Side Cha s

Assignment Project Exam Help **Key**

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

ciphertext

plaintext

Assignment Project Exam Help **Key**

https://eduassistpro.github.io/

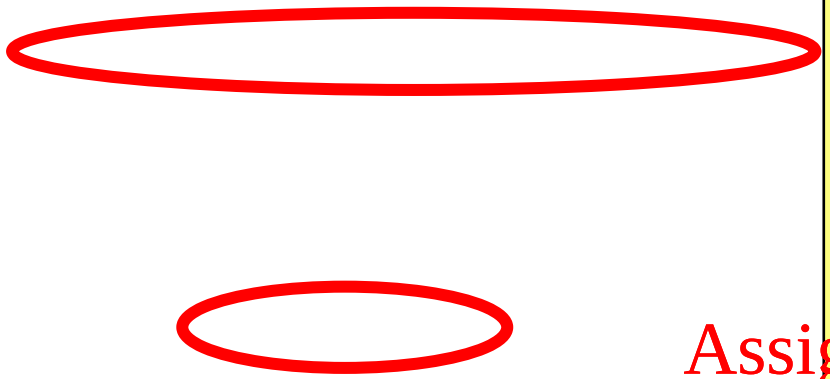
Add WeChat edu_assist_pro

plaintext

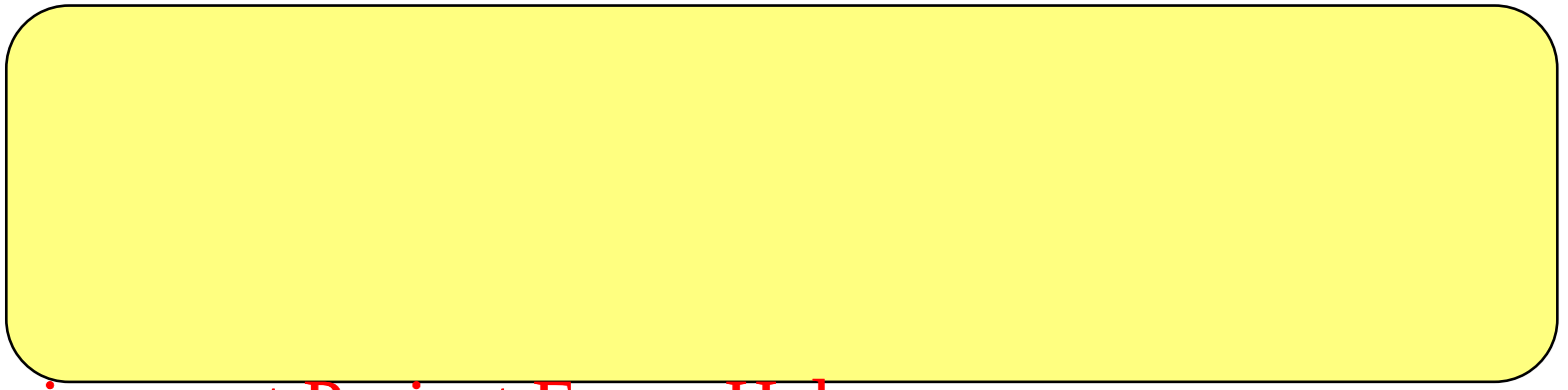
ciphertext

AES Implementation

AES Implementation



AES T-table access



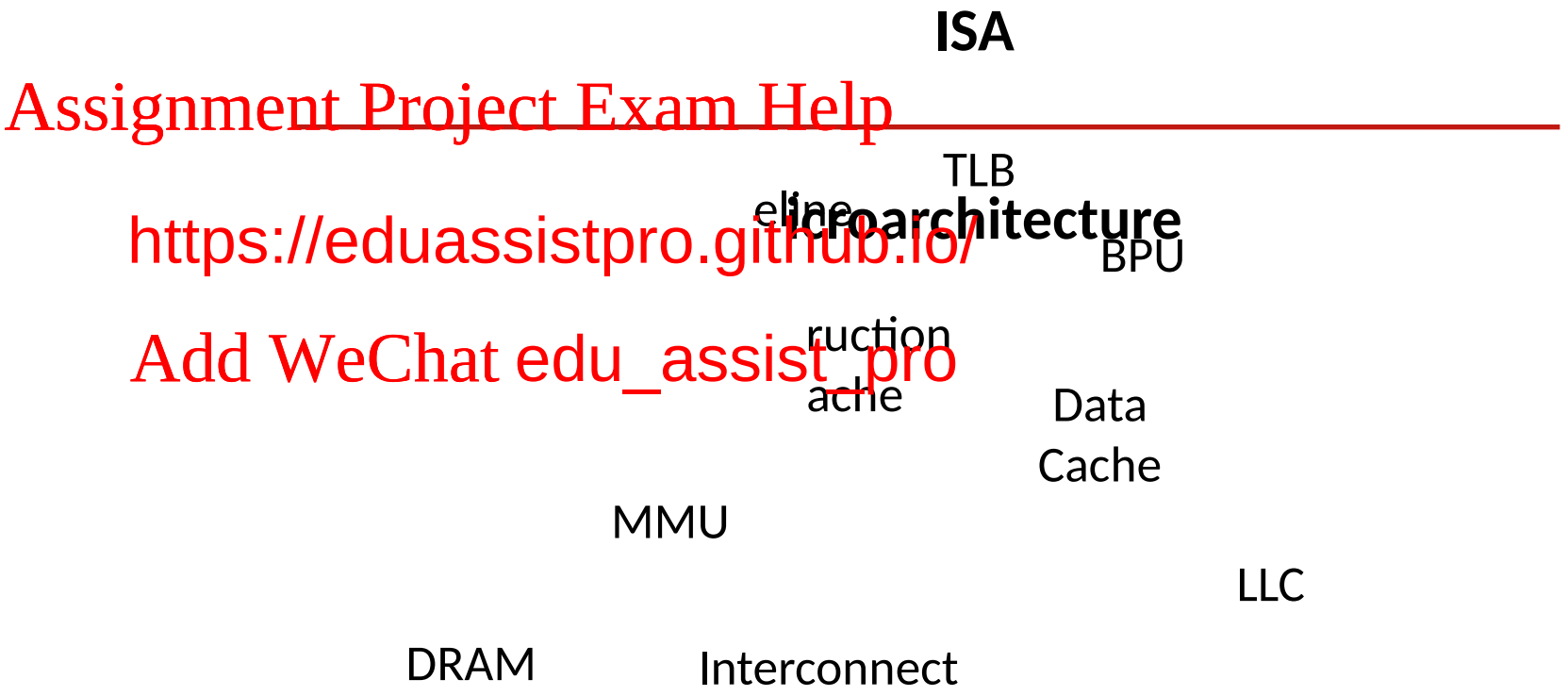

```
s0 = plaint
t0 = Te0[s0>>24]
```


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

AES Implementation

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

The Microarchitecture



CPU vs. Memory

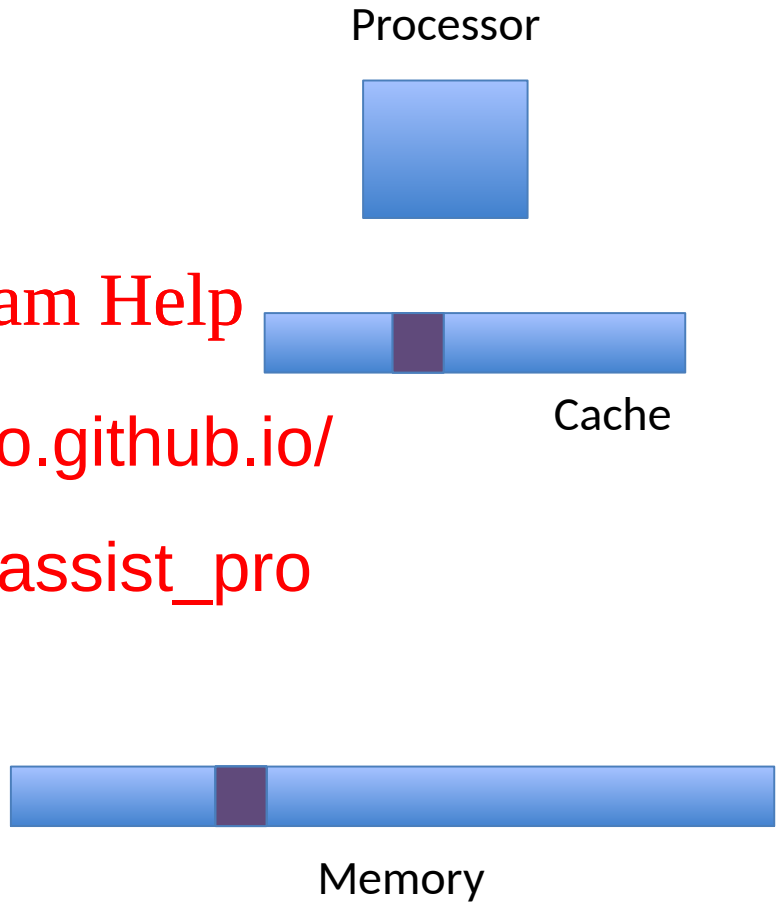
	Processor Speed	Memory Latency
		500 ns
	8*2600 MHz	63 ns

Bridging the gap

Cache utilises locality to bridge the gap

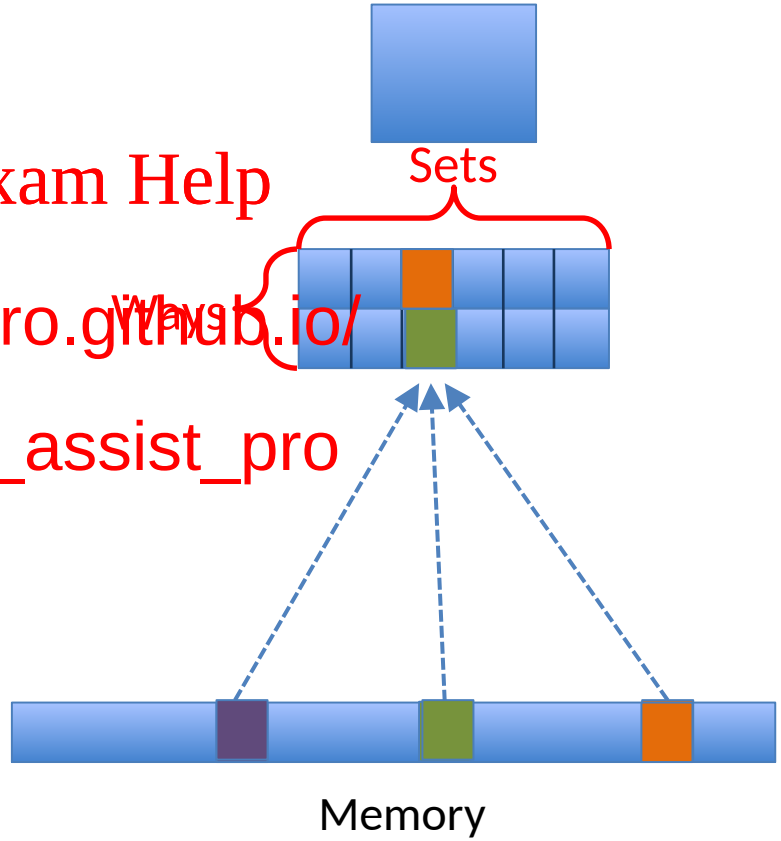
- Divides memory into *lines*
- Stores recently used l

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



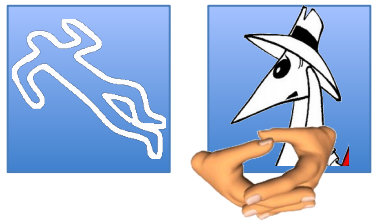
Set Associative Caches

- Memory lines map to *cache sets*. Multiple lines map to the same set.
- Sets consist of *ways*. A memory line can be st **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 ti access each cache set
  - Slow access indicates victim access to the set
- The probe phase primes the cache for the next round

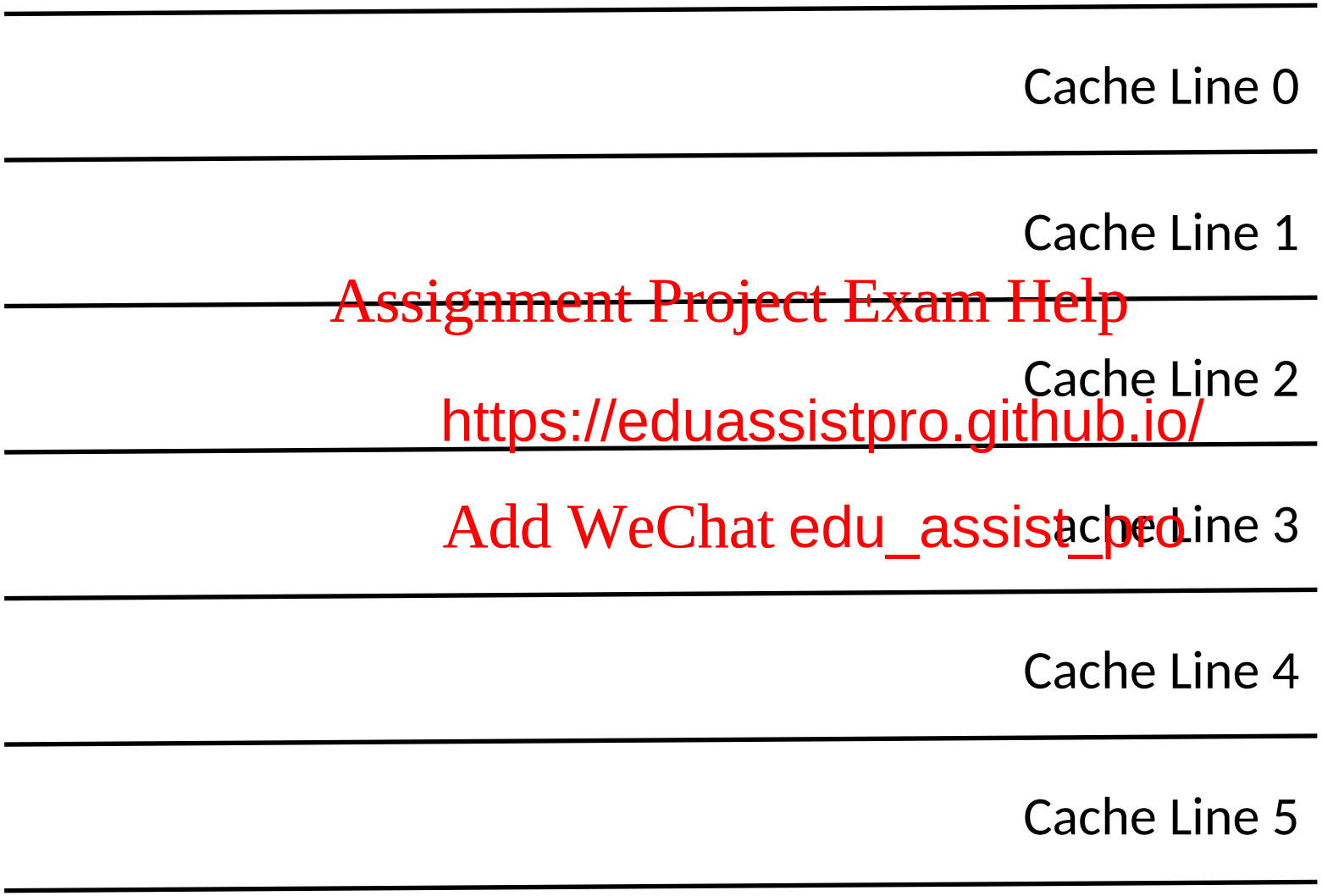


Memory

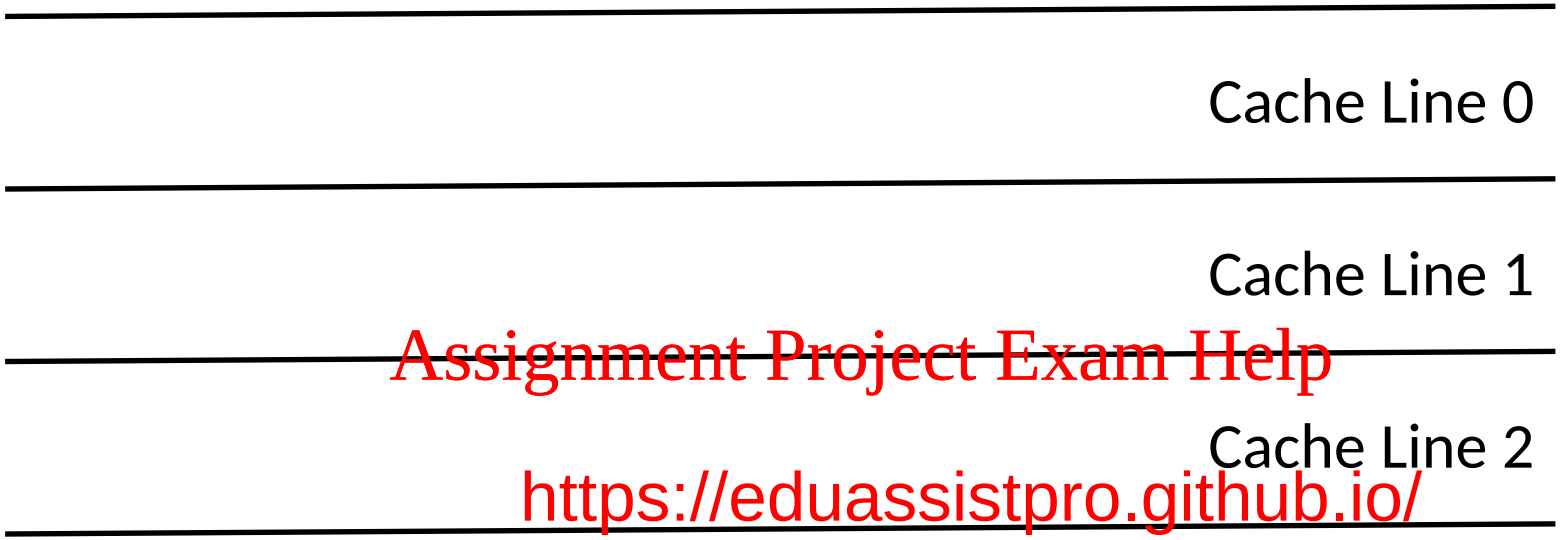
Sample Victim: Data Rattle

Cache Fingerprint of the Rattle Program

AES T-tables and cache lines



AES T-tables and cache lines



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

Analysing the AES Implementation

Analysing the AES Implementation

- Each round, Te0 is accessed 4 times
- AES has 10 rounds
  - Te0 is accessed 40 times in an AES encryption
- 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 ess time for each par
  - Identify the most significant half of the corresponding key byte

PP Attack on AES - Results

PP Attack on AES – More Results

What's now?

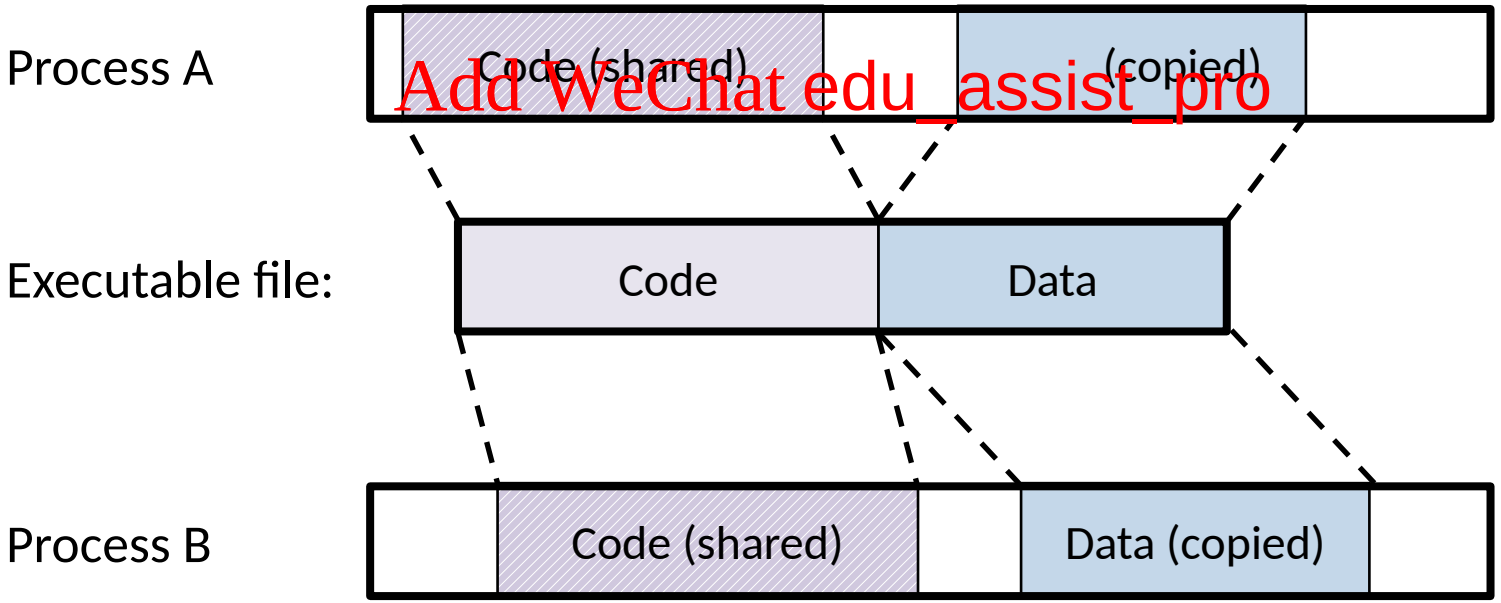
- Recover the second half of the key
  - Second round attack – similar but with ugly maths
- How to perform the attack
  - Easy: use Mastik:
    http://cs.ad
- How to defend?
  - Later…

The FLUSH+RELOAD Technique

- Leaks information on victim access to shared memory.
- Spy monitors victim's de
  - Spy can determine wh
  - Spy can infer the data the victim ope

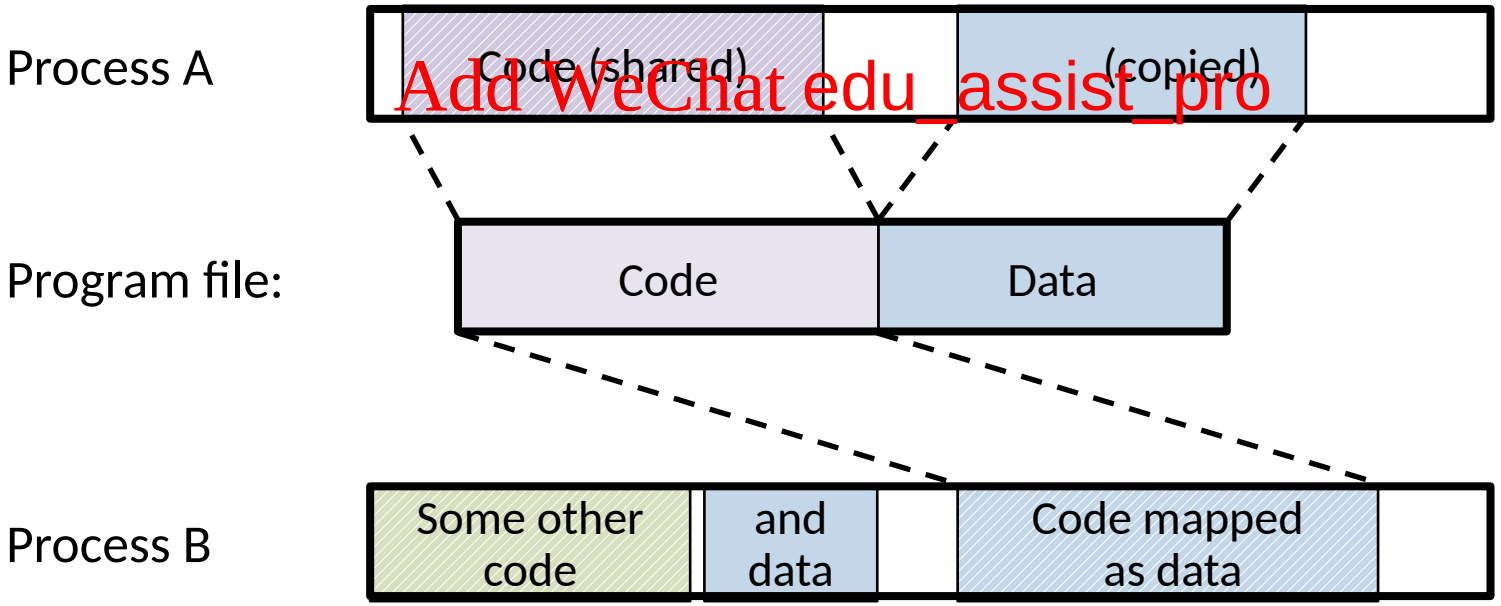
Code Sharing

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



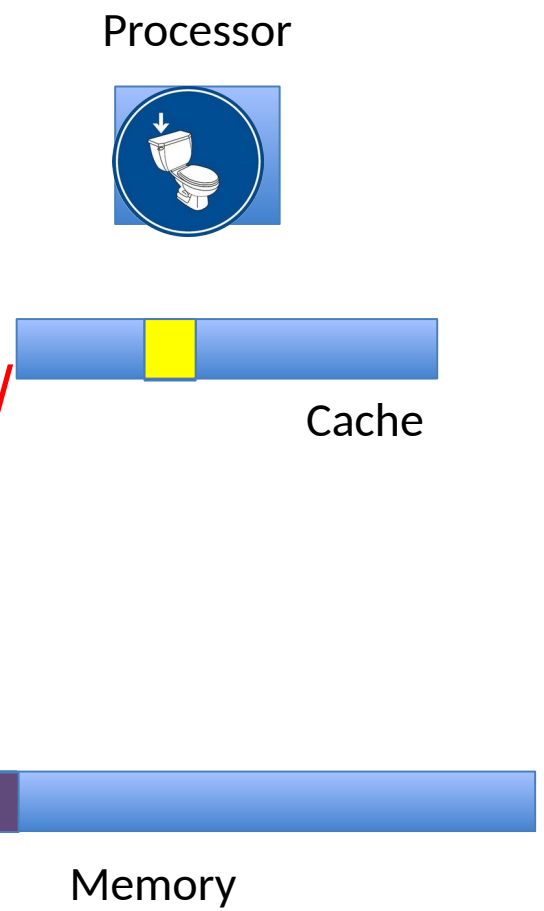
Code is Data

- In Von Neumann architectures code is a type of data



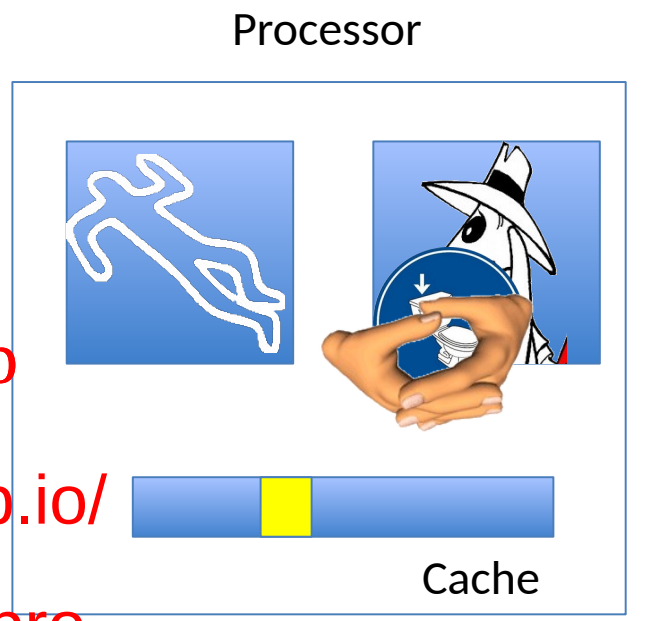
Cache Consistency

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



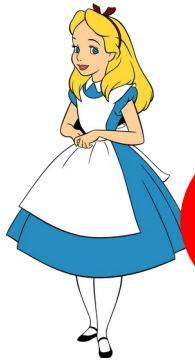
FLUSH+RELOAD

- **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



$M = C^d \bmod N$

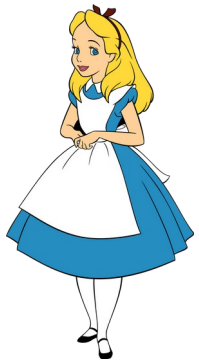
M

$= M^e \bmod N$

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
square			

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
- etch 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 implem
  - A single table of size
    - Reduces chance of missing a cache

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**
  $x \leftarrow x^2 \bmod n$
  $C \bmod n$
  $1)$ **then**
  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 a d on secret data

Eliminating Conditional Statements

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

```
t1 = f1()
2()
ct(t1, t2, condition)
```

Implementing select

- Case 1: condition evaluates to 0 or 1

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

- Case 2: condition (r non-0

```
mask = ((c ^ (
return (t1 & ~mask) |      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 proc and compiler matter
  - In most processors, division is n nt-time
  - In some processors multiplication is not constant-time
  - Compiler optimisations may kill constant-time code
  - **These issues have been exploited**