Útoky na LFSR šifry

InstallFest 2016

Jan Hrach

http://jenda.hrach.eu/

PGP: CD98 5440 4372 0C6D 164D A24D F019 2F8E 6527 282E

Implementace

- Dokumentace, materiály, papery, zdrojáky…

  https://brmlab.cz/project/gsm/deka/start

GSM z rychlíku

- HLR a SIM v mobilu mají sdílené tajemství Ki
- Před komunikací si dohodnou session key Kc

- Ki se neumí pasivně crackovat
- Kc jo

Komunikace telefonu

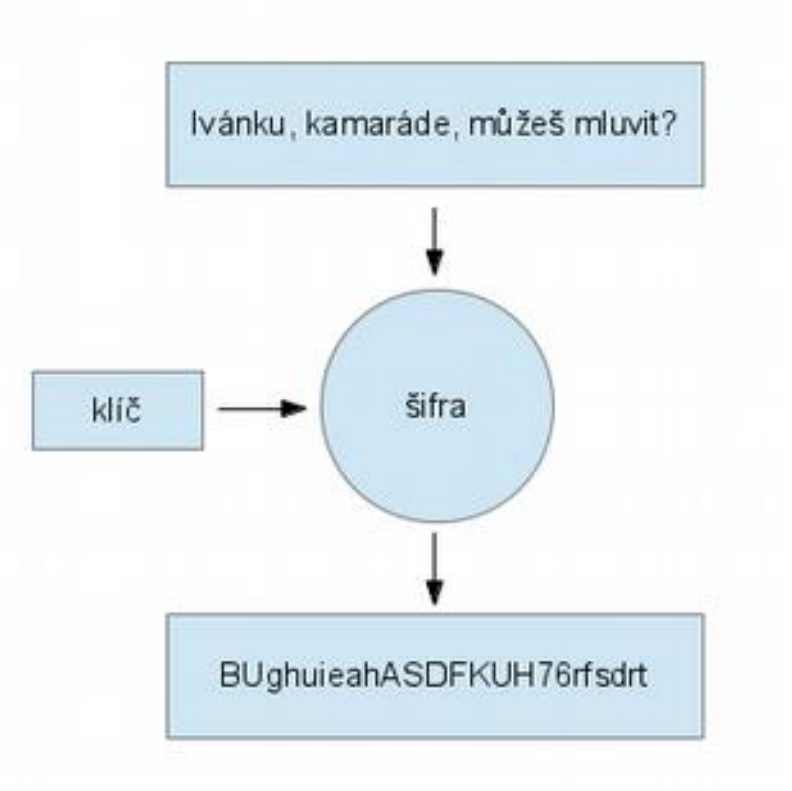
- První část nešifrovaná, nic moc zajímavého (občas IMSI a TA)
- Pak zapnou šifrování
- Pak nahráváme „nesmysly“
- ty chceme cracknout

Jak to sniffnout

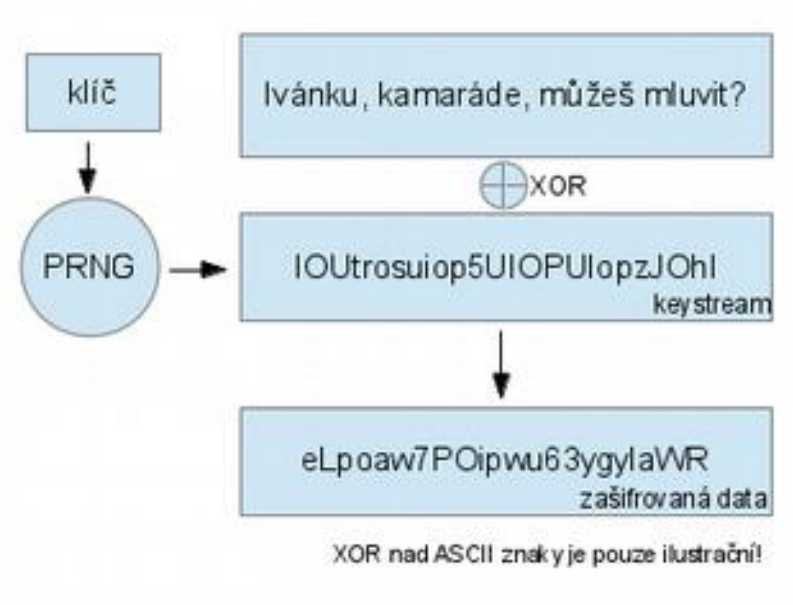
- Hacknutý mobil
  - https://brmlab.cz/user/jenda/gsm
- SDR
  - Airprobe
  - gr-gsm

…co s tím pak?

Šifry

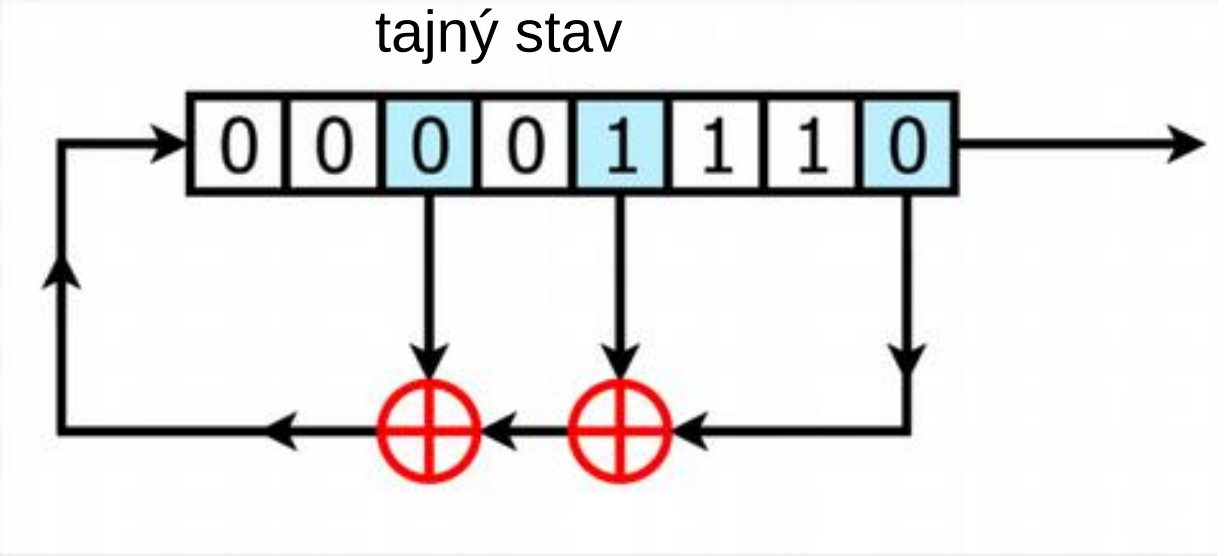


Bloková (AES, Blowfish, DES)

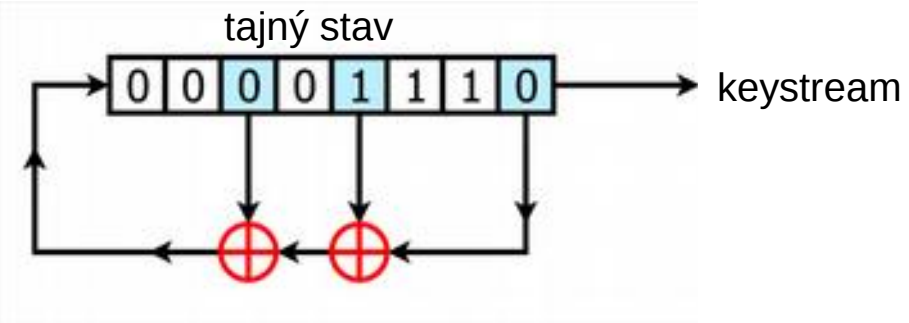


Proudová (RC4, **A5/1**)

Linear Feedback Shift Register

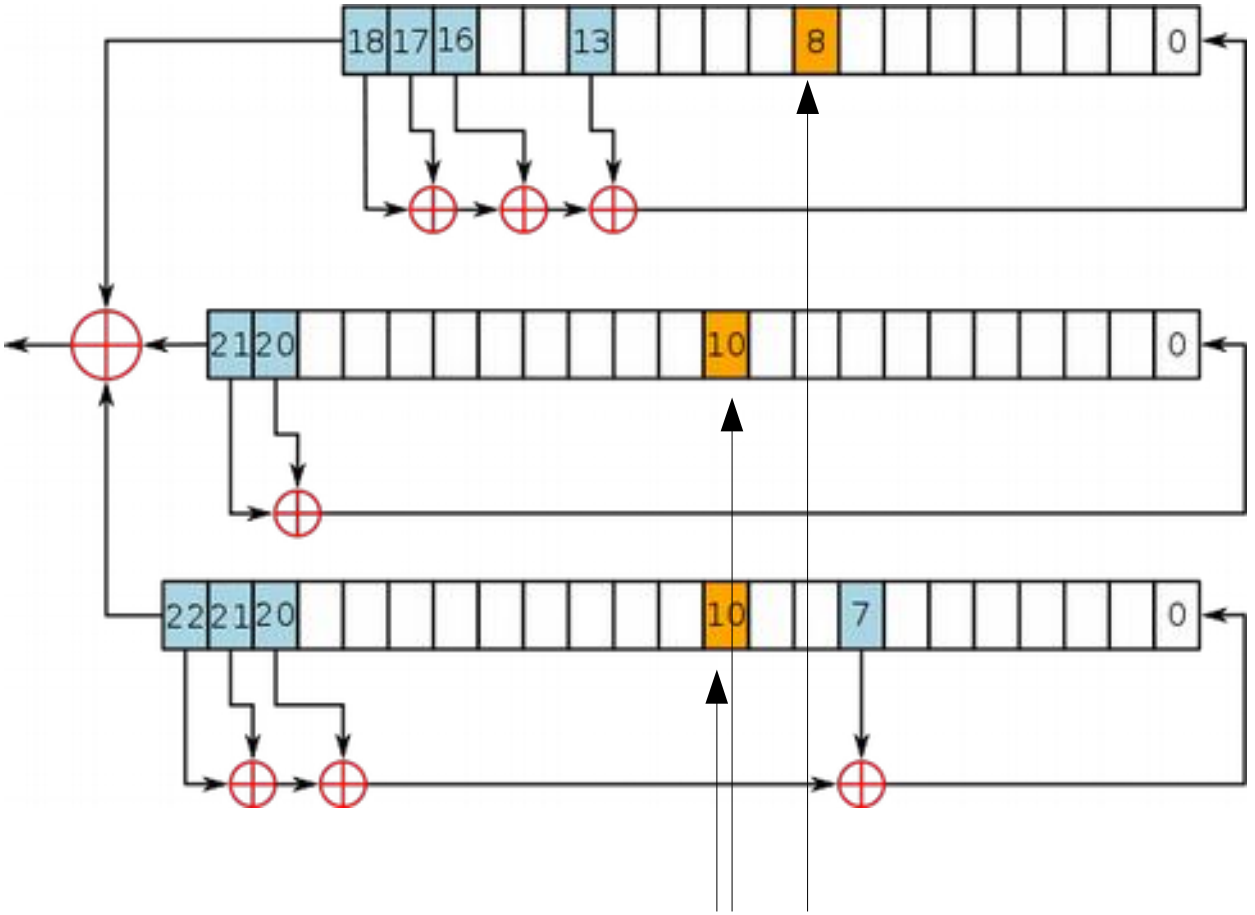


Lineární algebra (A5/2, A5-GMR)



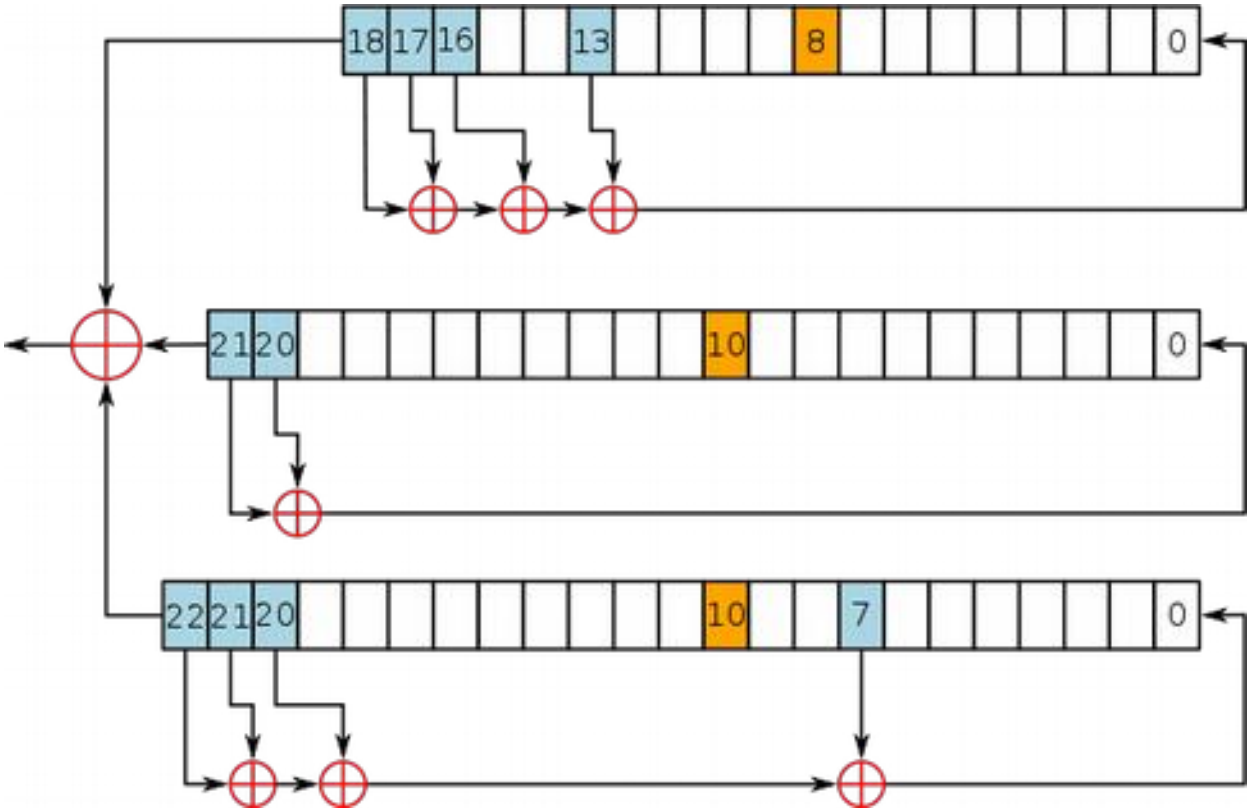
- XOR ~ sčítání mod 2
- Bitshift ~ násobení mocninami dvou
- Samoopravné kódy: XOR a shift, „pravá strana“ rovnice
  - → maticový a lingebra magic
  - → soustava lineárních rovnic
  - → umíme efektivně řešit
- http://www.npag.fr/project-a52hacktool
- https://www.youtube.com/watch?v=y15bXRsa3iA

A5/1



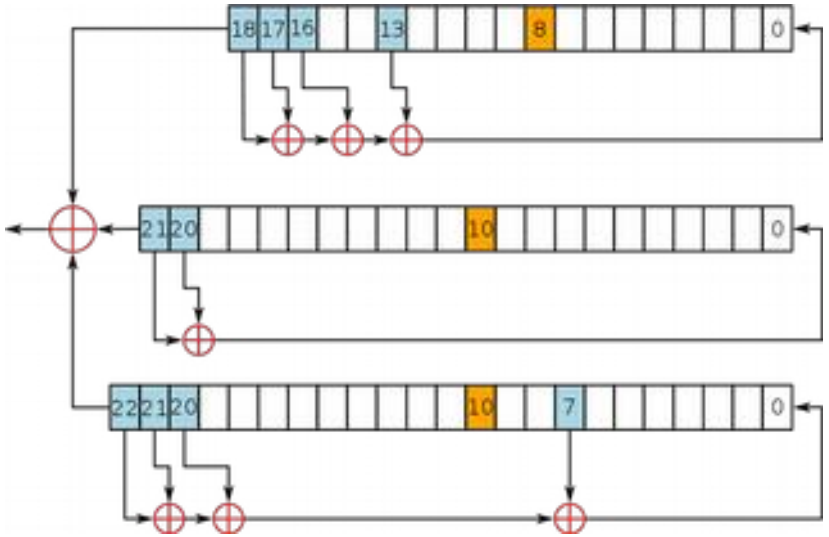
Majority clocking. Neumíme linearizovat.

A5/1



Jenom 64 bitů celkem!

Codebook



Internal state	Keystream
0x0000000000000000	0xabcdef12345678
0x0000000000000001	0x54632221afed03
0x0000000000000002	0x456dcd562b980e
...	
0xffffffffffffffff	0x002accd4dc51df

… 2^64 * 16B = 296 EB

1 Košík | **2** Dodání a platba | **3** Dodací údaje | **4** Souhrn objednávky

Všechny ceny jsou uváděny s DPH	počet	skladem	cena za kus	cena celkem
 156611 WD Green (EZRX) - 6TB	24595659	5+ kusů	6 582 Kč	161 895 489 727 Kč ×

Doplňkové služby

☐ **Doprava zdarma** za body — 2 990 b

Cena celkem bez DPH: **133 797 925 394 Kč**

Za tento nákup získáte **35 909 662 140 bodů** do Bonus klubu.

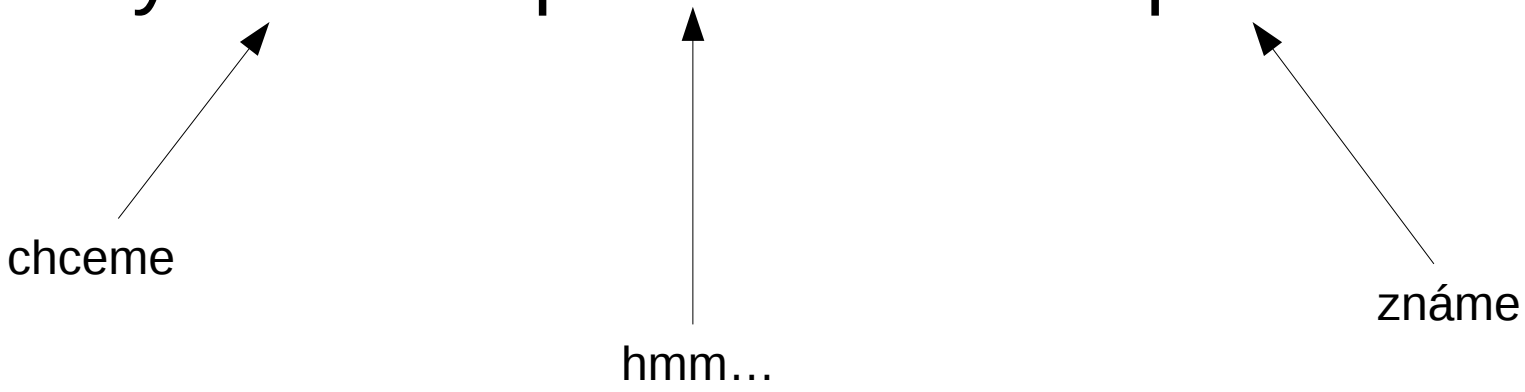
Cena celkem s DPH: **161 895 489 727 Kč**

Pokračovat ›

Btw. jak získat keystream?

- Ciphertext = plaintext XOR keystream

- Keystream = plaintext XOR ciphertext



BTS musí pořád vysílat
Definován padding 0x2B
Informace o okolních BTS, o síle signálu, …

Kolaps klíčového prostoru

- Do některých kombinací se nedá dostat
- Několika dummy cykly A5/1 tyto stavy eliminujeme
- Efektivní stavový prostor 2^61

~~… 2^64 * 16B = 296 EB~~

… 2^61 * 16B = 37 EB

Pokrytí

- GSM frame 4*114 bitů
- 64 bitů v šifře
- 4*(114-64+1) = 204 samplů
- Stačí pokrytí 1/204 = 0,5 % pro ~50% úspěšnost
- Framů je v komunikaci hodně

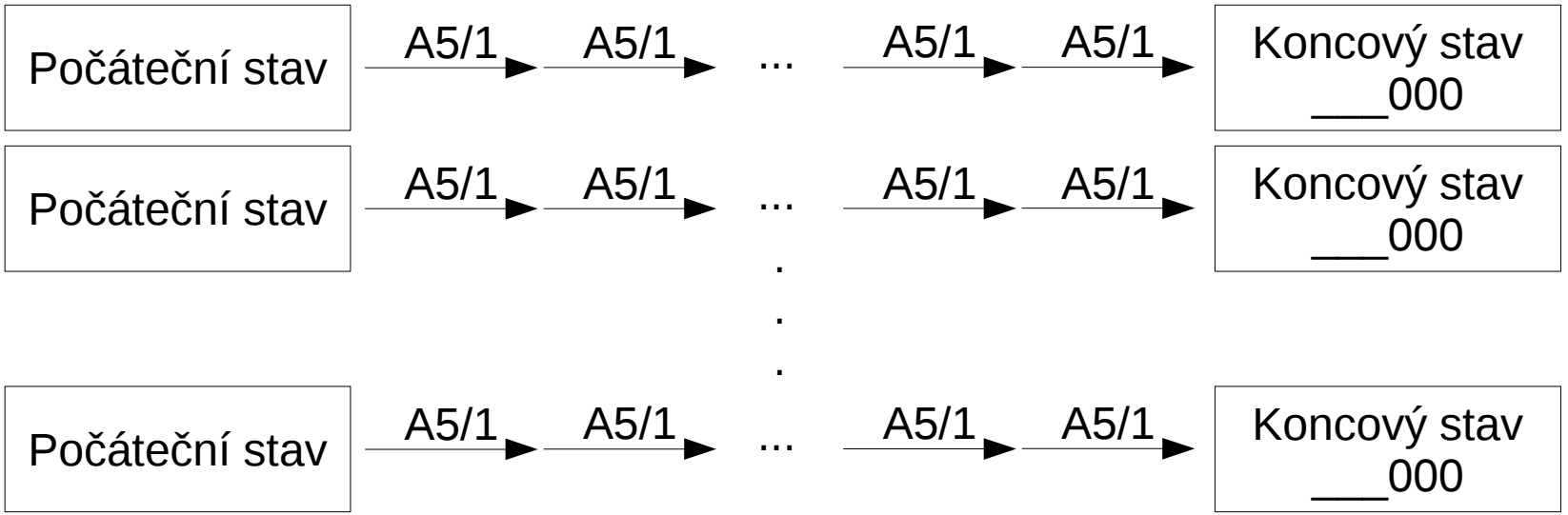
… ~~2^64 * 16B = 296 EB~~

… ~~2^61 * 16B = 37 EB~~

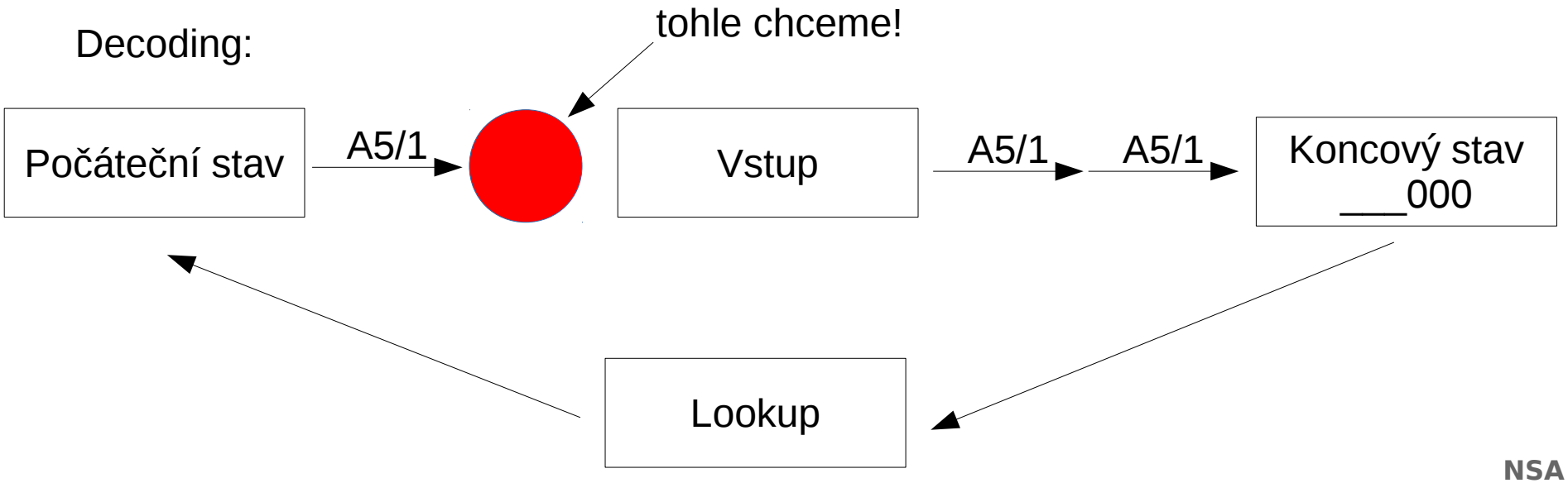
… 2^61 * 16B / 204 = 181 PB

Chainy

Encoding:



Decoding:



Chainy

- Koncový stav ~ 12 bitů nulových

  → průměrná délka chainu 2^12 = 4096

  ~~… 2^64 * 16B = 296 EB~~

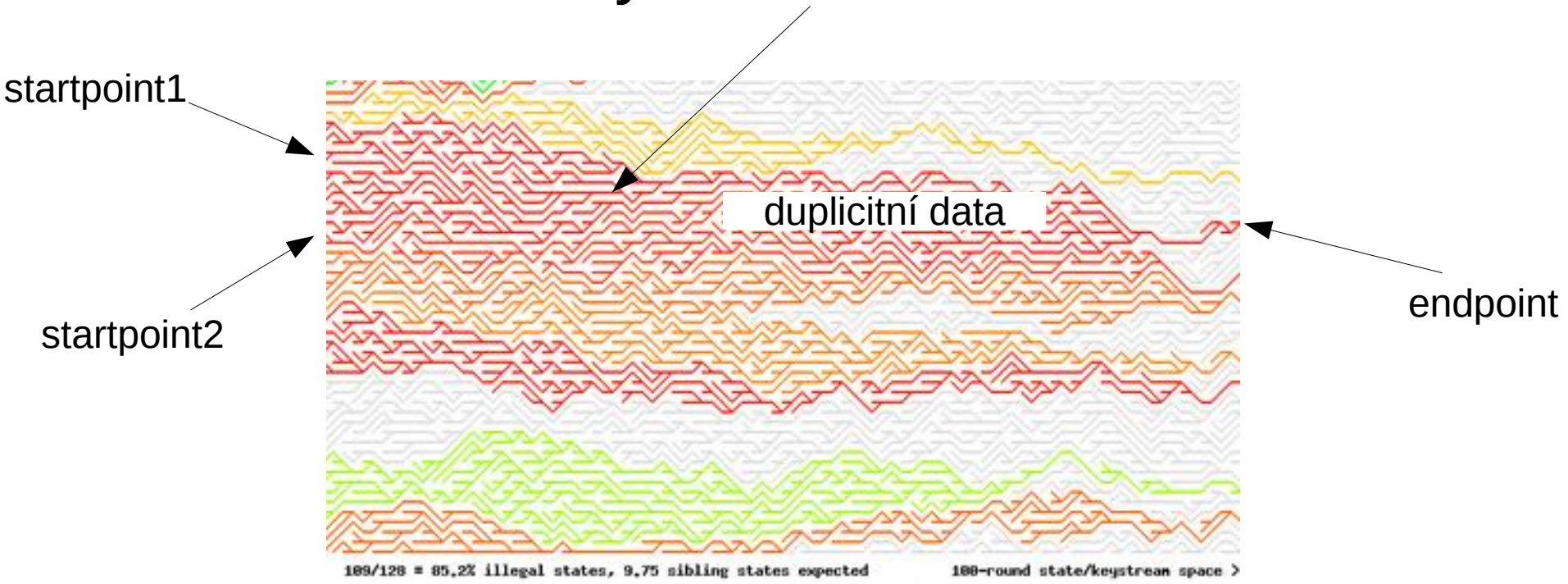
  ~~… 2^61 * 16B = 37 EB~~

  ~~… 2^61 * 16B / 204 = 181 PB~~

  … 2^61 * 16B / 204 / 4096 = 44 TB

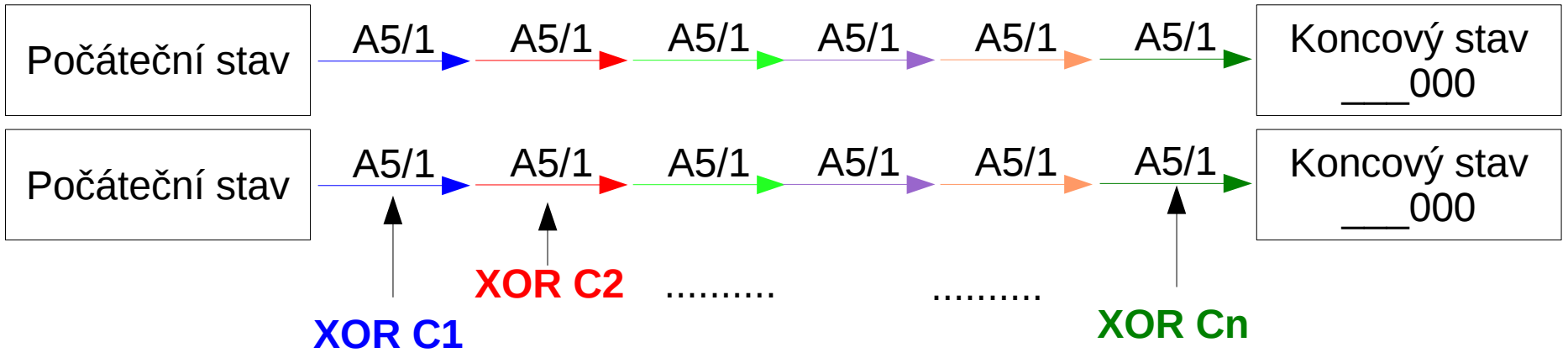
44 TB ... pořád trochu moc

- Prodloužit chainy? Kolize!



Barvy

- Barvy ~ konstanty, kterými po každé iteraci XORujeme



Stejné kolize v **různých** barvách jsou v každé iteraci algoritmu rozbity **rozdílnými konstantami**

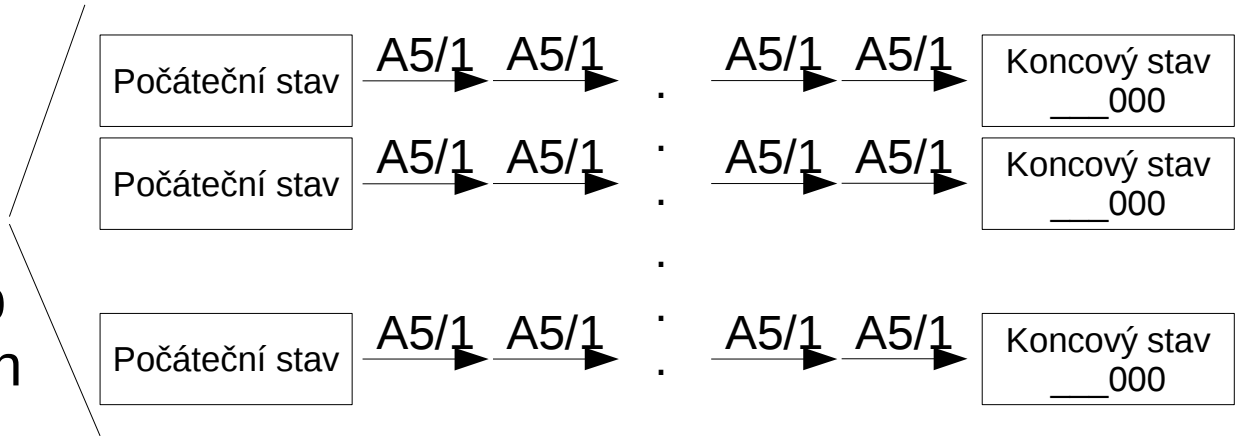
Pravděpodobnost, že ke kolizi dojde v jedné barvě, je n-krát menší

~~… 2^64 * 16B = 296 EB~~
~~… 2^61 * 16B = 37 EB~~
~~… 2^61 * 16B / 204 = 181 PB~~
~~… 2^61 * 16B / 204 / 4096 = 44 TB~~

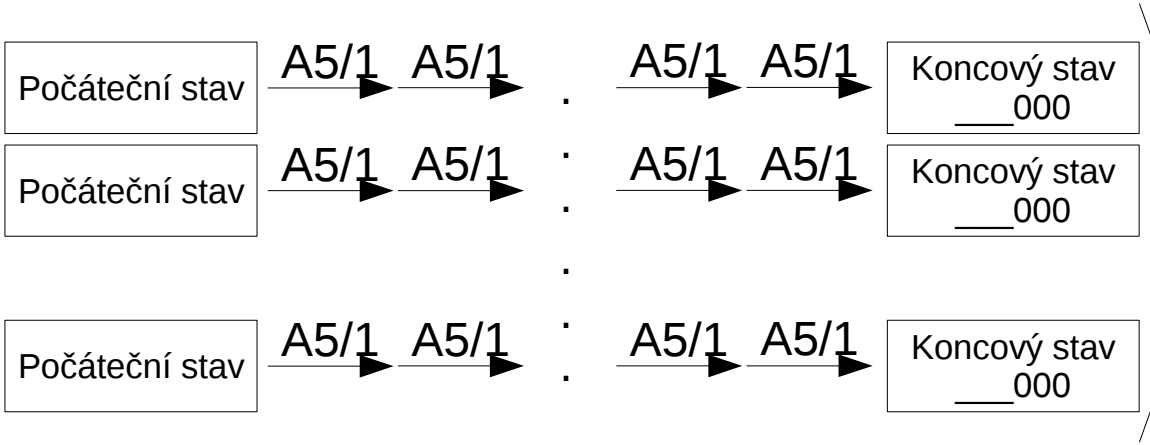
8 barev:
… 2^61 * 16B / 204 / 4096 / 8 = 5,5 TB

Startpoint encoding

výška tabulky 2^34
- → 34 bitů informace
- → zvolme hash fci 34→64b
- → ušetříme 30 bitů na chain



Endpoint encoding



2^34 setříděných čísel
efektivní prostor 2^(64-12)
průměrná mezera 2^(64-12-34)
encoding mezery pomocí 18 bitů!

→ **místo 128 bitů na chain nám stačí 34+18+režie**

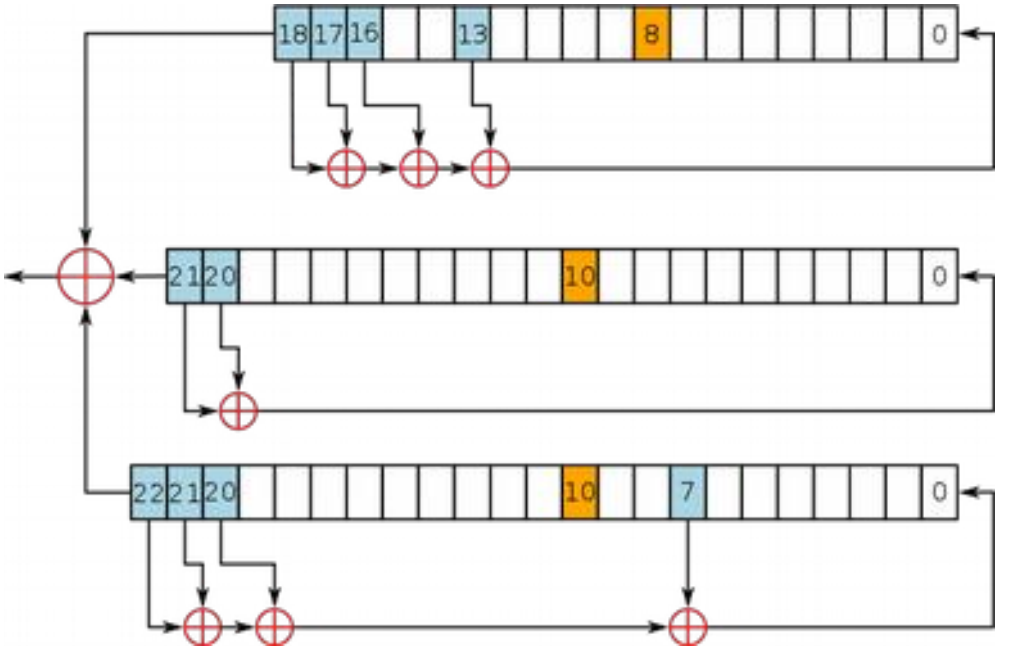
... ~~2^64 * 16B = 296 EB~~
... ~~2^61 * 16B = 37 EB~~
... ~~2^61 * 16B / 204 = 181 PB~~
... ~~2^61 * 16B / 204 / 4096 = 44 TB~~
... ~~2^61 * 16B / 204 / 4096 / 8 = 5,5 TB~~

```
/mnt/tables/gsm# du --si .
1.8T .
```

Realizace

- 40 tabulek * 8 barev * (8 barev * 4096 chainlinků od každé barvy) * 204 samplů
  → 2 miliardy A5/1 operací pro cracknutí jednoho framu
- AMD HD7970: 1 miliarda operací za sekundu
- Reálná úspěšnost hádání keystreamu je nižší…
- Podle operátora 5-60 sekund, úspěšnost 10-99 %
  - T-Mobile to fakt dobře zabezpečil a už to na něm skoro nefunguje
  - Vodafone je taky docela bezpečný
  - O2 je děravá

Naivní implementace A5/1



Z vašeho 64b procesoru se použije jen 22 bitů!

Tady se z 64bit registru používá dokonce jen 1 bit!!

Tady se rozbije branch prediction!

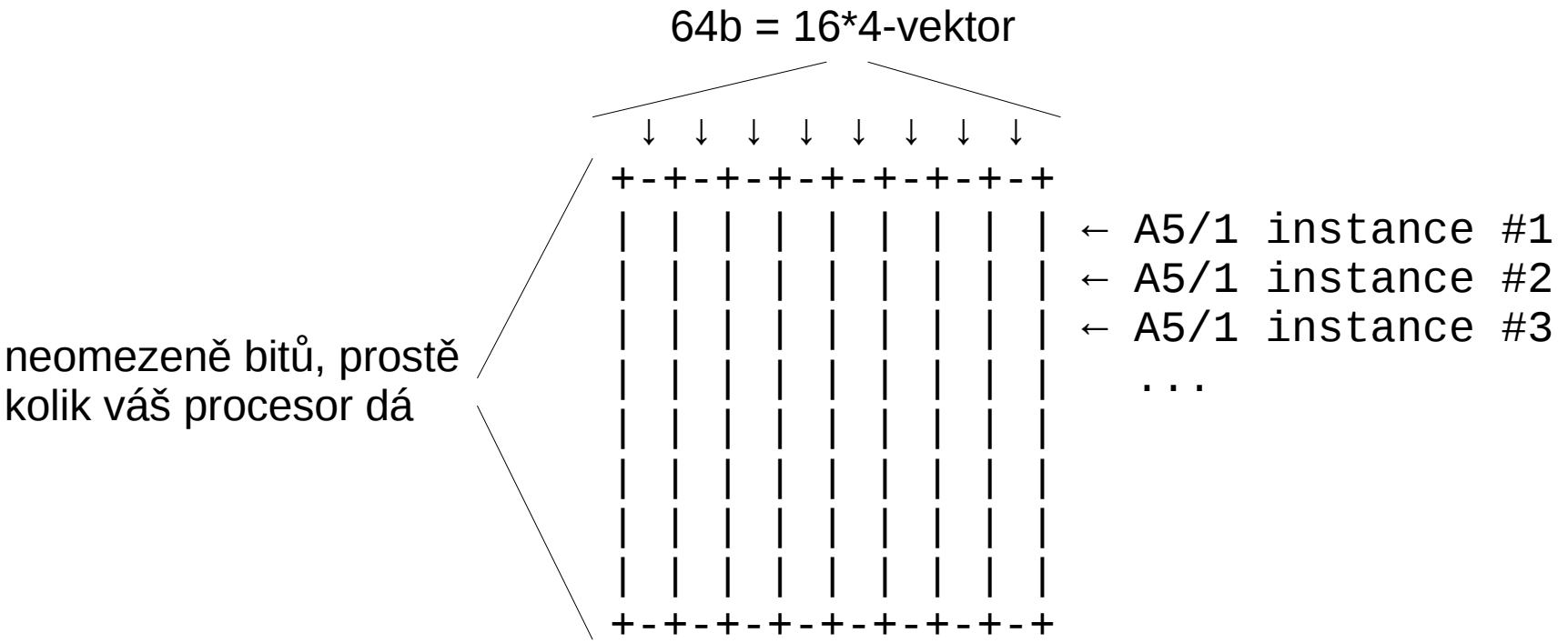
```
int reg1, reg2, reg3;
while(reg1 & 0xFFF) {
  bit1 = (reg1 >> 8)&1;
  ..
  ..
  xor1 = (reg1 >> 18) ^ (reg1 >> 17)..;
  ..
  ..
  if(...)
    reg1 = (reg1 << 1) | xor1;
  ..
  ..
}
```

Tohle na dnešních počítačích fungovat nebude...

Váš počítač…

- Pracuje s 256bit čísly najednou (SSE, AVX)
- Nedokáže dělat branch prediction, protože A5/1 je PRNG, který je prostě náhodný
- Na neuhádnutém skoku musí flushnout celou pipeline
- Je lepší spočítat třeba 4 zbytečné věci než skočit!

A5/1 slice machine



Shift registru: instrukce pro vektorovou rotaci, kruhové přiřazení
XOR: vektor s vektorem, třeba 256 bitů najednou

V každém taktu manipulujeme třeba s 64 instancemi šifry najednou!

Eliminace IF-ů

- Spočítáme oba stavy a pak si vybereme

```
_vector rotovat; // bitmaska
reg[i] = (reg[i+1] & rotovat) | (reg[i] & ~rotovat);
```

- reg je třeba pole uint64 – 64 instancí A5/1

- Pro detaily viz bohatě komentovaný “genkernel32” z Deky, nebo si mě odchytněte na afterparty

OpenCL - motivace

- ATI HD 7970:
  - 2048 jader, 8192 threadů na 1 GHz
  - 128bit vektory

    → 32 TB/s !!!

- OpenCL běží na grafikách, procesorech a trochu i na FPGAčkách
- OpenCL se vyplatí i na procesorech, abyste si nemuseli řešit paralelismus

OpenCL - architektura

- Aplikace předá OpenCL buffer a zdroják kernelu
- OpenCL spustí jeden kernel nad každým prvkem bufferu – kernely běží masivně paralelně
- Kernel přepíše vstupní data v bufferu výsledkem
- OpenCL vrátí buffer
- Aplikace buffer přečte

Kernel

```
kernel void brm(global ulong *buf) {

    // kernel index
    private size_t me = get_global_id(0);

    // local vars
    private ulong a;

    // read input data
    a = buf[me];

    // Compute something
    a = a + me*me;

    // write result back to memory
    buf[me] = a;
}
```

OpenCL - userspace

```
import pyopencl as cl, numpy as np

# create context
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

# create some input data, in this case array filled with zeros
a = np.array(np.zeros((20),dtype=np.uint64))

# create opencl buffer
buf = cl.Buffer(ctx, cl.mem_flags.READ_ONLY | cl.mem_flags.COPY_HOST_PTR, hostbuf=a)

# load kernel source
f=open("kernel.c","r")
SRC = ''.join(f.readlines())
f.close()

# compile it
prg = cl.Program(ctx, SRC).build()

print("Input:"); print(a)

# launch the kernel
event = prg.brm(queue, a.shape, None, buf)
event.wait()

# copy data back from opencl
cl.enqueue_copy(queue, a, buf)

# print it
print("CL returned:"); print(a)
```

OpenCL - run

```
~/tmp/ocltest> ./ocltest.py
Input:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
CL returned:
[  0   1   4   9  16  25  36  49  64  81 100 121
 144 169 196 225 256 289 324 361]
```

UAG