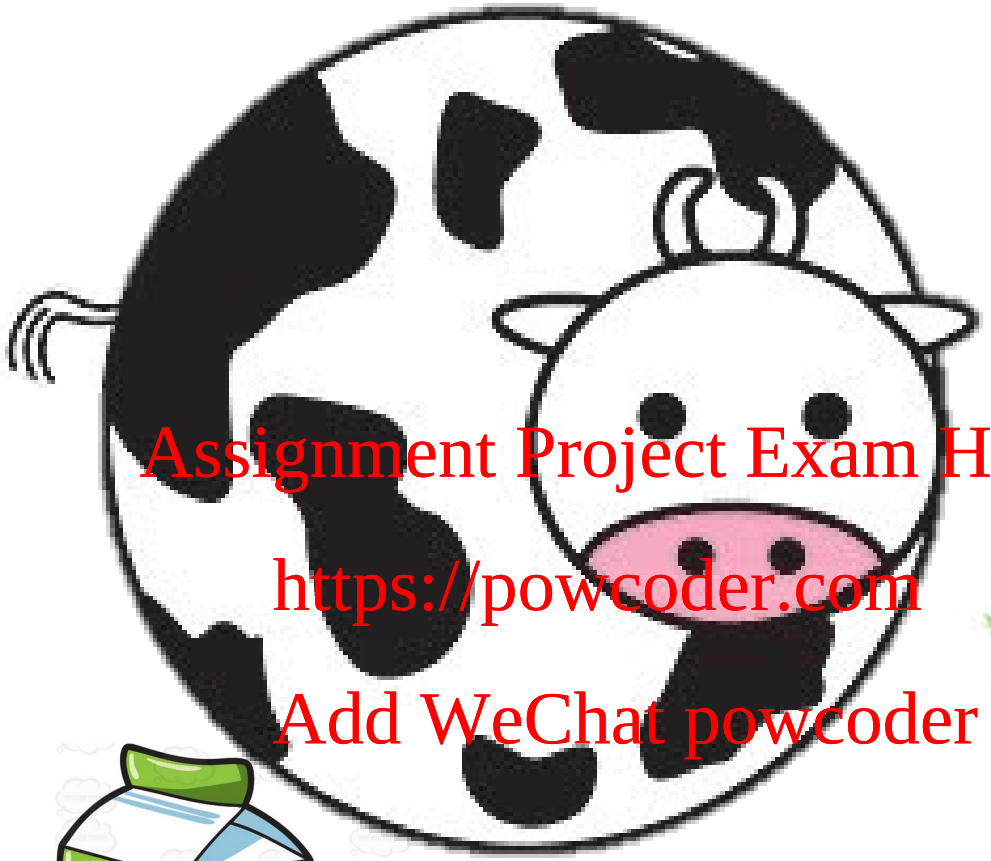
Side Channel Attacks



Key

plaintext

ciphertext

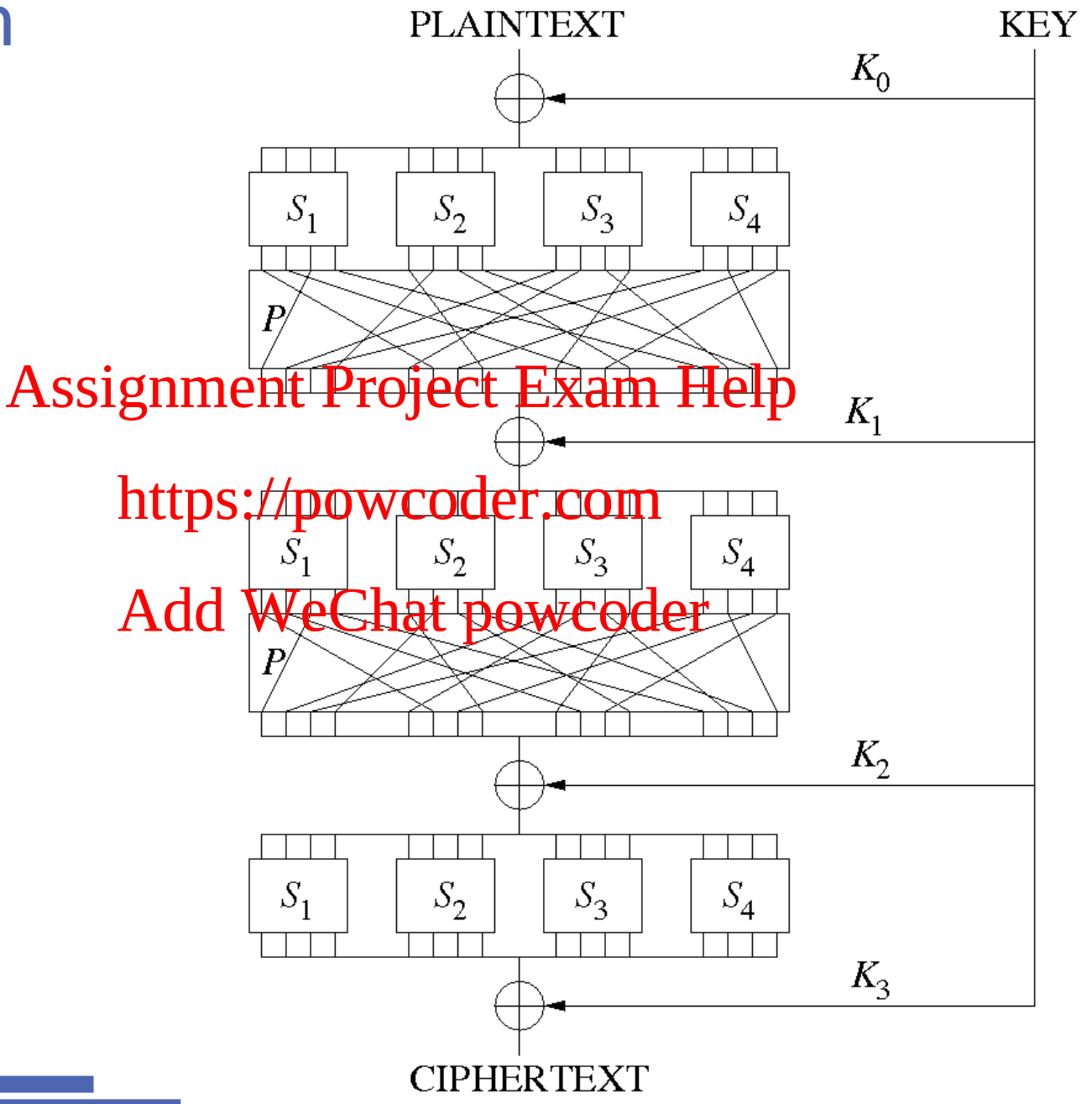
Key

Side channel

plaintext

ciphertext

AES Implementation



AES Implementation

```
    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 >>
    /* round 2: */
    s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >>  8) & 0xff] ^ Te3[t3 & 0xff] ^ rk[ 8];
    s1 = Te0[t1 >> 24] ^ Te1[(t2 >> 16) & 0xff] ^ Te2[(t3 >>  8) & 0xff] ^ Te3[t0 & 0xff] ^ rk[ 9];
    s2 = Te0[t2 >> 24] ^ Te1[(t3 >> 16) & 0xff] ^ Te2[(t0 >>  8) & 0xff] ^ Te3[t1 & 0xff] ^ rk[10];
    s3 = Te0[t3 >> 24] ^ Te1[(t0 >> 16) & 0xff] ^ Te2[(t1 >>  8) & 0xff] ^ Te3[t2 & 0xff] ^ rk[11];
    /* round 3: */
    t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >>  8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[12];
    t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >>  8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[13];
    t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >>  8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[14];
    t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >>  8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[15];
    /* round 4: */
    s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >>  8) & 0xff] ^ Te3[t3 & 0xff] ^ rk[16];
```

```
static const u32 Te0[256] = {
    0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
    0xfff2f20dU, 0xd66b6bbdU, 0xde6f6fb1U, 0x91c5c554U,
    0x60303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,
    0xe7fefe19U, 0xb5d7d762U, 0x4dababe6U, 0xec76769aU,
    0x8fcaca45U, 0x1f82829dU, 0x89c9c940U, 0xfa7d7d87U,
    0xeffafa15U, 0xb25959ebU, 0x8e4747c9U, 0xfbf0f00bU,
    0x42adadecU, 0xb3d4d467U, 0x5fa2a2fdU, 0x45afafeaU,
    0x239c9cbfU, 0x53a4a4f7U, 0xe4727296U, 0x9bc0c05bU,
```

AES T-table access

```
static const u32 Te0[256] = {
    0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
    0xfff2f20dU, 0xd66b6bbdU, 0xde6f6fb1U, 0x91c5c554U,
    0x60303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,
    0xe7fefe19U, 0xb5d7d762U, 0x4dababe6U, 0xec76769aU,
```



```
s0 = plaintext ^ key
t0 = Te0[s0>>24]
```

- Assume we know the plaintext and the index (s0>>24)
  - **We can recover the most significant byte of the key**

AES Implementation

```
    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 >> 16) & 0xff] ^ Te2[(s2 >>  8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[ 4];
    t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >>  8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[ 5];
    t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >>  8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[ 6];
    t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >>  8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[ 7];
    /* round 2: */
    s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >>  8) & 0xff] ^ Te3[t3 & 0xff] ^ rk[ 8];
```


- **If we know the plaintext and all of the indices in the first round we can recover the key.**

The Microarchitecture

CPU vs. Memory

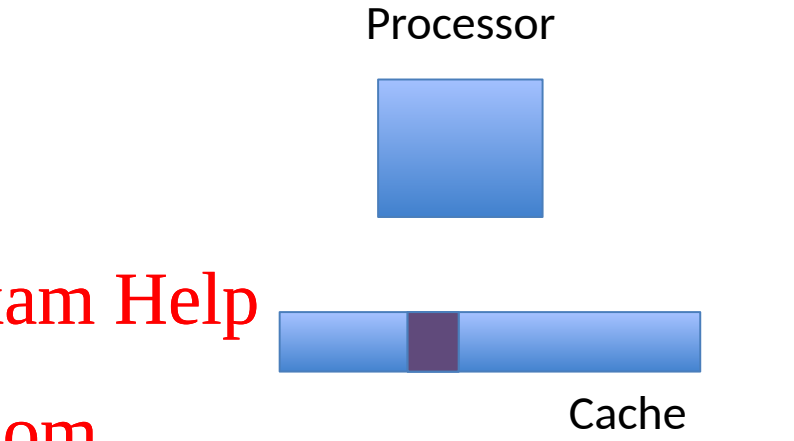
	Processor Speed	Memory Latency
	1 MHz	500 ns
	8*2600 MHz	63 ns

Bridging the gap

Cache utilises locality to bridge the gap

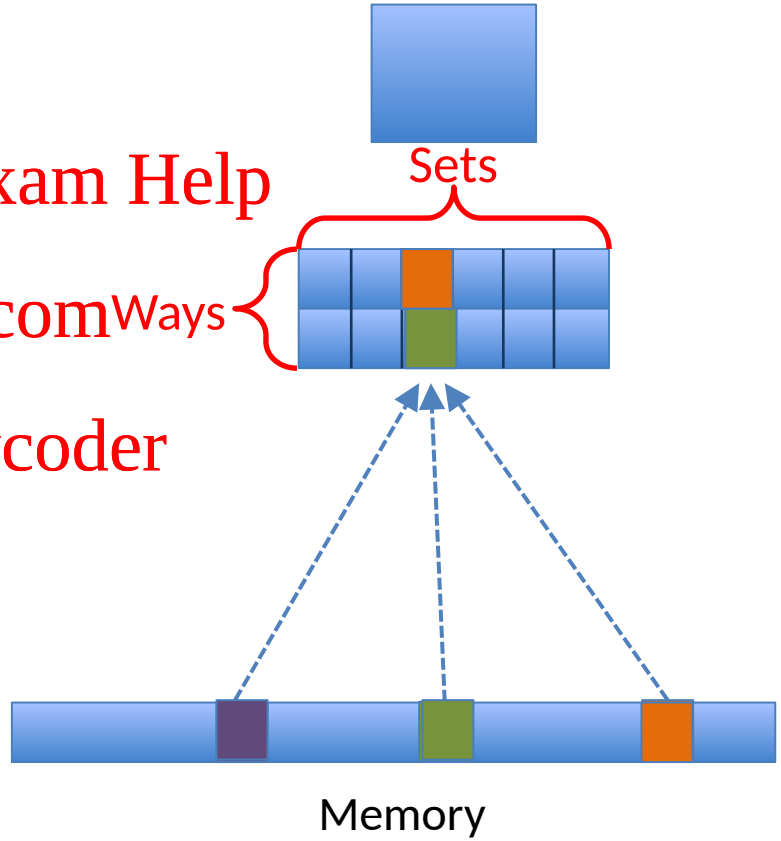
- Divides memory into *lines*
- Stores recently used lines

- In a *cache hit*, data is retrieved from the cache
- In a *cache miss*, data is retrieved from memory and inserted to the cache



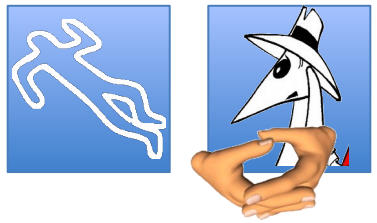
Set Associative Caches

- Memory lines map to *cache sets*. Multiple lines map to the same set.
- Sets consist of *ways*. A memory line can be stored in **any** of the ways of the set it maps to.
- When a cache miss occurs, one of the lines in the set is *evicted*.



The Prime+Probe Attack

- Allocate a cache-sized memory buffer
- *Prime*: fills the cache with the contents of the buffer
- *Probe*: measure the time to access each cache set
  - Slow access indicates victim access to the set
- The probe phase primes the cache for the next round



Memory

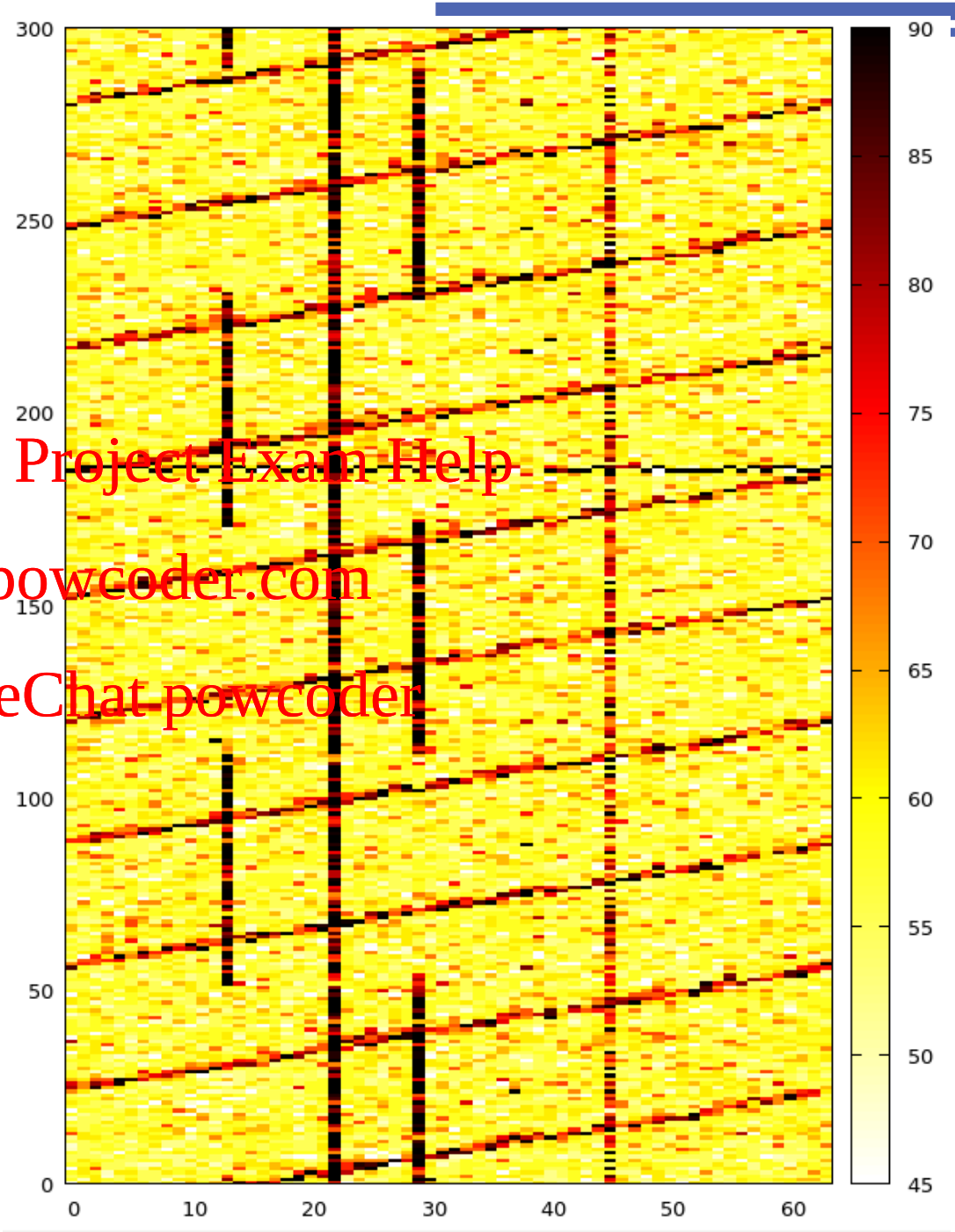
Sample Victim: Data Rattle

```
volatile char buffer[4096];

int main(int ac, char **av) {
  for (;;) {
    for (int i = 0; i < 64000; i++)
      buffer[800] += i;

    for (int i = 0; i < 64000; i++)
      buffer[1800] += i;
  }
}
```

Cache Fingerprint of the Rattle Program



AES T-tables and cache lines

```
static const u32 Te0[256] = {
    0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
    0xfff2f20dU, 0xd66b6bbdU, 0xde6f6fb1U, 0x91c5c554U,
    0x60303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,
    0xe7fefe19U, 0xb5d7d762U, 0x4dababe6U, 0xec76769aU,
    0x8fcaca45U, 0x1f82829dU, 0x89c9c940U, 0xfa7d7d87U,
    0xeffafa15U, 0xb25959ebU, 0x8e4747c9U, 0xfbf0f00bU,
    0x41adadecU, 0xb3d4d467U, 0x5fa2a2fdU, 0x45afafeaU,
    0x239c9cbfU, 0x53a4a4f7U, 0xe4727296U, 0x9bc0c05bU,
    0x75b7b7c2U, 0xe1fdfd1cU, 0x3d9393aeU, 0x4c26266aU,
    0x6c36365aU, 0x7e3f3f41U, 0xf5f7f702U, 0x83cccc4fU,
    0x6834345cU, 0x51a5a5f4U, 0xd1e5e534U, 0xf9f1f108U,
    0xe2717193U, 0xabd8d873U, 0x62313153U, 0x2a15153fU,
    0x0804040cU, 0x95c7c752U, 0x46232365U, 0x9dc3c35eU,
    0x30181828U, 0x379696a1U, 0x0a05050fU, 0x2f9a9ab5U,
    0x0e070709U, 0x24121236U, 0x1b809b5bU, 0xdfe2e23dU,
    0xcdebeb26U, 0x4e272769U, 0x7fb2b2cdU, 0xea75759fU,
    0x1209091bU, 0x1d83839eU, 0x582c2c74U, 0x341a1a2eU,
    0x361b1b2dU, 0xdc6e6eb2U, 0xb45a5aeeU, 0x5ba0a0fbU,
    0xa45252f6U, 0x763b3b4dU, 0xb7d6d661U, 0x7db3b3ceU,
    0x5229297bU, 0xdde3e33eU, 0x5e2f2f71U, 0x13848497U,
    0xa65353f5U, 0xb9d1d168U, 0x00000000U, 0xc1eded2cU,
    0x40202060U, 0xe3fcfc1fU, 0x79b1b1c8U, 0xb65b5bedU,
    0xd46a6abeU, 0x8dcbcb46U, 0x67bebed9U, 0x7239394bU,
    0x944a4adeU, 0x984c4cd4U, 0xb05858e8U, 0x85cfcf4aU,
    0xbbd0d06bU, 0xc5efef2aU, 0x4faaaae5U, 0xedfbfb16U,
    0x864343c5U, 0x9a4d4dd7U, 0x66333355U, 0x11858594U,
    0x8a4545cfU, 0xe9f9f910U, 0x04020206U, 0xfe7f7f81U,
    0xa05050f0U, 0x783c3c44U, 0x259f9fbaU, 0x4ba8a8e3U,
```

Cache Line 0

Cache Line 1

Cache Line 2

Cache Line 3

Cache Line 4

Cache Line 5

AES T-tables and cache lines

```
static const u32 Te0[256] = {
    0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
    0xfff2f20dU, 0xd66b6bbdU, 0xde6f6fb1U, 0x91c5c554U,
    0x60303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,
    0xe7fefe19U, 0xb5d7d762U, 0x4dababe6U, 0xec76769aU,
    0x8fcaca45U, 0x1f82829dU, 0x89c9c940U, 0xfa7d7d87U,
    0xeffafa15U, 0xb25959ebU, 0x8e4747c9U, 0xfbf0f00bU,
    0x41adadecU, 0xb3d4d467U, 0x5fa2a2fdU, 0x45afafeaU,
    0x239c9cbfU, 0x53a4a4f7U, 0xe4727296U, 0x9bc0c05bU,
    0x75b7b7c2U, 0xe1fdfd1cU, 0x3d9393aeU, 0x4c26266aU,
    0x6c36365aU, 0x7e3f3f41U, 0xf5f7f702U, 0x83cccc4fU,
    0x6834345cU, 0x51a5a5f4U, 0xd1e5e534U, 0xf9f1f108U,
    0xe2717193U, 0xabd8d873U, 0x62313153U, 0x2a15153fU,
    0x0804040cU, 0x95c7c752U, 0x46232365U, 0x9dc3c35eU,
    0x30181828U, 0x379696a1U, 0x0a05050fU, 0x2f9a9ab5U,
```

Cache Line 0

Cache Line 1

Cache Line 2

- If 0≤plaintext[0]^key[0]<16, Cache Line 0 is accessed.
- What if plaintext[0]^key[0]≥16?

Analysing the AES Implementation

```
    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 >> 16) & 0xff] ^ Te2[(s2 >>  8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[ 4];
    t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >>  8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[ 5];
    t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >>  8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[ 6];
    t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >>  8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[ 7];
    /* round 2: */
    s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >>  8) & 0xff] ^ Te3[t3 & 0xff] ^ rk[ 8];
    s1 = Te0[t1 >> 24] ^ Te1[(t2 >> 16) & 0xff] ^ Te2[(t3 >>  8) & 0xff] ^ Te3[t0 & 0xff] ^ rk[ 9];
    s2 = Te0[t2 >> 24] ^ Te1[(t3 >> 16) & 0xff] ^ Te2[(t0 >>  8) & 0xff] ^ Te3[t1 & 0xff] ^ rk[10];
    s3 = Te0[t3 >> 24] ^ Te1[(t0 >> 16) & 0xff] ^ Te2[(t1 >>  8) & 0xff] ^ Te3[t2 & 0xff] ^ rk[11];
    /* round 3: */
    t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >>  8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[12];
    t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >>  8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[13];
    t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >>  8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[14];
    t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >>  8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[15];
    /* round 4: */
    s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >>  8) & 0xff] ^ Te3[t3 & 0xff] ^ rk[16];
```

Analysing the AES Implementation

```
    s0 = GETU32(in     ) ^
    s1 = GETU32(in +  4) ^
    s2 = GETU32(in +  8) ^
    s3 = GETU32(in + 12) ^
#ifdef FULL_UNROLL
    /* round 1: */
    t0 = Te0[s0 >> 24] ^ Te
    t1 = Te0[s1 >> 24] ^ Te
    t2 = Te0[s2 >> 24] ^ Te
    t3 = Te0[s3 >> 24] ^ Te
    /* round 2: */
    s0 = Te0[t0 >> 24] ^ Te
    s1 = Te0[t1 >> 24] ^ Te
    s2 = Te0[t2 >> 24] ^ Te
    s3 = Te0[t3 >> 24] ^ Te
    /* round 3: */
    t0 = Te0[s0 >> 24] ^ Te
    t1 = Te0[s1 >> 24] ^ Te
    t2 = Te0[s2 >> 24] ^ Te
    t3 = Te0[s3 >> 24] ^ Te
    /* round 4: */
    s0 = Te0[t0 >> 24] ^ Te
```

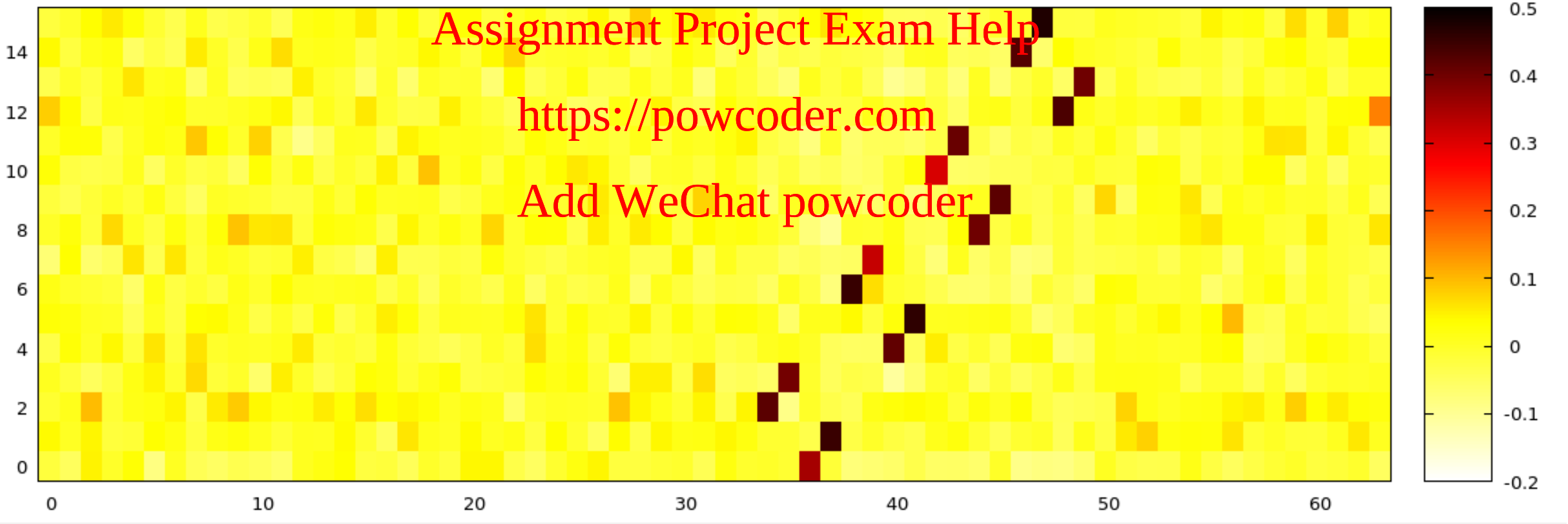
- Each round, Te0 is accessed 4 times
- AES has 10 rounds
  - Te0 is accessed 40 times in an AES encryption
- The first access misses Cache Line 0
- Each following access misses Cache Line 0 with a probability of 15/16
- The probability that all accesses miss Cache Line 0 is about 8%

Prime+Probe Attack on AES

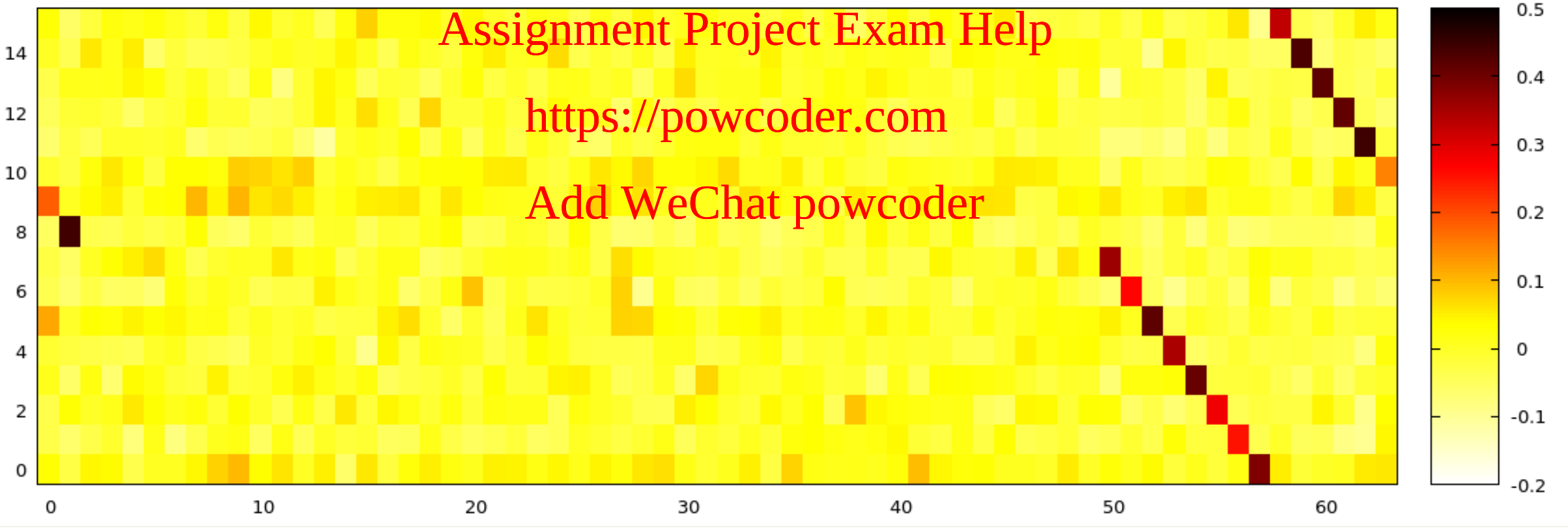
- Repeat 1000000 times:
  - Generate a random plaintext
  - Prime the cache
  - Encrypt the plaintext
  - Probe the cache and record results

- For each plaintext byte
  - Partition results based on the most significant half of a plaintext byte
  - Find the cache set with the slowest average access time for each partition
  - Identify the most significant half of the corresponding key byte

PP Attack on AES - Results



PP Attack on AES – More Results



What's now?

- Recover the second half of the key
  - Second round attack – similar but with ugly maths
- How to perform the attack
  - Easy: use Mastik:

    http://cs.ade
- How to defend?
  - Later…

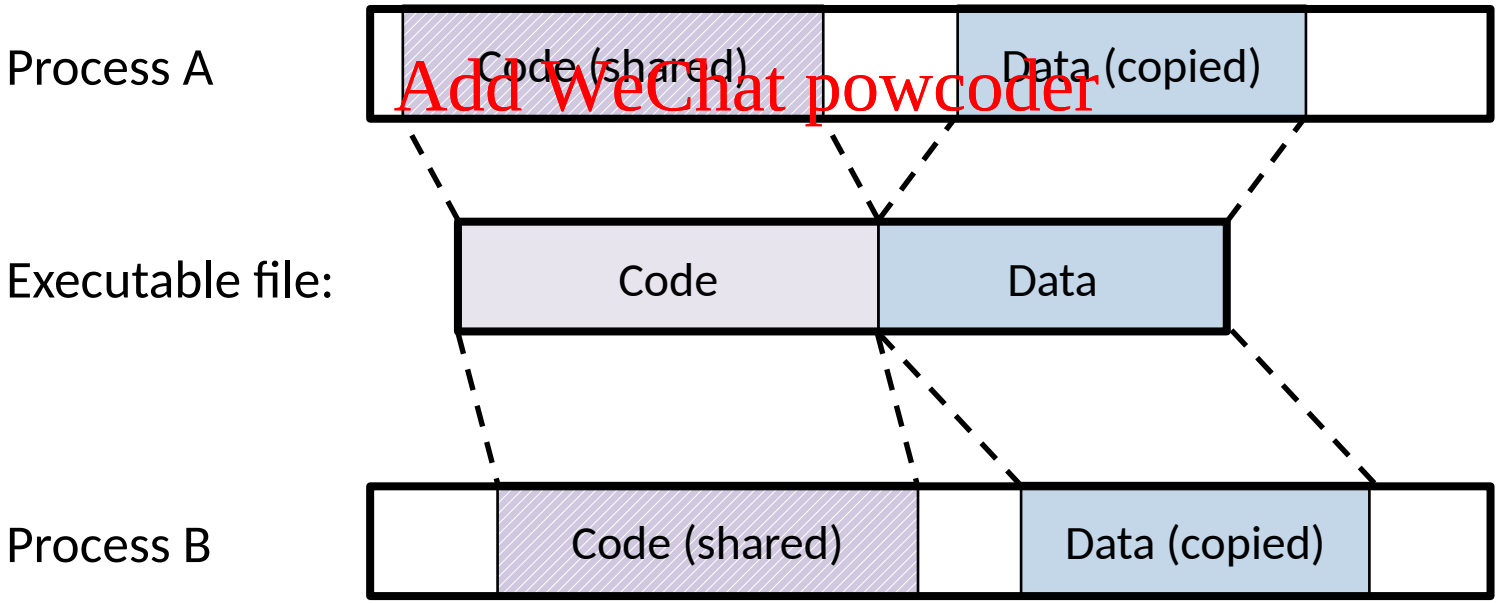

```
    for (int i = 0; i < NSAMPLES; i++) {
      randaes(input);
      l1_probe(l1, tmp);
      AES_encrypt(input, output, &aeskey);
      l1_bprobe(l1, tmp);
      for (int j = 0; j < 64; j++)
        results[i*64+map[j]] = tmp[j];
      for (int j = 0; j < 32; j++)
        rec[i][j] = input[j];
    }
```

The FLUSH+RELOAD Technique

- Leaks information on victim access to shared memory.
- Spy monitors victim's access to shared code
  - Spy can determine what victim does
  - Spy can infer the data the victim operates on

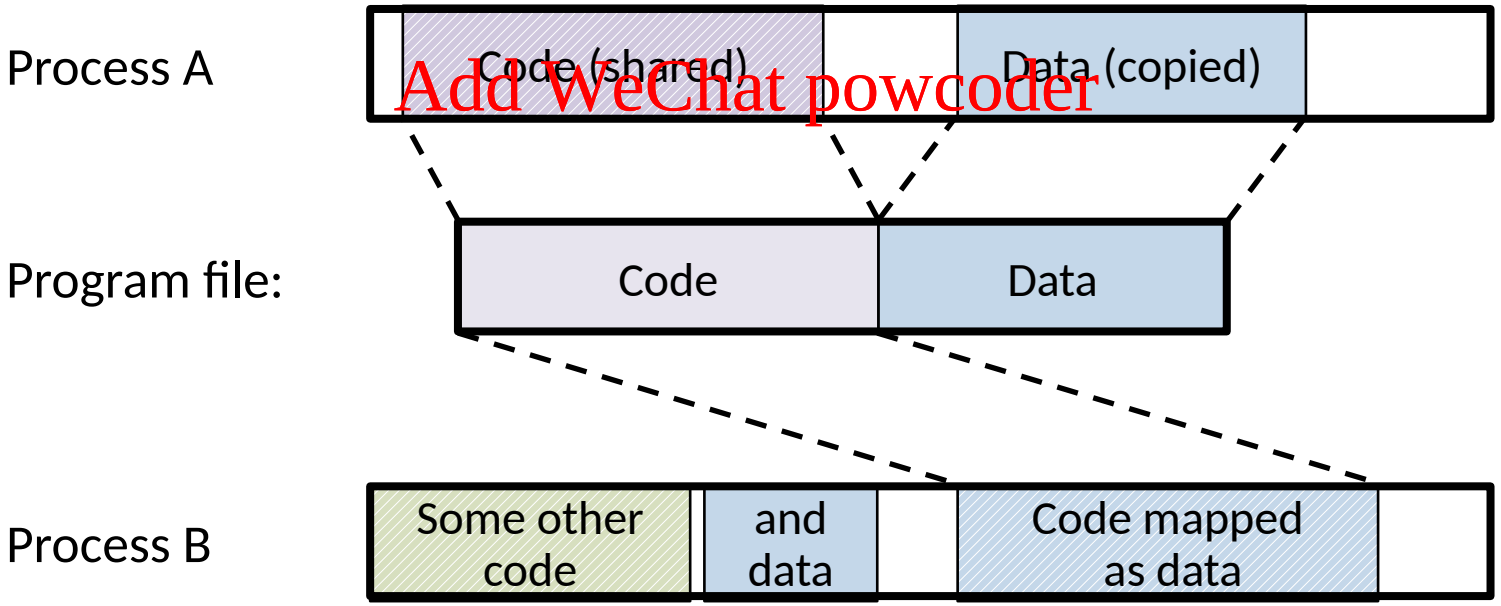
Code Sharing

- Recall that programs that run the same executable can share the code



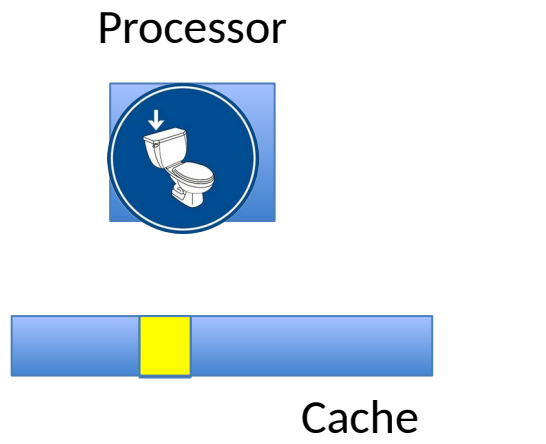
Code is Data

- In Von Neumann architectures code is a type of data



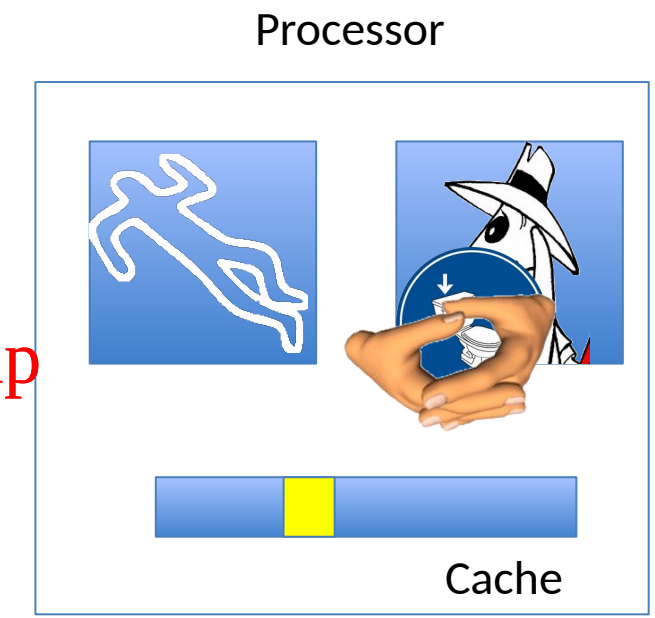
Cache Consistency

- Memory and cache can be in inconsistent states
  - Rare, but possible
- Solution: Flushing the cache contents
  - Ensures that the next load is served from the memory



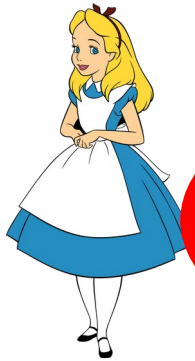
FLUSH+RELOAD

- **FLUSH** memory line
- Wait a bit
- Measure time to **RELOAD** line
  - slow-> no access
  - fast-> access
- Repeat



The RSA Encryption System

- The RSA encryption is a public key cryptographic scheme



$M = C^d \bmod N$

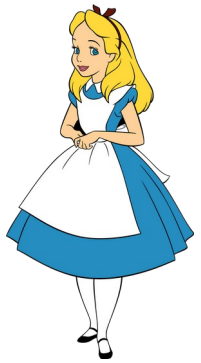
M

$C = M^e \bmod N$

Key Generation:

- Select random primes p and q
- Calculate $N = pq$
- Select a public exponent e(=65537)
- Compute $d=e^{-1} \bmod \varphi(N)$
- (N, e) is the public key
- (p, q, d) is the private key

Schnorr Signatures



$(A, \alpha) = \text{keypair}()$

A →

$R = g^r \bmod p$

$(R, r) = \text{keypair}()$

← M

$e = \text{Hash}(R, M)$

e →

$s = r - e\alpha$

s →

$R = g^s \cdot A^e \pmod p$

$e = ?\text{Hash}(R, M)$

GnuPG 1.4.13 Exponentiation

```
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
```

Example:

$11^5 \bmod 100 =$

$161{,}051 \bmod 100 = 51$

Operation	x	i	d_i
Multiply			
Square			
Reduce			
Square			
Reduce			
Multiply			

The private key is encoded in the sequence of operations !!!

Flush+Reload on GnuPG 1.4.13

FR vs. PP

- Flush+Reload tends to be more accurate
- Prime+Probe has less prerequisites

Variants

- Prime+Probe
  - Instruction cache
  - Last-level cache
  - TLB, BPU
  - Prime+Abort
- Flush+Reload
  - Flush-Flush
  - Evict+Reload
- Evict+Time
- CacheBleed
- Open DRAM rows
- Prefetch Channel

Countermeasures– System Level

- Avoid sharing hardware
  - Goes against modern software deployment trends
- Safe hardware implementations
  - Limited applicability
- Hardware partitioning
  - Partial support (if any)
- State sanitisation
  - Partial support (if any)
- Hardware randomisation
  - Not currently supported
- Clock randomisation
  - Ineffective

Software Countermeasures

- Preloading
  - Read all of the AES tables prior to decryption
    - Ineffective against asynchronous adversaries
- AES S-table implementation
  - A single table of size 256 bytes
    - Reduces chance of missing a cache line

GnuPG 1.4.14 Square and Multiply Always

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

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

Constant-Time Programming

- A programming style that avoids:
  - Instructions whose timing depends on secret data
  - Conditional execution based on secret data
  - Memory access to addresses that depend on secret data

Eliminating Conditional Statements

```
if (condition)
    t = f1()
else
    t = f2()
```

```
t1 = f1()
t2 = f2()
t = select(t1, t2, condition)
```

Implementing select

- Case 1: condition evaluates to 0 or 1

```
mask = condition - 1
return (t1 & ~mask) | (t2 & mask)
```


- Case 2: condition (`c`) evaluates to 0 or non-0


```
mask = ((c ^ (c-1)) & ~c)>>32
```



```
return (t1 & ~mask) | (t2 & mask)
```

Caveats

- The result of select depends on secret data. Anything that depends on it also depends on secret data.
  - In particular, swapping pointers using select does not produce constant-time code
- The choices of processor, languages and compiler matter
  - In most processors, division is not constant-time
  - In some processors multiplication is not constant-time
  - Compiler optimisations may kill constant-time code
  - **These issues have been exploited**