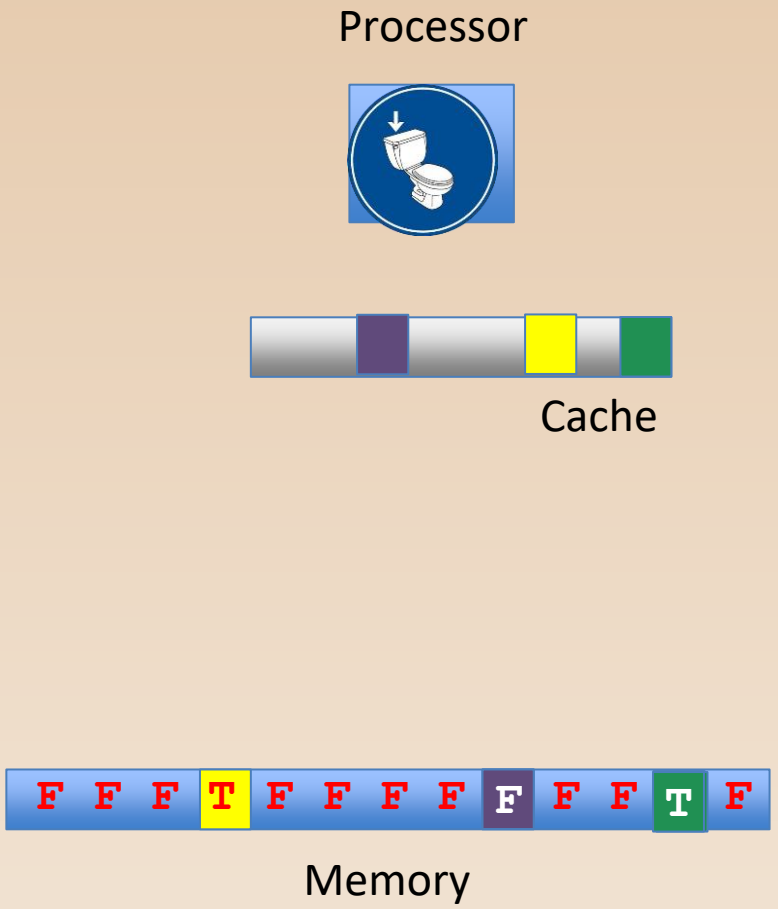
Gates of Time

Logical State of Cache

- Associate a logical value with memory addresses
  - TRUE – address is cached
  - FALSE – address is not cached
- Flushing sets a value to FALSE
- Accessing memory sets a value to TRUE (may also set another to FALSE)
- Measuring access time observes value (and set to TRUE)
- What else?



Conditional access

- What is the cache state of **`*out`** after execution?
- TRUE if **`*in != 0`**.
- What if **`*in == 0`**?

```
if (*in == 0)
    return;
a = *out
```

Speculative execution

- Evaluation of branch conditions can take time
- The CPU predicts future execution
  - Correct prediction – win
  - Incorrect prediction – rollback
  - Microarchitectural state remains

```
if (*in == 0)
    return;
a = *out
```

May be executed even if `*in == 0`

Conditional Speculative Execution

- Speculation terminates when condition is resolved

```
if (*in == 0)
    return;
a = *out
```

- `*in` in cache
  - Condition resolves fast
  - `*out` is not accessed

- `*in` not in cache
  - Condition resolution delayed
  - `*out` is accessed

*in	*out
TRUE	FALSE
FALSE	TRUE

out ← NOT(in)

Branch training

```
void not(int *out, int *in) {
  for (k=0; k < 128; k++)
    ;

  _mm_clflush(out);
  _mm_mfence();
  _mm_lfence();

  if (*in == 0)
    return;

  out *= 1;
  …
  a = *out;
}
```

```
void callnot(int *out, int *in) {
  int dummy = 1;
  not(&dummy, &dummy);
  not(&dummy, &dummy);

  not(out, in);
}
```

Return as branch

```
void not(int *out, int *in) {
    call 1f
    ; DELAY on %rax
    mov (%rdi, %rax), %rax
    lfence
  1:
    mov $2f, (%rsp)
    mov (%rsi), %r11
    add %r11, (%rsp)
    ret
  2:
}
```

Other functions

```
if (*in1 == 0)
    return;
if (*in2 == 0)
    return;
a = *out
```

*in1	*in2	*out
FALSE	FALSE	TRUE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	FALSE

out ← NOR(in1, in2)

```
if (*in1 + *in2 == 0)
    return;
a = *out
```

*in1	*in2	*out
FALSE	FALSE	TRUE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	FALSE

out ← NAND(in1, in2)

Multiple outputs

- Needed because gates destroy their inputs
- Limited by number of line fill buffers
  - Can handle up to 12 inputs and outputs

```
if (*in == 0)
    return;
a = *out1 + *out2
```

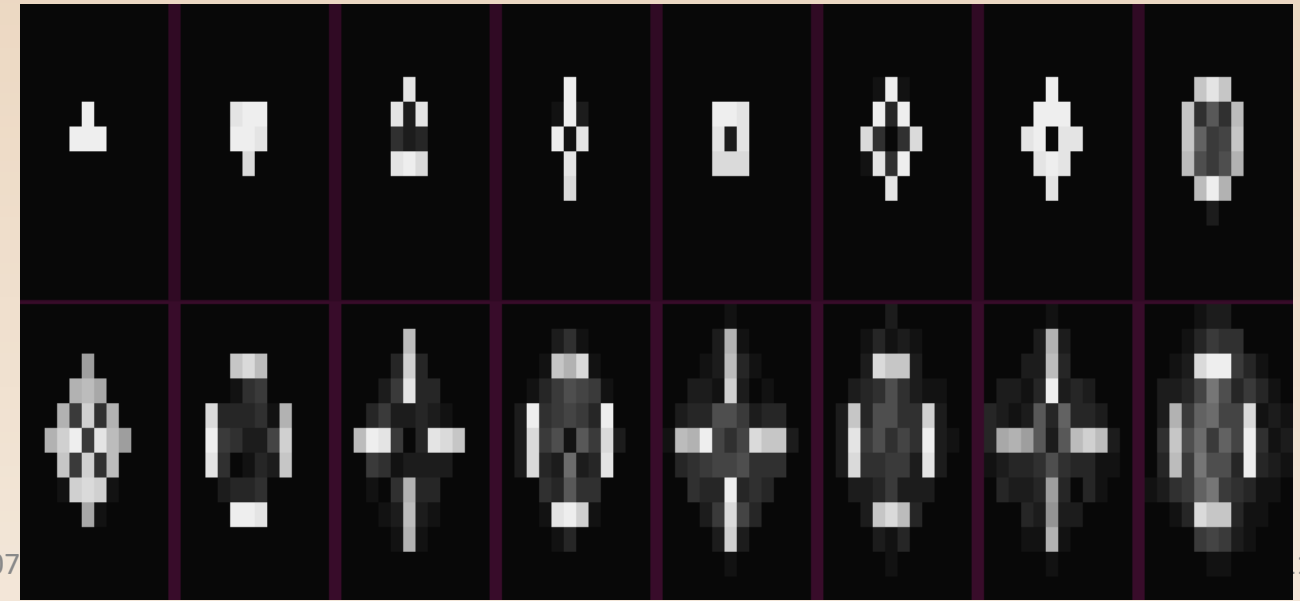
Minority Report

```
if (*in1 + *in2 == 0)
    return;
if (*in2 + *in3 == 0)
    return;
if (*in1 + *in3 == 0)
    return;
a = *out
```

*in1	*in2	*in3	*out
FALSE	FALSE	FALSE	TRUE
FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE
TRUE	FALSE	TRUE	FALSE
TRUE	TRUE	FALSE	FALSE
TRUE	TRUE	TRUE	FALSE

Circuits

- 4-bit ALU
  - 1258 gates, 84-95% accuracy
- SHA-1
  - One round: 2208 gates, 95% accuracy (67% with prefetcher)
  - Full (two blocks, with repetitions) 95% accuracy
- Game of Life
  - 7807 gates 73% accuracy for one generation, 25% for 20

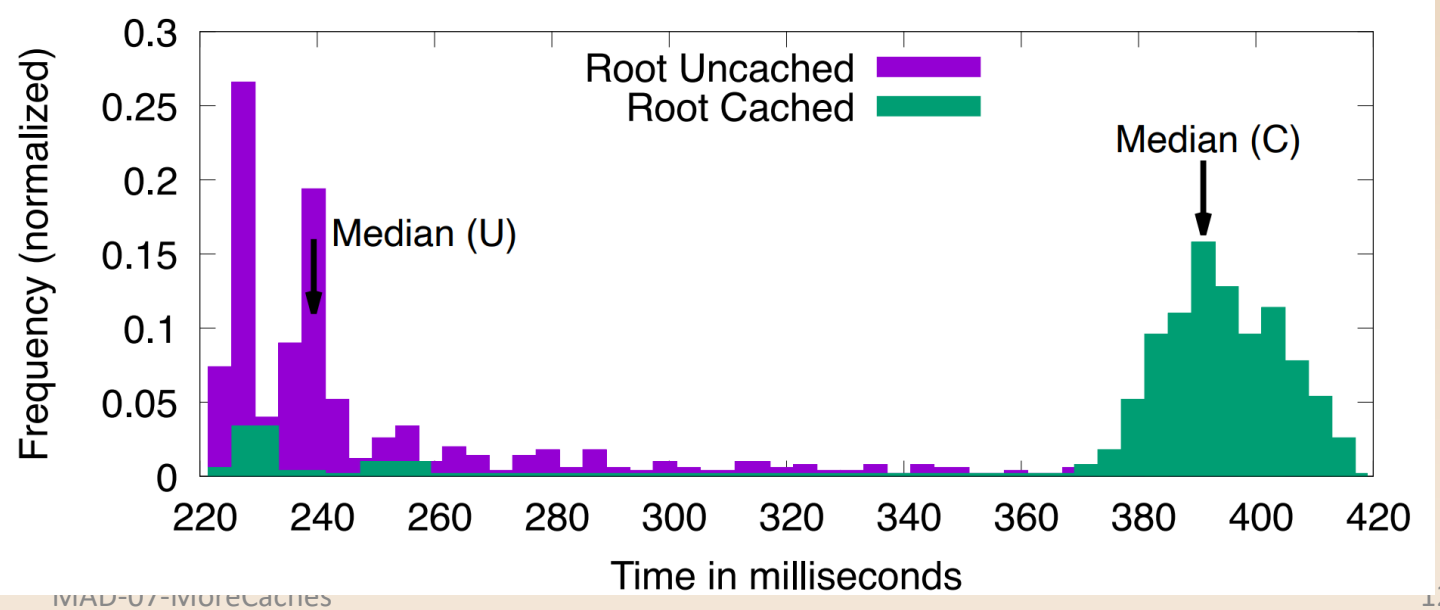


Amplification

- A NOT gate with a large fan-out amplifies the signal by a factor of 8
  - Two layers – 64
  - Three layers – 512
  - Four layers – 4096

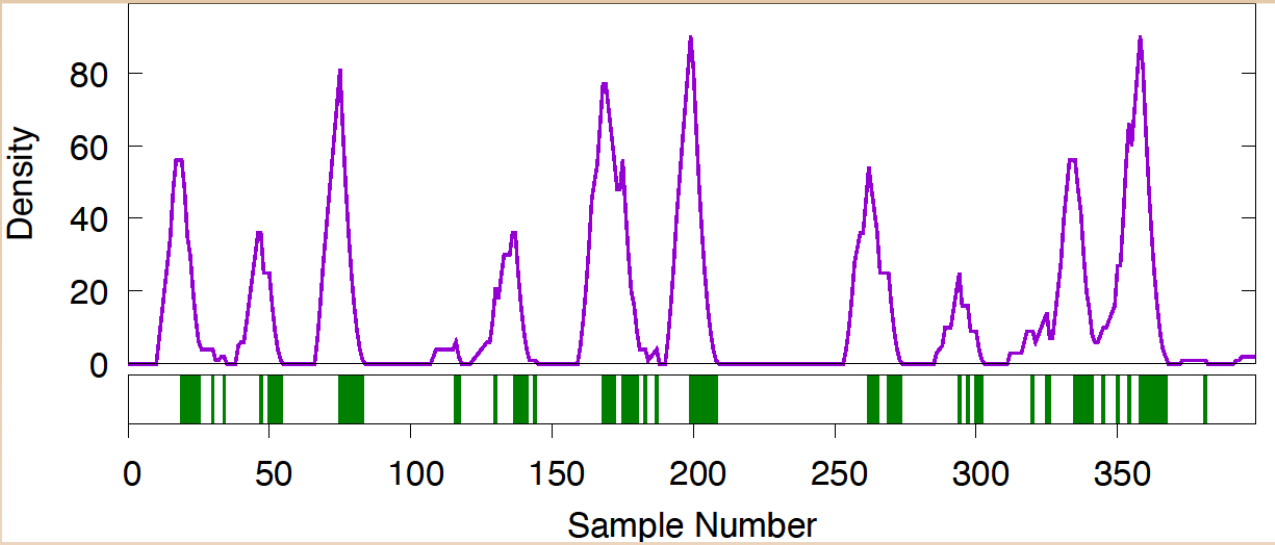
```
if (*in == 0)
    return;
a = *out1 + *out2 +
    *out3 + *out4 +
    *out5 + *out6 +
    *out7 + *out8;
```

- Amplify to a resolution of 0.1 second



Prime+Store: High Resolution Prime+Probe

- Probe is basically a NAND gate
- Do multiple probes of the same cache set. Store results.
- Amplify later
  - Decouples probing from time measurements
- Attack square-and-multiply ElGamal with a 0.1ms clock



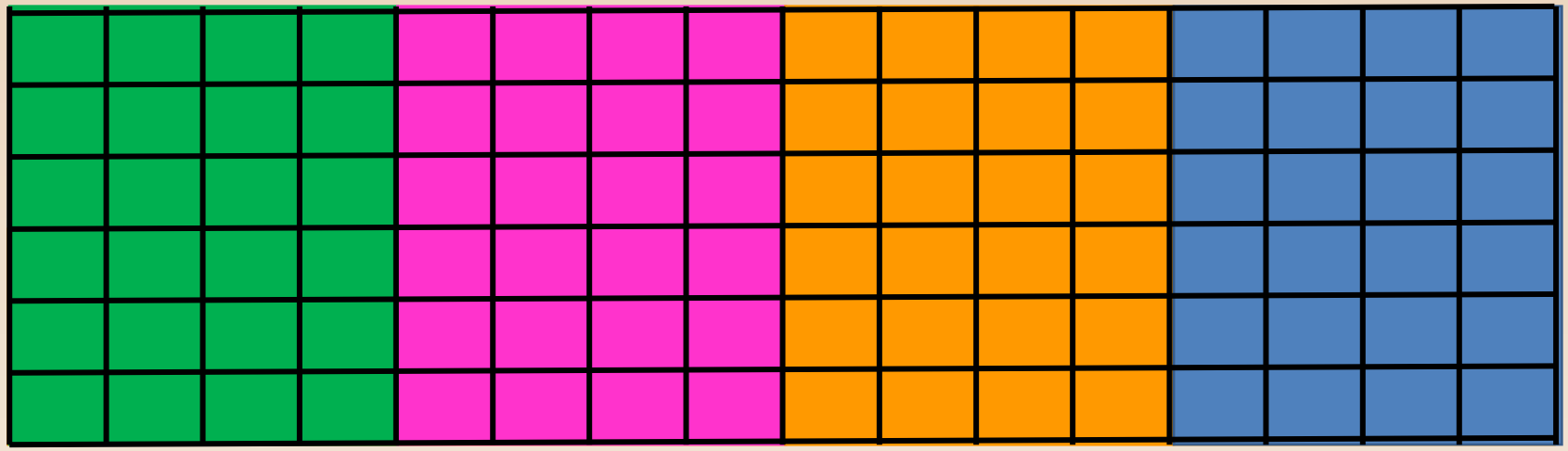
Countermeasures

Classification

- Hardware
  - Partitioning
  - Randomization
- Operating System
  - Spatial and temporal partitioning
  - Detection
- Software

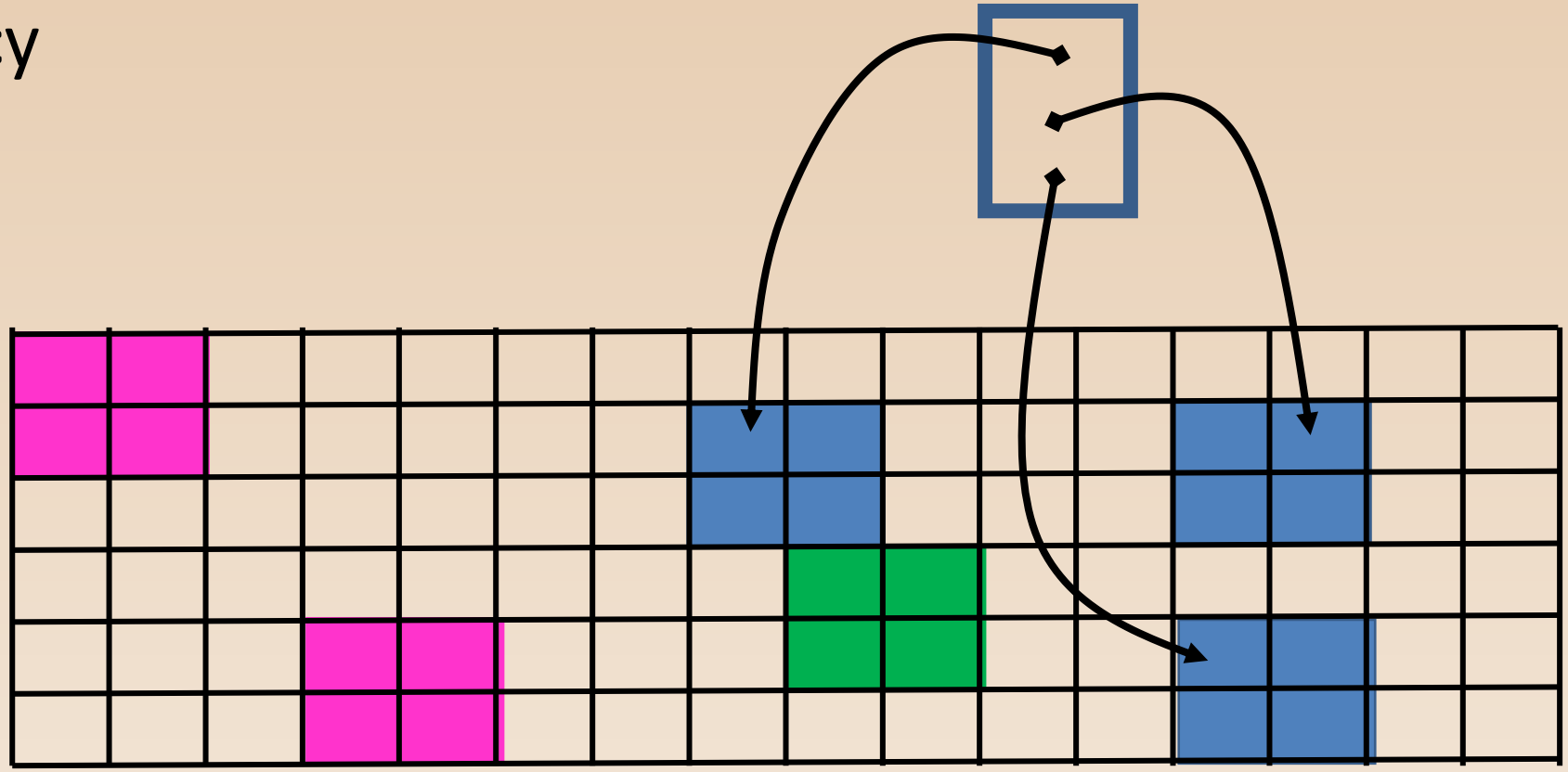
Hardware partitioning

- Set partitioning
- Way partitioning
- Cachelets



Challenges

- Mapping
  - Virtual Partitioning Table
- Coherency



Intel Cache Allocation Technology

- Performance management – prevent cache starvation
- Associates a mask of LLC ways with a (virtual) CPU
- Cache replacement only occurs within the masked ways

- Limitations:
  - Does not necessarily partition the cache
  - Can serve data from outside the masked ways
  - Does not prevent flush
  - Hardware functionality uses the cache
  - Does not protect the cache directory

Randomization

- Breaks (known) relationship between memory addresses and cache sets

- RPCache (Wang and Lee, ISCA 2007) – each process uses a different (random) permutation of the sets.

ScatterCache (Werner et al. USENIX Sec 2019)

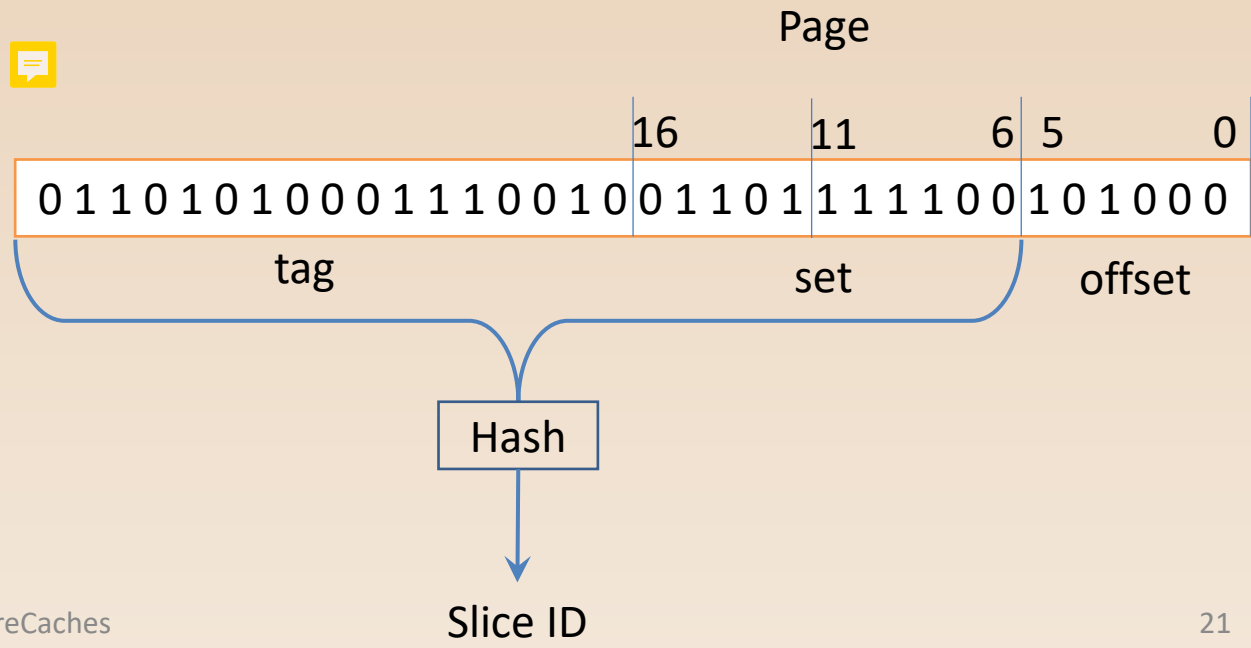
- Uses cryptographic function for mapping memory to cache sets
- A different map for each way
- Per process mapping
- Periodic rekeying
- Expected ~2 cycle overhead per cache access – negligible performance impact

Operating System

- Cache Coloring

- Basically, OS-based set partitioning.
  - Use 5 address bits + slice ID (for some processors)

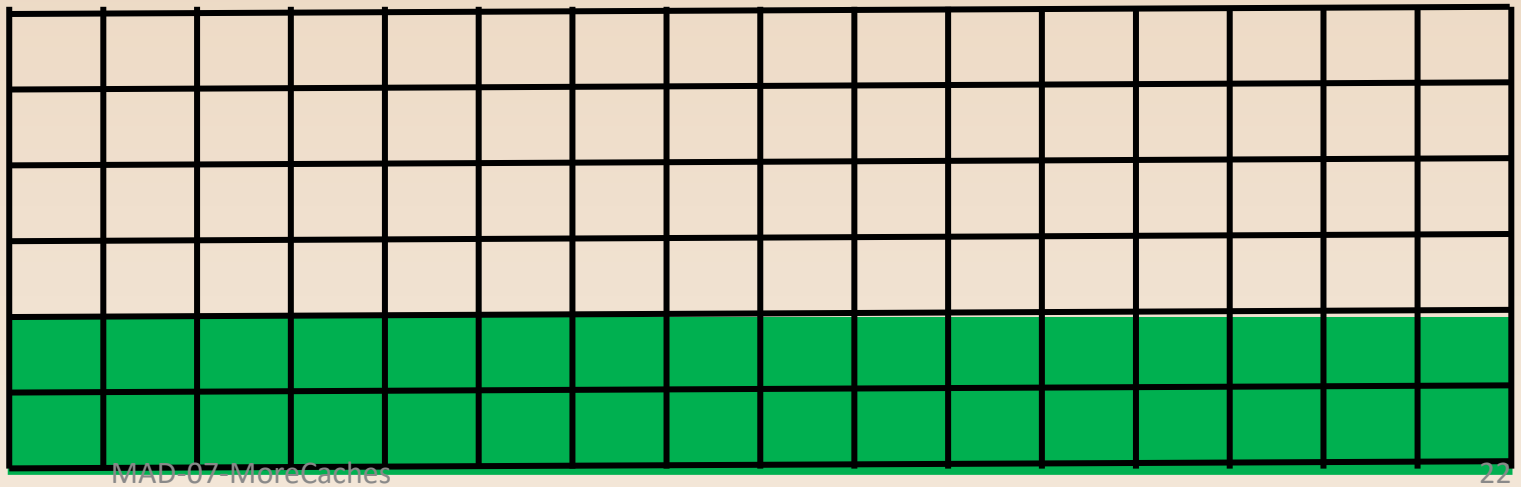
- Page table walk as confused deputy (van Schaik et al. USENIX Sec 2018)



CATalyst (Liu et al. HPCA 2016)

- Core idea: lock sensitive data in cache to prevent attacks.
  - Originally from Wang and Lee (ISCA 2007)

- Use CAT to define a secure region in the cache
  - Prevents replacement from processes
- Process requests to load memory to the secure region
- Operating system ensures that the data is cached until the process releases the region



Constant-time programming

- Three causes of leaks through side channel attacks:
- Secret-dependent control flow
- Secret-dependent memory accesses
- Secret arguments to instruction with data-dependent execution time

$x \leftarrow 1$
for $i \leftarrow |d|-1$ **downto** 0 do
  $x \leftarrow x^2 \bmod n$
  if $(d_i = 1)$ **then**
    $x = xC \bmod n$
  endif
done
return x

Attacking AES

```
    s0 = GETU32(in     ) ^ rk[0];
    s1 = GETU32(in +  4) ^ rk[1];
    s2 = GETU32(in +  8) ^ rk[2];
    s3 = GETU32(in + 12) ^ rk[3];
#ifdef FULL_UNROLL
    /* round 1: */
    t0 = Te0[s0 >> 24] ^ Te1[(s1 >>
    t1 = Te0[s1 >> 24] ^ Te1[(s2 >>
    t2 = Te0[s2 >> 24] ^ Te1[(s3 >>
    t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 1
    /* round 2: */
    s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >>
    s1 = Te0[t1 >> 24] ^ Te1[(t2 >> 16) & 0xff] ^ Te2[(t3 >>
```

```
static const u32 Te0
    0xc66363a5U, 0xf
    0xfff2f20dU, 0xd
    0x60303050U, 0x0
    0xe7fefe19U, 0xb
    0x8fcaca45U, 0x1
    0xeffafa15U, 0xb
    0x41adadecU, 0xb
    0x239c9cbfU, 0x5
```

Eliminating secret-dependent branches

- Compute both branches
- Choose the results of the correct branch
  - How to do that without secret dependent branches?

Square and Multiply

```
x ⟵1
for i ⟵|d|-1 downto 0 do
    x ⟵x^2 mod n
    if (d_i = 1) then
        x = xC mod n
    endif
done
return x
```

Square and Multiply always

```
x ⟵1
for i ⟵|d|-1 downto 0 do
    x ⟵x^2 mod n
    t = xC mod n
    if (d_i = 1) then
        x = t
    endif
done
return x
```

Constant-time select

- X = ct_select(cond, a, b)

- Use conditional move:

  MOV RDX, [a]

  CMOVcc RDX, [b]

- Use arithmetic

Class exercise

- What do the following functions compute?

```
int32_t f1(int32_t a) {
  return a & -a;
}

int32_t f2(int32_t a) {
  return (a | -a) >> 31;
}

uint32_t f3(int32_t a, uint32_t b, uint32_t c) {
  a = (a | -a) >> 31;
  return (b & a) | (c & ~a);
}
```

Secret dependent memory accesses

- Access all memory locations
- Use constant-time select to choose the correct value

Caveats

- Accessing each cache line is not enough
- Constant-time select of pointers is not good enough

```
static byte GetTable8(const byte* t, byte o)
{
#if WC_CACHE_LINE_SZ == 64
    byte e;
    byte hi = o & 0xf0;
    byte lo = o & 0x0f;

    e  = t[lo + 0x00] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0x10] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0x20] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0x30] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0x40] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0x50] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0x60] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0x70] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0x80] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0x90] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0xa0] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0xb0] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0xc0] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0xd0] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0xe0] & ((word32)0 - (((word32)hi - 0x01) >> 31)); hi -= 0x10;
    e |= t[lo + 0xf0] & ((word32)0 - (((word32)hi - 0x01) >> 31));

    return e;
#else
```

Summary

- Secure cache designs
- Operating system countermeasures
- Constant-time programming

- Next lecture: Spectre-type attacks
  - Read: P. Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”, IEEE SP 2019.