Security and Cryptography 2022

IV. Cache Attacks

Mirosław Kutyłowski

Cache attacks against a process

- side channel attack via measuring time
- same mechanism as for Meltdown: detecting cache misses indicates some particular execution pattern

Example: "Cache Missing for fun and profit" by Colin Percival

goal: find the RSA private key from OpenSSL executed on Pentium4 (original attack)

L 1

practical issues about cache:

- if there is a victim thread and a spy thread, then in the time between switching victim to spy the whole L1 can be evicted anyway as it is small
- L1: is very fast, time differences between a hit and miss and fetching from L2 are not big, problematic time measuring with rdtsc by the spy thread
- instructions are not loaded into L1 as to L2, no noise of this kind in L1
- problems with hardware prefetcher: if a few cache misses occurs on subsequent addresses then a few cache line fetched "just for the case" – so the spy process inspecting cache cannot ignore it
- TLB **(translation lookaside buffer)** influences miss time as well, TLB does not cover whole L2

OpenSSL RSA implementation

- Chinese Remainder Theorem (CRT) used:
  - instead of computing $a^d \bmod n$, where $n = p \cdot q$
  - one computes $a^d \bmod p$ and $a^d \bmod q$ and combines the results according to CRT
  - so: substantial speedup due to operations on smaller numbers
- sliding window exponentiation method
  - precomputed values: $a^3, a^5, \ldots, a^{31} \bmod p$
  - "square and multiply" method: a series of squarings $x := x^2 \bmod p,$ and multiplications $x := x \cdot a^{2k+1}$
  - squaring and multiplication use different algorithms with different "footprints" left in the cache
  - footprint also indicates approximately k from $x := x \cdot a^{2k+1}$

a^d : $a^1, a^2, a^3, \ldots\ a^{31}$

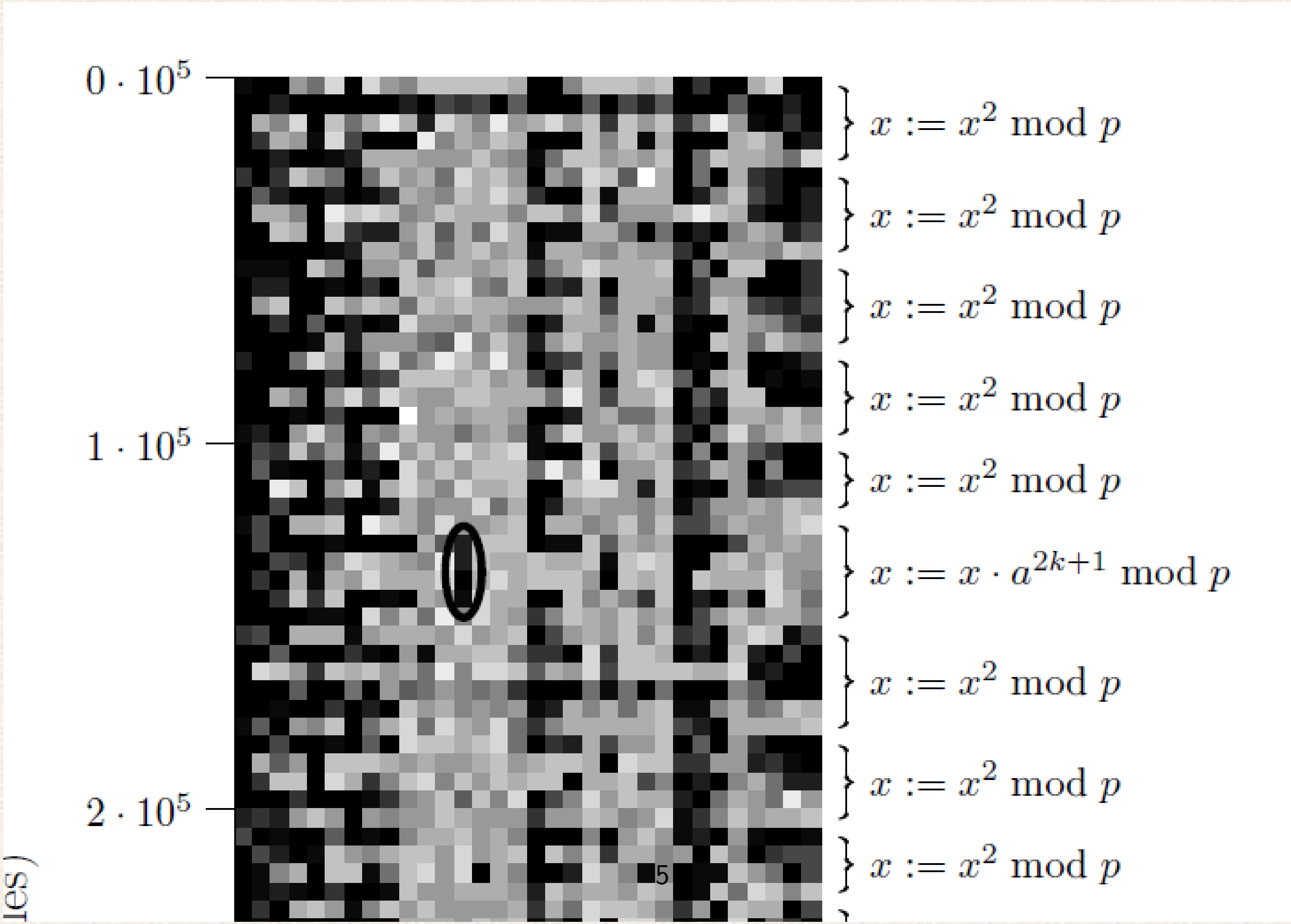
5 bit

$d = \boxed{17}\boxed{5}\boxed{}\boxed{}\boxed{3}\boxed{17}$

$(((((a^{17})^2)^2)^2)^2)^2 \quad a^{\boxed{17}\boxed{5}}$

$a^{\boxed{17}\boxed{0}}$

$\boxed{} \cdot a^5$



$0 \cdot 10^5$
$1 \cdot 10^5$
$2 \cdot 10^5$
$x := x^2 \bmod p$
$x := x^2 \bmod p$
$x := x^2 \bmod p$
$x := x^2 \bmod p$
$x := x^2 \bmod p$
$x := x \cdot a^{2k+1} \bmod p$
$x := x^2 \bmod p$
$x := x^2 \bmod p$
$x := x^2 \bmod p$

Austria: → CRT

Remark

- libraries often guard against such problems – no subroutines with variable time
- .. but frequently not the case:
  - example: the public key not stored but only encrypted secret key, then public key is recomputed (ECDSA)
  - computation must be based on exponentiation, where the exponent is the secret
  - so: a potential point of leakage via cache timings if sliding window used

Secure processing in a Data Center

- multiprocess architectures, with strict separation between processes offered by the system: hypervisor and virtualization, sandboxing, ...
- an attacker process tries to get secrets from victim processes without having any priviledges
  - theoretically virtualization solves the problem
- despite separation protection the processes share cache
- there is a strict control over the cache content but **cache hits and cache misses** might be detected by **timing for the attacker's process** (and not of the victim process)
- the timing for cache access should somehow depend on the sensitive information to be retreived
- difficulty: other than in the classical cryptanalysis – access to plaintext or ciphertext might be impossible (they belong to the victim process) - the attacker can only guess something

CASE STUDY: AES encryption

AES software implementation:

- particularly vulnerable because of its design
- AES defined in algebraic terms, but lookup table is typically faster
- there are arguments against algebraic implementations as the execution time may provide a side channel
- key expansion: round zero: simply the key bytes directly, other rounds: key expansion reversable (details irrelevant for the attack)
- fast implementation based on lookup tables T_0, T_1, T_2, T_3 and $T_0^{(10)}, T_1^{(10)}, T_2^{(10)}, T_3^{(10)}$ for the last round (with no `MixColumns`)

LUT

T_0

- round operation

$$\left(x_0^{(r+1)}, x_1^{(r+1)}, x_2^{(r+1)}, x_3^{(r+1)}\right) := T_0(x_0^r) \oplus T_1(x_5^r) \oplus T_2(x_{10}^r) \oplus T_3(x_{15}^r) \oplus K_0^{(r+1)}$$

$$\left(x_4^{(r+1)}, x_5^{(r+1)}, x_6^{(r+1)}, x_7^{(r+1)}\right) := T_0(x_4^r) \oplus T_1(x_9^r) \oplus T_2(x_{14}^r) \oplus T_3(x_3^r) \oplus K_1^{(r+1)}$$

$$\left(x_8^{(r+1)}, x_9^{(r+1)}, x_{10}^{(r+1)}, x_{11}^{(r+1)}\right) := T_0(x_8^r) \oplus T_1(x_{13}^r) \oplus T_2(x_2^r) \oplus T_3(x_7^r) \oplus K_2^{(r+1)}$$

$$\left(x_{12}^{(r+1)}, x_{13}^{(r+1)}, x_{14}^{(r+1)}, x_{15}^{(r+1)}\right) := T_0(x_{12}^r) \oplus T_1(x_1^r) \oplus T_2(x_6^r) \oplus T_3(x_{11}^r) \oplus K_3^{(r+1)}$$

4096 / 4·32 = 16

attack notation:

– $\delta = B/\text{entrysize of lookup table}$, typically: entrysize=4bytes, $\delta = 16$, (so δ entries of a lookup table are within the same cache line – this is a complication for the attack!)

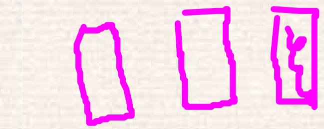
– for a byte y let $\langle y \rangle = \lfloor y/\delta \rfloor$, it indicates a memory block of y in T_l

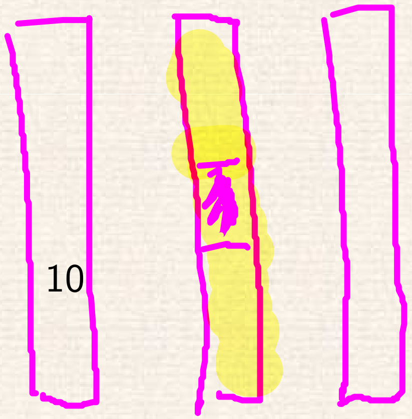
– if $\langle y \rangle = \langle z \rangle$, then x and y correspond to requests to the same memory block of the lookup table and therefore to the same cache line

– $\boxed{Q_K(p,l,y)} = 1$ iff AES encryption of plaintext p under key K accesses memory block of index y in T_l at least once in 10 rounds

– $\boxed{M_k(p,l,y)} =$ measurement, its expected value is bigger when $Q_k(p,l,y) = 1$ than in case $Q_k(p,l,y) = 0$

cache





“synchronous attack”

– **plaintext random but known**, corresponds to the situation where one can trigger encryption (e.g. VPN with unknown key, dm-crypt of Linux)

– phase 1: measurements, phase 2: analysis

– from experiments: AES key recovered using 65 ms of measurements (800 writes) and 3 sec analysis

attack on round 1:

i. accessed indices for lookup tables are simply $x_i^{(0)} = p_i \oplus k_i$ for $i = 0, \ldots, 15$

ii. goal: find information $\langle k_i \rangle$ of k_i – one cannot derive information on lsb; candidates for k_i are denoted by $\bar{k_i}$

iii. if $\langle k_i \rangle = \langle \bar{k_i} \rangle$ and $\langle y \rangle = \langle p_i \oplus \bar{k_i} \rangle$, then $Q_K(p, l, y) = 1$ for the lookup $T_l(x_i^{(0)})$

iv. if $\langle k_i \rangle \neq \langle \bar{k_i} \rangle$, then there is no lookup in block y for T_l during the 1st round, **but**

– there are $4 \cdot 9 - 1 =$ 35 other accesses affected by other plaintext bits during the entire encryption (4 per round, 9 rounds in total as the last round uses different look-up tables)

– probability that none of them accesses block y for T_l is

$$\left(1 - \frac{\delta}{256}\right)^{35} \approx 0.104 \text{ for } \delta = 16$$

$$\approx \left(1 - \frac{1}{16}\right)^{16 \cdot 2} \approx \frac{1}{e^2}$$

v. few dozens of samples required to find a right candidate for $\langle k_i \rangle$

vi. together we determine $\log(256/\delta) = 4$ bits of each byte of the key

vii. nothing more possible for the 1st round, still 64 key bits to be found, too much for brute force

viii. in reality more samples needed due to noise in measurements $M_K(p, l, y)$

$k_0 = k_5 \oplus \Delta$ $p_0 = p_5 \oplus \Delta$

attack on round 2: the goal is to find the still unknown key bits

i. we exploit equations derived from the Rijndeal specification:

$$x_2^{(1)} = s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5) \oplus 2 \bullet s(p_{10} \oplus k_{10}) \oplus 3 \bullet s(p_{15} \oplus k_{15}) \oplus s(k_{15}) \oplus k_2$$

$$x_5^{(1)} = s(p_4 \oplus k_4) \oplus 2 \bullet s(p_9 \oplus k_9) \oplus 3 \bullet s(p_{14} \oplus k_{14}) \oplus s(p_3 \oplus k_3) \oplus s(k_{14}) \oplus k_1 \oplus k_5$$

...

$$x_8^{(1)} = ...$$

...

$$x_{15}^{(1)} = ...$$

where s stands for the Rijndael Sbox, and $\bullet$ means multiplication in the field with 256 elements

ii. lookup for $T_2(x_2^{(1)})$:

$$x_2^{(1)} = s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5) \oplus 2\bullet s(p_{10} \oplus k_{10}) \oplus 3\bullet s(p_{15} \oplus k_{15}) \oplus s(k_{15}) \oplus k_2$$

– $\langle k_0 \rangle, \langle k_5 \rangle, \langle k_{10} \rangle, \langle k_{15} \rangle, \langle k_2 \rangle$ already known

– low level bits of $\langle k_2 \rangle$ influence only low bits of $x_2^{(1)}$ so not important for cache access pattern

– the upper bits of $x_2^{(1)}$ can be determined after guessing low bits of k_0, k_5, k_{10}, k_{15}: there are δ^4 possibilities ($=16^4$)

– a correct guess yields a lookup in the right place

– an incorrect guess: some $k_i \neq \bar{k_i}$ so

$$x_2^{(1)} \oplus \bar{x}_2^{(1)} = c_i\bullet s(p_i \oplus k_i) \oplus c_i\bullet s(p_i \oplus \bar{k_i}) \oplus \ldots$$

where ... depends on different random plaintext bits and therefore random

- differential properties of AES studied for AES competition:

$$\Pr\left[\, c_i \bullet s(p_i \oplus k_i) \oplus c_i \bullet s(p_i \oplus \bar{k_i}) \neq z \right] > \left(1 - \frac{\delta}{256}\right)^3$$

  so the false positive for lookup in T_2 at a given block:

  - $\left(1 - \frac{\delta}{256}\right)^3$ for not computing $T_2\left(x_2^{(1)}\right)$
  - $\left(1 - \frac{\delta}{256}\right)$ for not refering to the same cache line as $T_2\left(x_2^{(1)}\right)$ while computing each of the remaining invocations of T_2
  - together no access with pbb about $\left(1 - \frac{\delta}{256}\right)^{38}$
- this yields about 2056 samples necessary to eliminate all wrong candidates
- it must be repeated 3 more times to get other nibbles of key bytes

iii. optimization: guess $\Delta = k_i \oplus k_j$ and take $p_i \oplus p_j = \Delta$, then i.e. $s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5)$ cancels out and we have to guess less bits (4 instead of 8)

– **similar attack: last round** - created ciphertext must be known to the attacker, otherwise similar. Subkey from the last round learnt, but key schedule is reversible

cache measurement strategy: Evict+Time

i. procedure:

   1. trigger encryption of a plaintext p
   2. **evict:** access memory addresses so that **one cache set overwritten** completely
   3. trigger encryption of the plaintext p

ii. in the evicted cache set one cache line from T_l **is missing**

iii. measure time: if long, then cache miss and the encryption refers to evicted δ positions from the lookup table

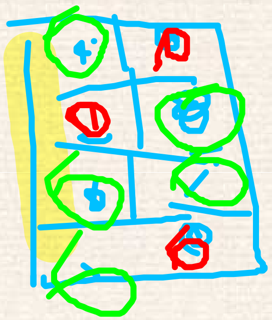
iv. practical problem: triggering may invoke other activities and timing is not precise

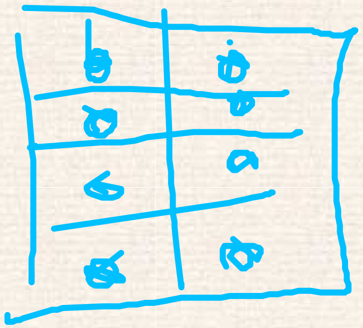
- **measurement: Prime+Probe**

  i. procedure

  1. **prime:** overwrite entire cache by reading A: a contiguous memory of the size of the cache
  2. trigger an encryption of p – it results in **eviction** at places where lookup has occurred
  3. **probe:** read memory addresses of A and **detect which locations have been evicted**

  ii. easier: probe timing checked, not the time at encryption





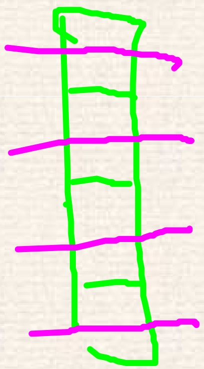
– **complications in practice:**

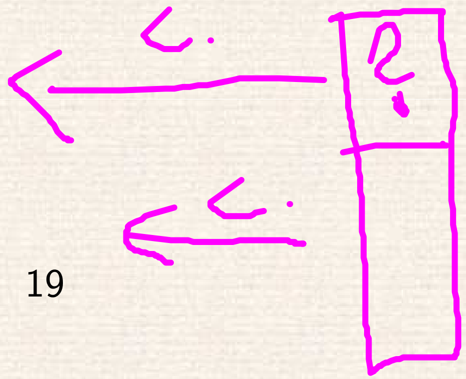
i. **address of lookup tables in the memory** - how they are loaded to the cache remains unknown – offset can be found by considering all offsets and then statistics for each offset (experiments show good results even in a noisy environment)

ii. **hardware prefetcher** may disturb the effects. Solution: read and write the addresses of A according to a pseudorandom permutation

– **practical experiments:** e.g. Athlon 64, no knowledge of adresses mapping, 8000 encryptions with Prime & Probe

Linux dm-crypt (disk, filesystem, file encryption): with knowledge of addressing, 800 encryptions (65 ms), 3 seconds analysis, full AES key





extensions of the attack:

- on some platforms timing shows also position of the cache line (better resolution for one-round attack)
- remote attacks (VPN, IPSec): via requests that trigger immediate response (unclear practicality)

“asynchronous attrack” on round 1

- no knowledge of plaintext, no knowledge of ciphertext
- based on frequency F of bytes in e.g. English texts, frequency score for each of $\frac{256}{\delta}$ blocks of length δ
- F is nonuniform: most bytes have high nibble = 6 (lowercase characters “a” through “o”)
- find j such that j is particularly frequent indicates $j = 6 \oplus \langle k_i \rangle$ and shows $\langle k_i \rangle$
- complication: this frequency concerns at the same time k_0, k_5, k_{10}, k_{15} affecting T_0 so we learn 4 nibbles but not their actual allocation to k_0, k_5, k_{10}, k_{15}
- the number of bits learnt is roughly: $4 \cdot (4 \cdot 4 - \log 4!) \approx 4 \cdot (16 - 3.17) \approx$ 51 bits
- experiment: OpenSSL, measurements 1 minute, 45.27 info bits o on the 128-bit key gathered

Bernstein's attack

– an alternative way of computing AES, algorithm applied in OpenSSL:

- → two constant 256-byte tables: S and S'
- → expanded to 1024-byte tables T_0, T_1, T_2, T_3

  $$T_0[b] = (S'[b], S[b], S[b], S[b] \oplus S'[b])$$

  $$T_1[b] = (S[b] \oplus S'[b], S'[b], S'[b], S[b], S[b])$$

  $....$
- → AES works with 16-byte arrrays x and y, where x initialized with the key k, y initialized with $p \oplus k$, where p is the plaintext

→ AES computation = modifications of x and y:

i. x viewed as (x_0, x_1, x_2, x_3) (4 bytes parts)

ii. $e := (S[x_3(1) \oplus 1], S[x_3(2)], S[x_3(3)], S[x_3(0)])$

iii. replace (x_0, x_1, x_2, x_3) with $(e \oplus x_0, e \oplus x_0 \oplus x_1, e \oplus x_0 \oplus x_1 \oplus x_2, e \oplus x_1 \oplus x_2 \oplus x_3)$

iv. replace $y = (y_0, y_1, y_2, y_3)$ with

$$(T_0[y_0[0]] \oplus T_1[y_1[1]] \oplus T_2[y_2[2]] \oplus T_3[y_3[3]] \oplus x_0,$$
$$(T_0[y_1[0]] \oplus T_1[y_2[1]] \oplus T_2[y_3[2]] \oplus T_3[y_0[3]] \oplus x_1,$$
$$(T_0[y_2[0]] \oplus T_1[y_3[1]] \oplus T_2[y_0[2]] \oplus T_3[y_1[3]] \oplus x_2,$$
$$(T_0[y_3[0]] \oplus T_1[y_0[1]] \oplus T_2[y_1[2]] \oplus T_3[y_2[3]] \oplus x_3$$

v. 2nd round uses $\oplus 2$ instead of $\oplus 1$ for x, otherwise the same. Similar changes corresponding to rounds up to 9

vi. in round 10 use $S[], S[], S[], S[]$ instead of $T's$

vii. y is the final output

it is embarassing how simple the attack is:

→ it has been checked in practice that execution depends on $k[i] \oplus p[i]$ - which is a position in the table:

- try many plaintexts p
- collect statistics for each byte for $p[13]$
- the maximum occurs for z
- the maximum corresponds to a fixed value for $k[13] \oplus p[13]$, say c
- compute $k[13] = c \oplus z$

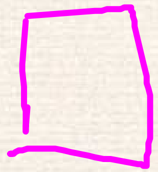
→ for different bytes different statistics observed: for some t a few values $k[t] \oplus \mathrm{plaintext}[t]$, where substantially higher time observed

→ statistic gathered, different packet lengths

→ finally brute force checking all possibilites

Countermeasures

- "**no reliable and practical countermeasure**" so far
- implementation based on **no-lookup**: instead algebraic algorithm (slow!!!) or bitslice implementation (sometimes possible and nearly as efficient as lookup)
- **alternative lookup tables:** if smaller, then smaller leakage (but easier cryptanalysis for small Sboxes)
- **data-independent access to memory** blocks - every lookup causes a redundant read in all memory blocks, generally: oblivious computation possible theoretically, but overhead makes it inattractive
- **masking operations:** ≈"we are not aware of any method that helps to resist our attack"
- **cache state normalization:** load all lookup tables - equires deep changes in OS and reduces efficiency, even then LRU cache policy may leak information which part has been used!
- **process blocking:** again, deep changes in OS
- **disable cache sharing**: deep degradation of performance



- **”no-fill” mode** during crypto operations:
  - preload lookup tables
  - activate “no-fill”
  - crypto operation
  - deactivate “no-fill”

  the first two steps are critical and no other process is allowed to run

  possible only in priviledged mode, cost of operation prohibitive
- **dynamic table storage:** e.g. many copies of each table, or permute tables

  details architecture dependent and might be costly
- **hiding timing information:** adding random values to timing makes the statistical analysis harder but still feasible
- **protect some rounds** (the first 2 and the last one) with any mean – but may be there are other attack techniques...

- **cryptographic services at system level**: good but not flexible
- **sensitive status for user processes:** erasing all data when interrupt
- **specialized hardware support:** crypto co-processor seems to be the best choice

but the problem is not limited to AES or crypto – many sensitive data operations are not cryptographic and a coprocessor does not help