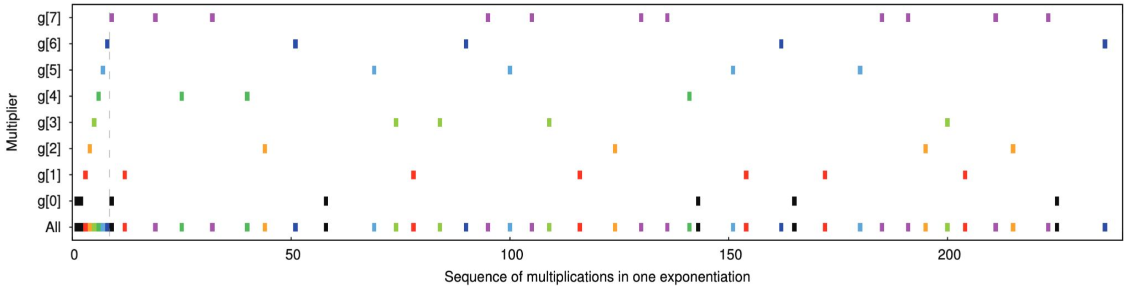


Sliding-Window Exponentiation Attack (Skip?)

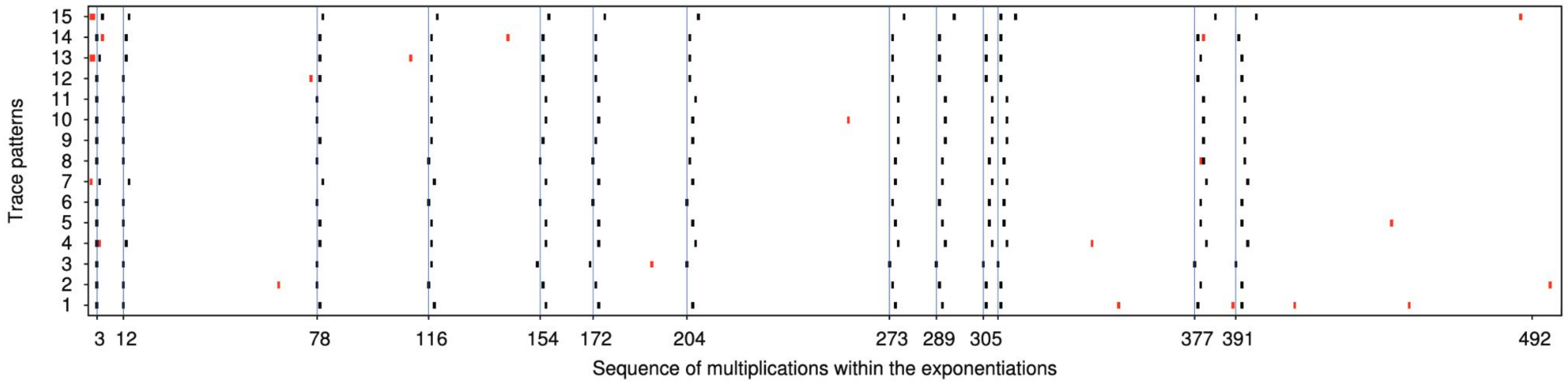
- The previous attack recovered secret-dependent execution paths
- The LLC Prime+Probe can also recover secret-dependent data access patterns

Algorithm 4: Sliding-window exponentiation

```
input : window size $S$, base $b$, modulo $m$,
        $N$-bit exponent $e$ represented as $n$ windows $w_i$ of
        length $L(w_i)$
output: $b^e$ mod $m$

//Precomputation
$g[0] \leftarrow b$ mod $m$
$s \leftarrow$ MULT($g[0], g[0]$) mod $m$
for $j$ from 1 to $2^{S-1}$ do
    $g[j] \leftarrow$ MULT($g[j-1], s$) mod $m$
end

//Exponentiation
$r \leftarrow 1$
for $i$ from $n$ downto 1 do
    for $j$ from 1 to $L(w_i)$ do
        $r \leftarrow$ MULT($r, r$) mod $m$
    end
    if $w_i \neq 0$ then $r \leftarrow$ MULT($r, g[(w_i - 1)/2]$) mod $m$;
end
return $r$
```



Sliding-Window Exponentiation Results



Mitigations

- Fix GnuPG
  - Exponent blinding: split the exponent into two parts, run modular exponentiation on both, and then combine
  - Use a constant-time implementation like we saw last lecture
    - a “constant-time” implementation of OpenSSL is still susceptible to timing attacks at least on the ARM architecture (Cook et al., 2014)
- Avoid resource contention
  - Don’t allow co-residency of VMs
  - Use cache partitioning
    - One option is StealthMem which gives a few stealth pages for secret-sensitive code and data

Discussions

Provided Discussion Questions

1. In Section III-C, the authors stated that “constructing an efficient PRIME+PROBE attack on the LLC is much harder than on the L1 caches” and listed 5 challenges. Briefly describe what methods the authors use to tackle each of these challenges.

2. Consider designing a new cache that uses a secret mapping function. Specifically, given an address, it uses an encryption algorithm (with a key unknown to the attacker) to decide the cache set index. Can you come up with an approach (inspired from this paper) to construct an eviction set on this cache? Briefly describe your approach.

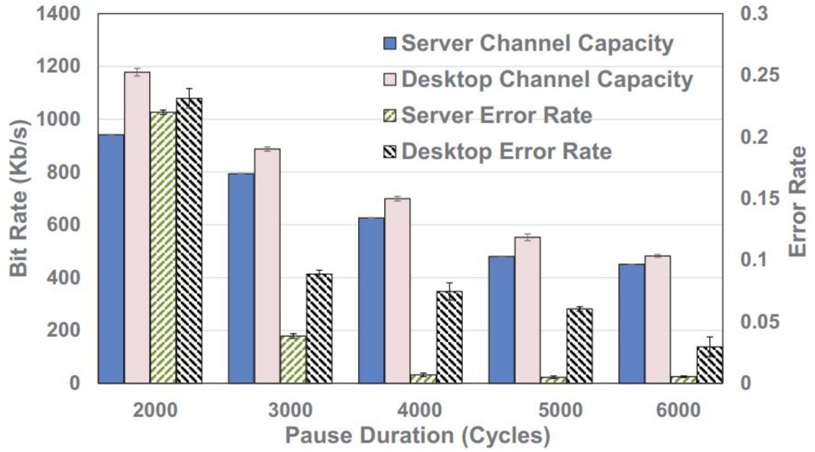
Student Questions

- This paper is from 2015. What was the response to mitigate/blunt the attack over the past 8 years?
- To what extent do exclusive cache hierarchies eliminate this attack? Can this attack still be pulled off now that modern Intel processors use non-inclusive LLCs?
- Why does the eviction set need to worry about the cache slices?
- In a cloud environment, would it be possible to make hardware changes to "turn off" the LLC entirely? If there is no resource to be shared, then you wouldn't have to worry about resource contention? Is there too great of a performance impact?

Additional Slides

Probing Bandwidth

- Has a sender and receiver that are both continuously running
  - Sender probes the same line many times to send 1s, different line for 0s
  - Receiver probes the 0/1 set repeatedly
- Up to 1.2MHz BW with 22% error rate
- 6 times faster than past LLC attacks


Server Channel Capacity
Desktop Channel Capacity
Server Error Rate
Desktop Error Rate
Bit Rate (Kb/s)
Error Rate
Pause Duration (Cycles)