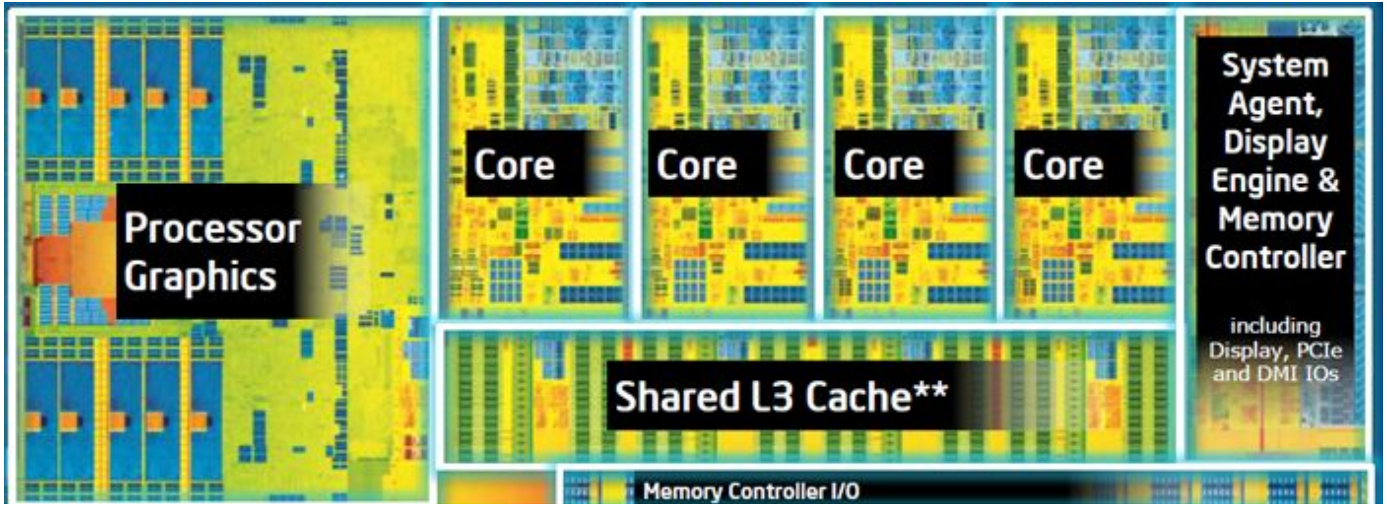
Cache and Syphilis

RootedCON 2019


FULL STACK DEVELOPERS
WE'LL PUT THEIR NAME TO THE TEST
imgflip.com
Warner Bros



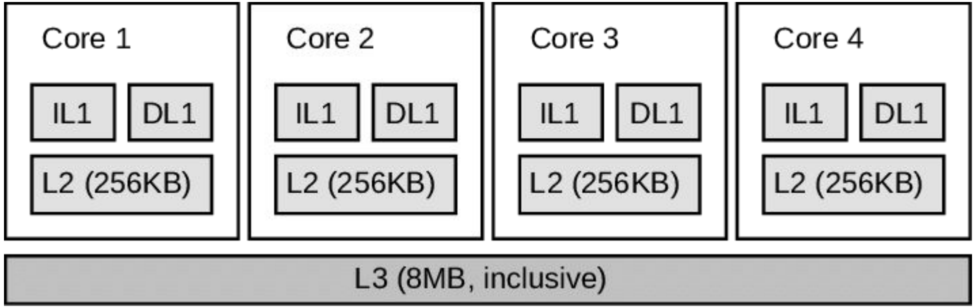
Haswell (4th generation) architecture

Cache latencies:

- L1 ~5 cycles
- L2 ~12 cycles
- L3 ~50 cycles
- DRAM ~50 cycles + 50 ns (RAM)

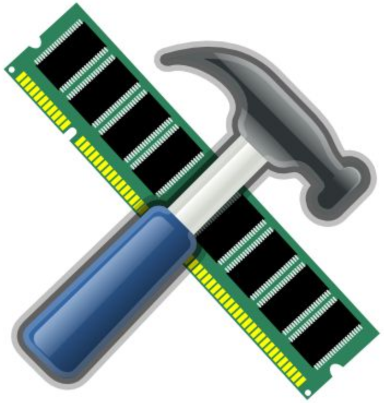
Coherence:

- Inclusive vs. non-inclusive vs. exclusive



/microarchitectural_attacks

Rowhammer



GPU

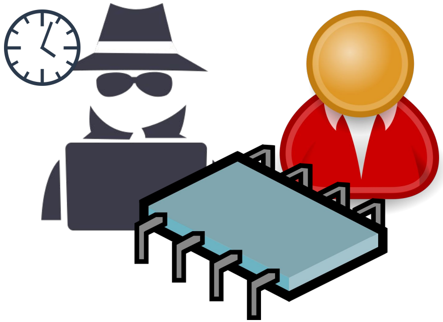
MMU and TLB

Port contention

Memory deduplication

Cache Attacks

Evict+Time, **Prime+Probe**, Flush+Reload, etc.



SPECTRE

Meltdown

Foreshadow

/rowhammer

Single Event Upsets (SEUs) in electronics first proposed in 1962 (J.T. Wallmark and S.M. Marcus)

- Cosmic rays can limit the scaling of devices

Can *random* bit-flips in physical memory be exploited?

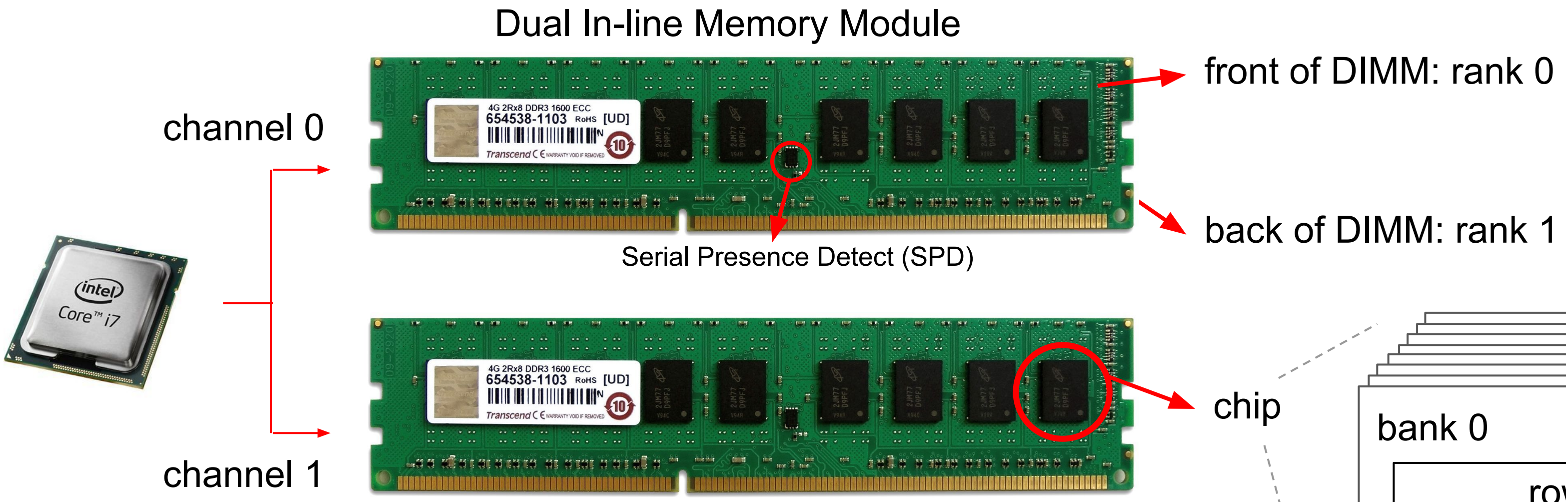
Rowhammer allows to induce *random* bit-flips via software in an often *repeatable* fashion

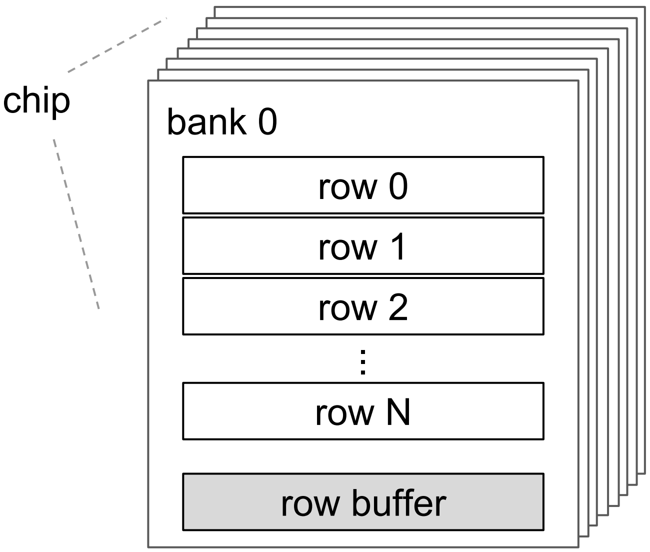
- <u>repetition</u> is what makes exploitation reliable

Some real exploits:

- **NaCl** bit-flip in x86 instructions to a non 32-byte-aligned address
- **Linux Kernel** bit-flip in physical frame number of PTE with R/W permission
- **RSA keys** (ssh and apt-get): bit-flip in public key allows easy factorization
- **Trusted Zone** bit-flip in private key, recover secret from signature
- **Opcode flipping** bit-flip to ignore privilege checks in `setuid` binaries

/rowhammer





Bank = matrix of cells

Cell = capacitor + transistor = 1-bit

Cells grouped into rows

Typical row size: 8K

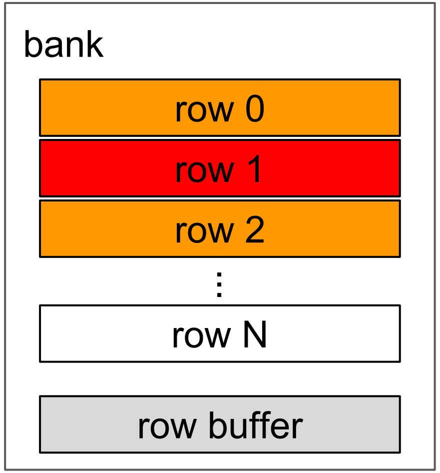
Row “activation”

Cells leak charge, need to refresh

Refresh rate ~64ms

/rowhammer

Hammering a row = repeatedly ***activating*** a row



- Higher storage capacity -> Higher cell density -> Lower isolation
- An aggressor row that is repeatedly activated can cause victim row's cells to bit-flip.
- Defective cells are randomly distributed

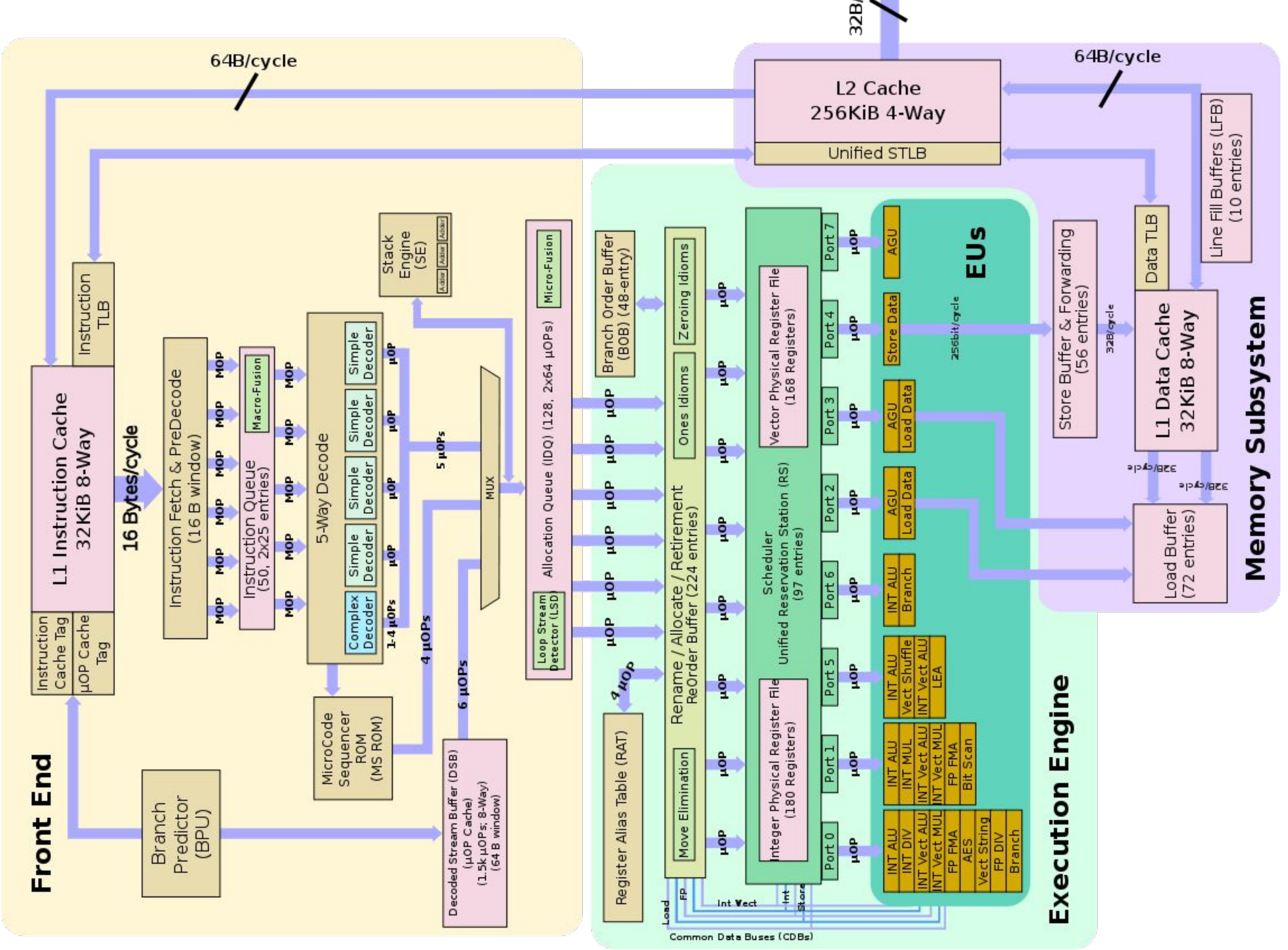
```
loop:
     mov (A), %eax // Read from address A (row 1)
     mov (B), %ebx // Read from address B (row k)
     clflush (A) // Flush A from cache
     clflush (B) // Flush B from cache
     jmp loop
```

/spectre-v1

- instruction fetch
- out-of-order

  execution
- branch prediction
- speculative

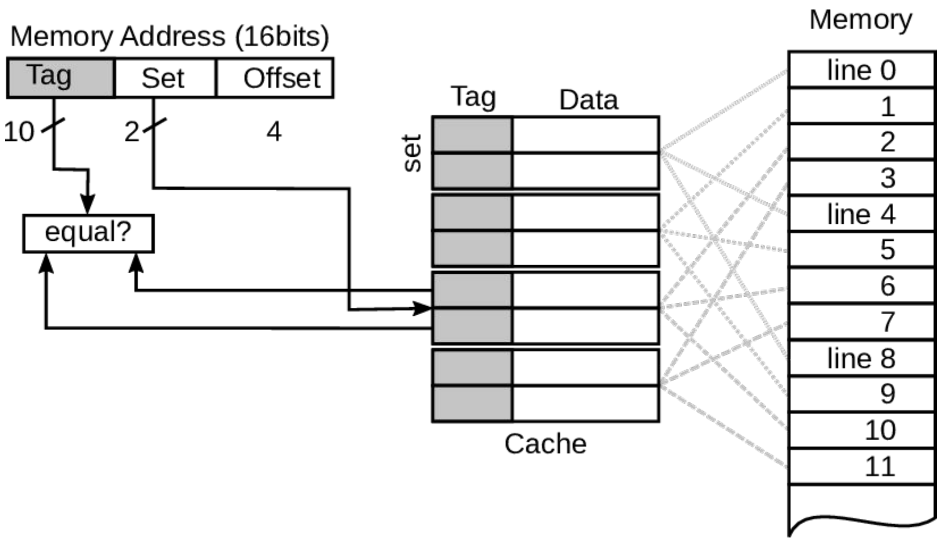
  execution



/spectre-v1

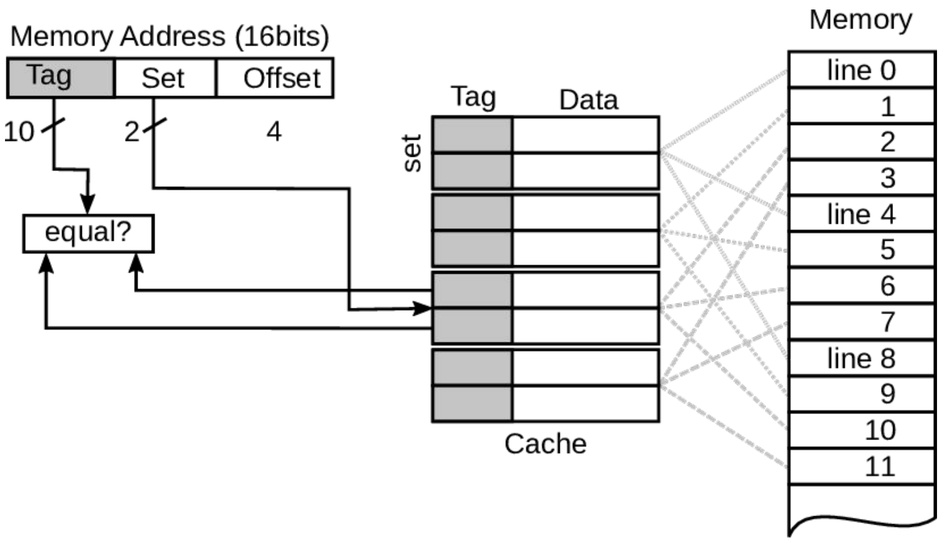
```
victim_func:                                                # void victim_func(int offset) {
        mov     eax, dword ptr [rip + arr1_size]           #
        cmp     rax, rdi                                    #   if (offset < arr1_size) {
        jbe     .OOB                                        #
        lea     rax, [rip + arr1]                           #
        movzx   eax, byte ptr [rdi + rax]                   #     eax = arr1[offset];
        shl     rax, 6                                      #     rax = rax * 64;
        lea     rcx, [rip + arr2]                           #
        mov     al, byte ptr [rax + rcx]                    #     al = arr2[rax];
        and     byte ptr [rip + temp], al                   #     temp = temp & al;
.OOB:                                                       #   }
        ret                                                 # return;
                                                            # }

arr1_size:
        .long   16                      # 0x10
        .size   arr1_size, 4
arr1:
        .ascii  "\001\002\003\004\005\006\007\b\t\n\013\f\r\016\017\020"
        .size   arr1, 16
temp:
        .byte   0                       # 0x0
        .size   temp, 1
arr2:
        .comm   arr2,131072,16
```

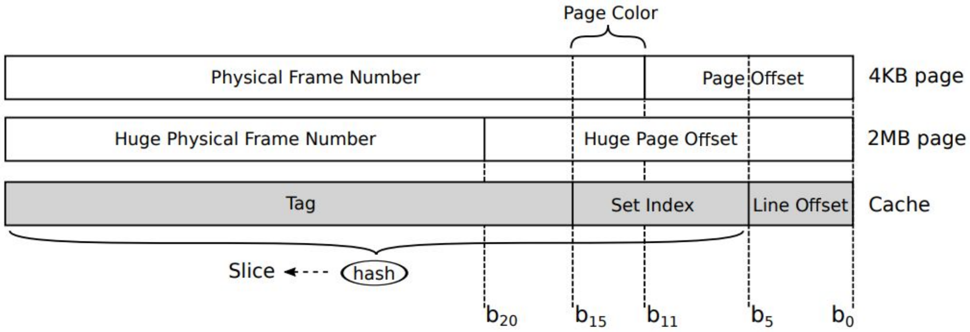
/caches



- Memory splitted in “blocks” (64B)
- Set-associative cache (*n* ways)
- Physically vs. virtually indexed
- Blocks “collide” in a cache set
- Replacement policy

/caches





- Memory splitted in “blocks” (64B)
- Set-associative cache (*n* ways)
- Physically vs. virtually indexed
- Blocks “collide” in a cache set
- Replacement policy

Example:

How many sets there are in a 6MB 12-way cache?

6*1024*1024 / (12*64) = 8192 sets

We need 13 set-index bits! (w/o slicing)

/caches

512 bytes 4-way toy cache

Set 0	Set 1

```
...
load @192 // @000011000000 (3)
load @264 // @000100001000 (4)
load @324 // @000101000100 (5)
load @096 // @000001100000 (1)
load @003 // @000000000010 (0)
load @464 // @000111010000 (7)
load @324 // @000101000100 (5)
load @576 // @001001000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

512 bytes 4-way toy cache

Set 0	Set 1

```
...
 load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
load @576 // @00100 1000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

512 bytes 4-way toy cache

Set 0	Set 1

MISS!

```
...
load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
load @576 // @00100 1000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

512 bytes 4-way toy cache

Set 0	Set 1
	D

```
...
 load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
load @576 // @00100 1000000 (9)
...
```

/caches

512 bytes 4-way toy cache

Set 0	Set 1
	D

MISS!

```
...
load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
load @576 // @00100 1000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

512 bytes 4-way toy cache

Set 0	Set 1
E	D

```
...
load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
load @576 // @00100 1000000 (9)
...
```

/caches

512 bytes 4-way toy cache

Set 0	Set 1
E	D

MISS!

```
...
load @192 // @000011000000 (3)
load @264 // @000100001000 (4)
load @324 // @000101000100 (5)
load @096 // @000001100000 (1)
load @003 // @000000000010 (0)
load @464 // @000111010000 (7)
load @324 // @000101000100 (5)
load @576 // @001001000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

512 bytes 4-way toy cache

Set 0	Set 1
E	D
	F

```
...
load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
load @576 // @00100 1000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

512 bytes 4-way toy cache

Set 0	Set 1
E	D
	F

MISS!

```
...
load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
 load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
load @576 // @00100 1000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

512 bytes 4-way toy cache

Set 0	Set 1
E	D
	F
	B

```
...
load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
 load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
load @576 // @00100 1000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

512 bytes 4-way toy cache

Set 0	Set 1
E	D
	F
	B

MISS!

```
...
load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
load @576 // @00100 1000000 (9)
...
```

/caches

512 bytes 4-way toy cache

Set 0	Set 1
E	D
A	F
	B

```
...
load @192 // @000011000000 (3)
load @264 // @000100001000 (4)
load @324 // @000101000100 (5)
load @096 // @000001100000 (1)
 load @003 // @000000000010 (0)
load @464 // @000111010000 (7)
load @324 // @000101000100 (5)
load @576 // @001001000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

512 bytes 4-way toy cache

Set 0	Set 1
E	D
A	F
	B

MISS!

```
...
load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
load @576 // @00100 1000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

512 bytes 4-way toy cache

Set 0	Set 1
E	D
A	F
	B
	H

```
...
load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
load @576 // @00100 1000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

512 bytes 4-way toy cache

Set 0	Set 1
E	D
A	**F**
	B
	H

HIT!

```
...
load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
load @576 // @00100 1000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

512 bytes 4-way toy cache

Set 0	Set 1
E	D
A	B
	H
	F

HIT!

```
...
load @192 // @000011000000 (3)
load @264 // @000100001000 (4)
load @324 // @000101000100 (5)
load @096 // @000001100000 (1)
load @003 // @000000000010 (0)
load @464 // @000111010000 (7)
load @324 // @000101000100 (5)
load @576 // @001001000000 (9)
...
```

Least Recently Used policy keeps the order!

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

512 bytes 4-way toy cache

Set 0	Set 1
E	D
A	B
	H
	F

MISS!

```
...
load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
 load @576 // @00100 1000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

512 bytes 4-way toy cache

Set 0	Set 1
E	~~D~~
A	B
	H
	F

Replace least recently used element

```
...
load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
 load @576 // @00100 1000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

512 bytes 4-way toy cache

Set 0	Set 1
E	~~D~~ B
A	H
	F

Update order

```
...
load @192 // @000011000000 (3)
load @264 // @000100001000 (4)
load @324 // @000101000100 (5)
load @096 // @000001100000 (1)
load @003 // @000000000010 (0)
load @464 // @000111010000 (7)
load @324 // @000101000100 (5)
load @576 // @001001000000 (9)
...
```

4K toy RAM

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/caches

512 bytes 4-way toy cache

Set 0	Set 1
E	B
A	H
	F
	J

Insert element in MRU position

```
...
load @192 // @00001 1000000 (3)
load @264 // @00010 0001000 (4)
load @324 // @00010 1000100 (5)
load @096 // @00000 1100000 (1)
load @003 // @00000 0000010 (0)
load @464 // @00011 1010000 (7)
load @324 // @00010 1000100 (5)
 load @576 // @00100 1000000 (9)
...
```

4K toy RAM

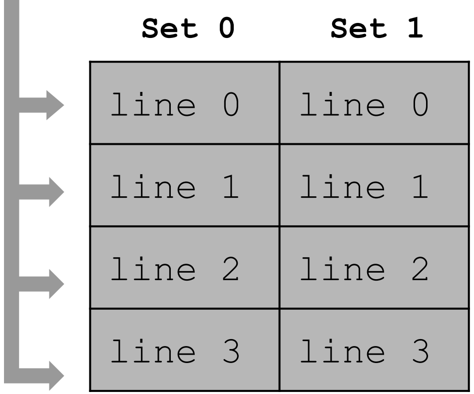
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q

...

/prime+probe

PRIME

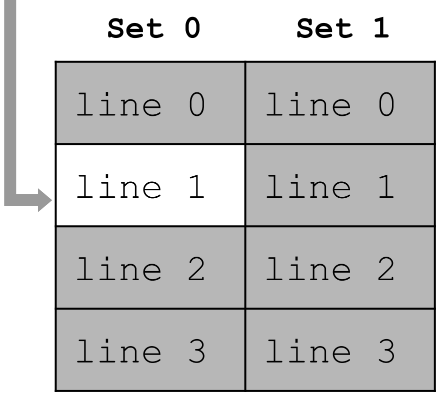
```
for each set : s
    for each line : l in s
        access (l)
```



Attacker’s put the cache in a known state

Victim

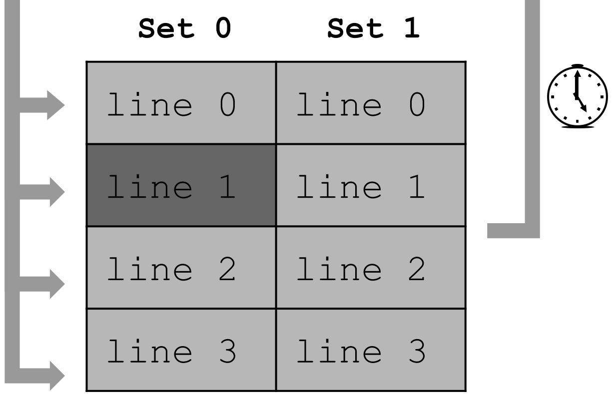
```
if (key[i] == 1):
    access (A)
else
    nop
```



Victim performs sensible computation

PROBE

```
for each set : s
    for each line : l in s
        time (access(l))
```



Attacker measure cache and learns which blocks have been evicted. Attacker can infer victim’s secret.

/problem

Now… In JavaScript (and many other environments) you do not have `` `clflush` `` instructions

You don't even have pointers: i.e. no knowledge of virtual address (much less of physical address)

- How can we exploit Rowhammer?
- How can we surgically evict specific content from the cache?
- How can we put the cache in a controlled state or probe it?

/problem

Now… In JavaScript (and many other environments) you do not have `` `clflush` `` instructions

You don't even have pointers: i.e. no knowledge of virtual address (much less of physical address)

- How can we exploit Rowhammer?
- How can we surgically evict specific content from the cache?
- How can we put the cache in a controlled state or probe it?

SOLUTION: Eviction Sets!

/eviction_sets

Two addresses ***x*** and ***y*** are <u>congruent</u> if `set(x) == set(y) & slice(x) == slice(y)`

```
set(paddr) := (paddr >> 6) & (NSETS/NSLICES-1)

slice(paddr) := /* RE hash function for Intel Core CPU with 8 slices */
                    [0x1b5f575440, 0x2eb5faa880, 0x3cccc93100]
```

“Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters” by Maurice et al.

An <u>eviction set</u> for an address ***v*** is a set of at least *associativity* congruent addresses.

(accessing an eviction sets ensures that ***v*** is evicted from the cache)*

Without info about the physical address, we can <u>test</u> if a set is an eviction set for $\boldsymbol{a_v}$:

$$a_v \ \underbrace{a_0 \ a_1 \ \dots \ a_{n-1}} \ \boxed{a_v}$$

/eviction_sets

size associativity

Any sufficiently large set is an eviction set, but efficient attacks require minimal eviction sets

How hard is to find them?

- Depends on the amount of information about the physical address: **γ**
- Probability of two random addresses with **γ** identical set-index bits to collide: $\mathbf{p=2^{\gamma-s-c}}$
- Probability of a set of size ***N*** of having, at least, ***a*** collisions with some ***x***:

c = set-index bits
s = slice bits

$$
\begin{aligned}
P(|S \cap [x]| \geq a) &= 1 - P(X < a) \\
&= 1 - \sum_{k=0}^{a-1} \binom{N}{k} p^k (1-p)^{N-k}
\end{aligned}
$$

/eviction_sets

Let's imagine an attacker with native execution on a system with:
4KB pages, 8192 cache sets, 8 slices, and associativity 12

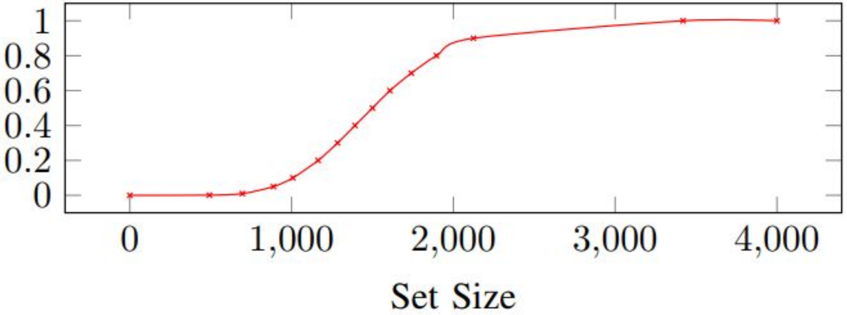
Virtual : b_{47} … b_{16} b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0

Physical: b_{39} … b_{16} b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0

s_2 s_1 s_0 (has control over 12 bits -> P(collision) = 2^{-7})

Page Offset
Set index
Line offset
Slice index

Probability of a having an eviction set (associativity=12) for a given address as a function of the set size:



/eviction_sets

But how do we find minimal eviction sets for ***v***? We use our test: $a_v\ a_0\ a_1\ \ldots\ a_{n-1}\ \boxed{a_v}$

~~Bruteforce: try all subsets of size associativity until one evicts the address ***v***~~

Simple reduction:

```
1.   start with a large enough eviction set S
2.   C := pop(S) // pick one candidate element
3.   if not (S is eviction set) append(S, C) // C is congruent
4.   if (|S| = associativity) return S
5.   else goto(2)
```

(It’s also possible to keep track of congruent elements and stop once you find assoc many, but it’s less robust against noise)

Problem? **O(n^2)** memory accesses worst case (or $n^2/2$). Can we do better?

Intuition: One can improve performance by finding trade offs with a smaller initial set size, or by changing step-2 to remove several elements instead of one (but it’s still asymptotically quadratic)

/group_testing

https://en.wikipedia.org/wiki/Group_testing

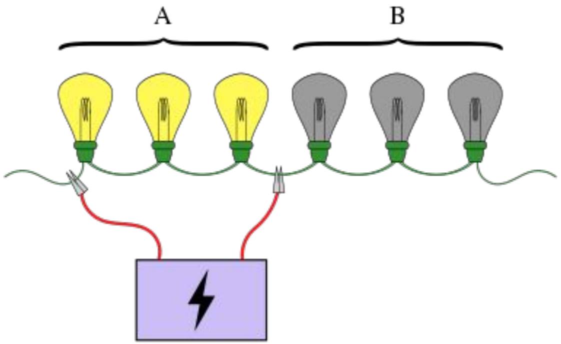
“Robert Dorfman in 1943 propose to the novel idea of **group testing** to reduce the expected **number of tests** needed to weed out **all syphilitic men** in a given pool of soldiers:

- **put the soldiers into groups** of a given size,
- and use individual **testing** (testing items in groups of size one) **on the positive groups** to find which were infected.

Dorfman tabulated the **optimum group sizes** for this strategy against the prevalence rate of **defectiveness** in the population.”

Improved by **Generalised binary-splitting algorithm**: find **d** defective elements in **~d*log(n/d)** tests.

Light Bulb problem example:



/eviction_sets

We can do better inspired by group testing techniques.

In particular, by using **threshold group-testing**:

- Generalization to group-testing by Peter Damaschke (2006)
- A test gives:
  - a **positive** answer if the group contains at least ***u*** defective elements
  - a **negative** answer if the group contains at most ***l*** defective elements
  - an arbitrary answer otherwise

Our test, gives a positive answer if the set contains at least *associativity* elements, and a negative answer otherwise. **<u>Cache congruence is a threshold group-test!</u>**

More specifically, this is a t**hreshold group-testing without gap**:

$$\begin{cases} \mathbf{u = associativity} \\ \mathbf{l = u - 1} \end{cases}$$

And it is possible to identify ***d*** defective elements in **O(d * log (n))** tests.

/eviction_sets

Our problem is slightly different: we need to minimize the number of memory accesses, not the number of tests (i.e. the size of each test matters)

Group-testing reduction:

```
1.   start with a large enough eviction set S
2.   T := split S in ASSOC+1 equally sized subsets
3.   i := 1
4.   while not (S\T_i is eviction set) do // pidgeon principle guarantees termination
5.        i := i+1
6.   done
7.   S := S\T_i // discard unnecessary subset
8.   if (|S| = associativity) return S
9.   else goto(2)
```

This takes us **O(n)** memory accesses worst case!*

*For a complete cost analysis check the paper: $a^2n+an-a^3-a^2$.

/eviction_sets

Tool for finding eviction sets: https://github.com/cgvwzq/evsets

```
$ sudo ./evsets -b 3000 -c 6 -s 8 -a g -e 2 -n 12 -o 4096 -r 10 -t 95 -C 0 --verify --retry --backtracking --nohugepages
[+] 11 MB buffer allocated at 0x7fb6dd79a000 (192000 blocks)
[+] Default Threshold = 95
[+] Initial candidate set evicted victim
[+] Created linked list structure (3000 elements)
[+] Starting group reduction...
[+] Reduction time: 0.057381 seconds
[+] Total execution time: 0.086908 seconds
[+] (ID=0) Found minimal eviction set for 0x7fb6d579a000 (length=12): 0x7fb6ddc25000 0x7fb6dfe85000 0x7fb6e08d5000 0x7fb6e4475000
0x7fb6dde75000 0x7fb6e47e5000 0x7fb6ddb69000 0x7fb6e0715000 0x7fb6e4d85000 0x7fb6dfb55000 0x7fb6de63d000 0x7fb6df995000
[+] Verify eviction set (only in Linux with root):
 - victim pfn: 0x2e1a95000, cache set: 0x140, slice: 0x2
 - element pfn: 0x448915000, cache set: 0x140, slice: 0x2
 - element pfn: 0x2e18f5000, cache set: 0x140, slice: 0x2
 - element pfn: 0x2e34f5000, cache set: 0x140, slice: 0x2
....
 - element pfn: 0x2f91d5000, cache set: 0x140, slice: 0x2
 - element pfn: 0x2e6325000, cache set: 0x140, slice: 0x2
 - element pfn: 0x2e4c65000, cache set: 0x140, slice: 0x2
[+] Verified!
```

/eviction_sets

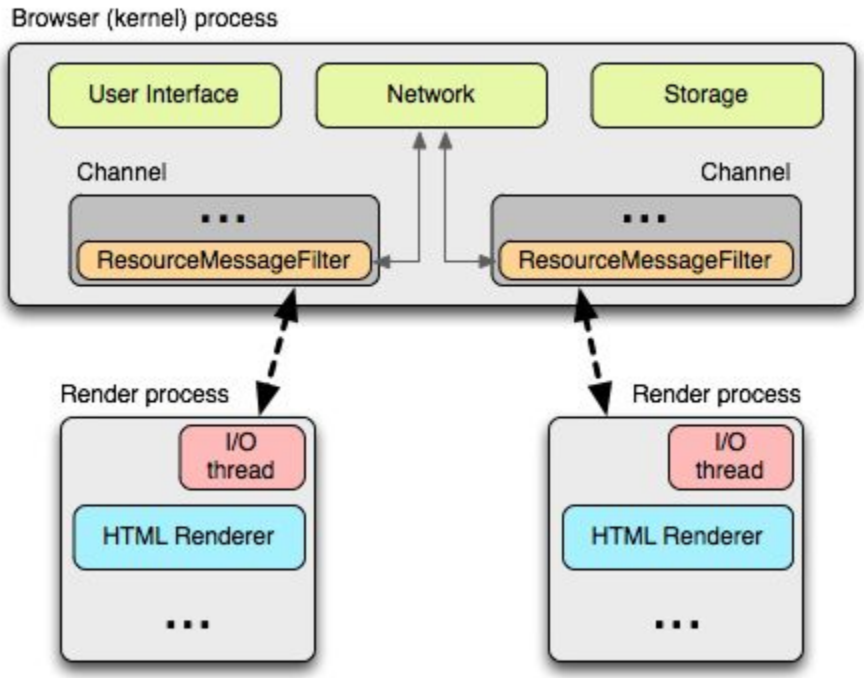
Tool for finding eviction sets: https://github.com/cgvwzq/evsets

```
$ sudo ./evsets -b 3000 -c 6 -s 8 -a g -e 2 -n 12 -o 4096 -r 10 -t 95 -C 0 --verify --retry --backtracking --nohugepages
[+] 11 MB buffer allocated at 0x7fb6dd79a000 (192000 blocks)
[+] Default Threshold = 95
[+] Initial candidate set evicted victim
[+] Created linked list structure (3000 elements)
[+] Starting group reduction...
[+] Reduction time: 0.057381 seconds
[+] Total execution time: 0.086908 seconds
[+] (ID=0) Found minimal eviction set for 0x7fb6d579a000 (length=12): 0x7fb6ddc25000 0x7fb6dfe85000 0x7fb6e08d5000 0x7fb6e4475000
0x7fb6dde75000 0x7fb6e47e5000 0x7fb6ddb69000 0x7fb6e0715000 0x7fb6e4d85000 0x7fb6dfb55000 0x7fb6de63d000 0x7fb6df995000
[+] Verify eviction set (only in Linux with root):
 - victim pfn: 0x2e1a95000, cache set: 0x140, slice: 0x2
 - element pfn: 0x448915000, cache set: 0x140, slice: 0x2
 - element pfn: 0x2e18f5000, cache set: 0x140, slice: 0x2
 - element pfn: 0x2e34f5000, cache set: 0x140, slice: 0x2
....
 - element pfn: 0x2f91d5000, cache set: 0x140, slice: 0x2
 - element pfn: 0x2e6325000, cache set: 0x140, slice: 0x2
 - element pfn: 0x2e4c65000, cache set: 0x140, slice: 0x2
[+] Verified!
```

Sure... But what about JS?

/chrome

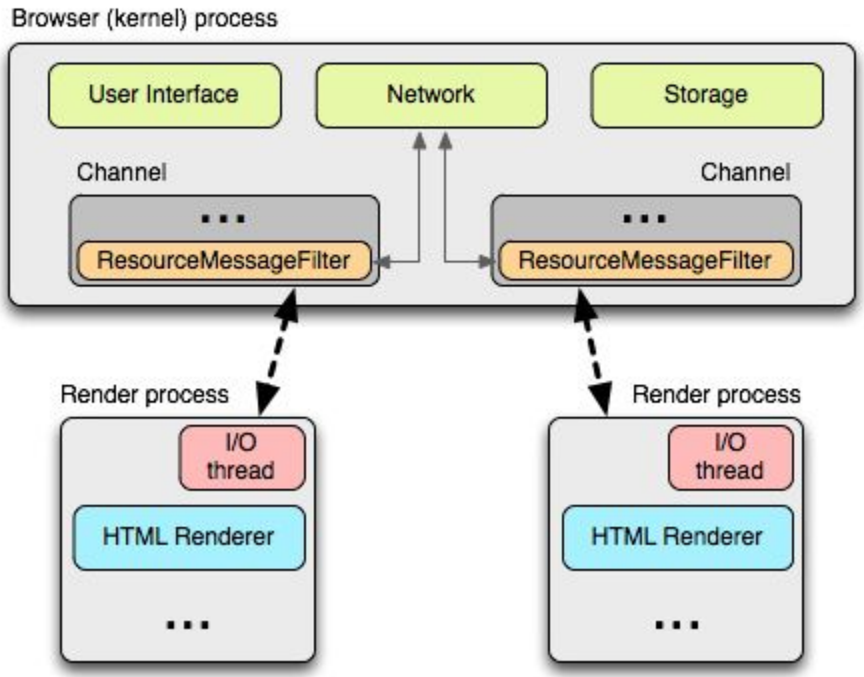
Multi-process architecture



https://www.aosabook.org/en/posa/high-performance-networking-in-chrome.html

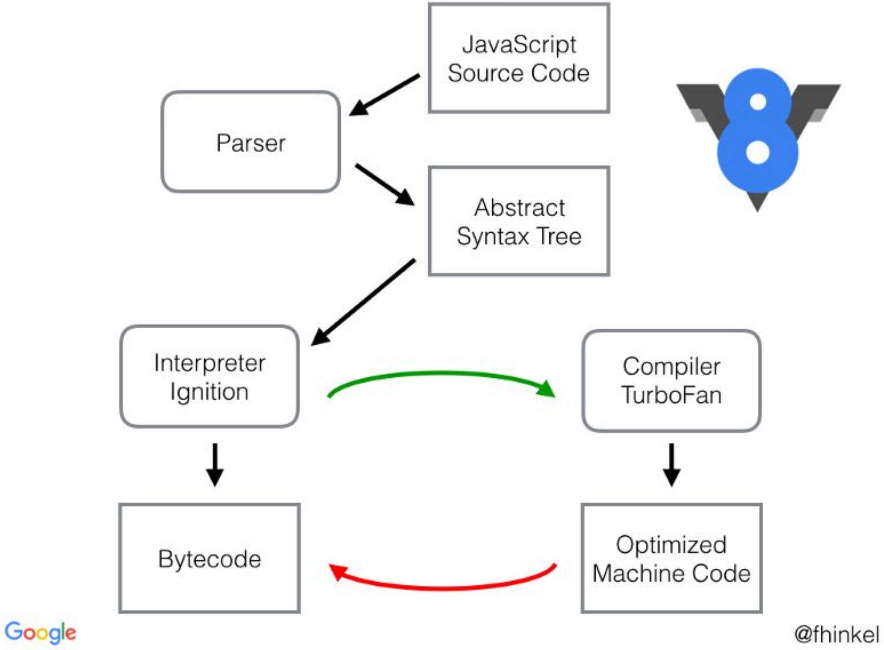
/chrome

Multi-process architecture



https://www.aosabook.org/en/posa/high-performance-networking-in-chrome.html

TurboFan optimizer compiler



https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775

/wasm

WebAssembly

- Stack machine
- Harvard architecture
- Statically typed language
- S-expressions
- Flat and sandboxed (fault isolation) memory
- More performance and predictability
- LiftOff vs. TurboFan (in Chrome)

/chrome

Memory allocation:

- PartitionAlloc: Blink's default memory allocator with built-in exploit mitigations (substitutes previous tcmalloc)
- Oilpan: PatitionAlloc is no longer used to allocate memory for DOM related C++ objects in Chrome
- Oilpan (or BlinkGC) is a new garbage collected heap
- Allocator can be changed at compile time (tcmalloc vs. jemalloc vs. winheap)

Some subsystems, such as the V8 JavaScript engine, handle memory management autonomously:

- Orinoco GC: from sequential and stop-the-world to mostly parallel and concurrent (512KB pages, 3 generations, scavenges and full mark-compact)
- Different Object spaces: new, old, code, map, large, code large, new large, read only.
- We are interested in "LargeObjectSpace": allocations greater than a region (512KB) use `mmap` directly.

`mmap` -> page aligned region -> control over the page offset from JS! :)

/timers

We also need a high-resolution timer:

- `performance.now()` is too coarse…
- there are many alternatives and amplification techniques
- but since `SharedArrayBuffer` has been re-enabled, let’s use it

We can implement a high-resolution clock with a thread (aka `Worker`) and shared memory counter:

```
// file: clock.js

var arr;
onmessage = function(evt) {
    arr = new Uint32Array(evt.memory.buffer);
    while (true) {
        arr[0]++;
    }
}
```

```
const clock = new Worker('clock.js');

const buffer = new SharedArrayBuffer(4);
const rdtsc = new Uint32Array(buffer);
clock.postMessage({"memory”: buffer});
...
let t0 = rdtsc[0];
foo();
let t1 = rdtsc[0] - t0;
```

/wasm

(on V8 v7.4.0)

a.wat

```
(module
  (import "env" "mem" (memory 1 1 shared))
  (export "access" (func $x))

  ;; simple memory access
  (func $x (param $ptr i32) (result i32)
    (i32.load (get_local $ptr))
    (return)
  )
)
```

How to build?

```
$ wat2wasm --enable-threads -o a.wasm a.wat

$ cat wrapper.js

        const mem = new WebAssembly.Memory({
                initial: 1, maximum: 1, shared: true
        });
        const buf = new Uint8Array(readbuffer(arguments[0]));
        const ffi = {env:{mem:mem}};

        const module = new WebAssembly.Module(buf);
        const instance = new WebAssembly.Instance(module, ffi);
        instance.exports.access(0);

$ d8 --experimental-wasm-threads --code-comments
--wasm-tier-mask-for-testing 255 --print-wasm-code
--experimental-wasm-bigint wrapper.js -- a.wasm
```

TurboFan x86_64 output

```
0   55                    push rbp                          prologue
1   4889e5                movq rbp,rsp
4   6a0a                  push 0xa
6   56                    push rsi
7   488b9ea7000000        movq rbx,[rsi+0xa7] ;; memory
e   8bd0                  movl rdx,rax                      sign check
10  49ba0000000001000000  movq r10,0x100000000
1a  4c3bd2                cmpq r10,rdx
1d  7320                  jnc <+0x3f>
                          ...
3f  8b0413                movl rax,[rbx+rdx*1]
42  4c8b15c9ffffff        movq r10,[rip+0xffffffc9]
49  4c3bd0                cmpq r10,rax
4c  731d                  jnc <+0x6b>
                          ...
6b  488be5       epilogue movq rsp,rbp
6e  5d                    pop rbp
6f  c3                    retl
```

Body (size = 608 = 593 + 15 padding)

```
wasm-instance = rsi

args = rax, rdx, rcx, rbx, r9, (stack)

return = rax
```

/wasm

(on V8 v7.4.0)

a.wat

```
(module
  (import "env" "mem" (memory 1 1 shared))
  (export "access" (func $x))

  ;; simple memory access
  (func $x (param $ptr i32) (result i32)
    (i32.load (get_local $ptr))
    (return)
  )
)
```

a.wat

```
;; optimized memory access
(func $x (param $ptr i32) (result i64)
  (i64.load
    (i32.and
      (i32.const 0xffffffff)
      (get_local $ptr)))
  (return)
)
```

TurboFan x86_64 output

```
0  55               push rbp                        ┐
1  4889e5           movq rbp,rsp                    │ prologue
4  6a0a             push 0xa                        │
6  56               push rsi                        ┘
7  488b9ea7000000   movq rbx,[rsi+0xa7] ;; memory
e  25ffffff00       andl rax,0xff
13 488b0403         movq rax,[rbx+rax*1]
17 488be5           ┌ movq rsp,rbp
1a 5d      epilogue │ pop rbp
1b c3               └ retl
```

Body (size = 384 = 357 + 27 padding)

- `i64.load` vs `i32.load`
- apply bitmask to offset removes sign check
- safe bytes and control layout :)

/wasm

(on V8 v7.4.0)

clock.wat

```
;; load-store increment loop
(func $x
  (loop $ccc
    (i64.store
      (i32.const 0)
      (i64.add
        (i64.load (i32.const 0))
        (i64.const 1)))
    (br $ccc))
)
```

clock.wat

```
;; atomic increment loop
(func $x
  (loop $ccc
    (i64.atomic.rmw.add
      (i32.const 0)
      (i64.const 1))
    (br $ccc))
)
```

TurboFan x86_64 output

```
0   55                     push rbp
1   4889e5                 movq rbp,rsp
4   6a0a                   push 0xa
6   56                     push rsi
7   4883ec08               subq rsp,0x8
b   488b86a7000000         movq rax,[rsi+0xa7] ;; memory
12  e913000000             jmp <+0x2a>
17  660f1f840000000000     nop
20  488b18                 movq rbx,[rax]
23  4883c301               addq rbx,0x1
27  488918                 movq [rax],rbx
2a  488b9ec700000          movq rbx,[rsi+0xc7]
31  483923                 cmpq [rbx],rsp
34  72ea                   jc <+0x20>
                           ...
;; infinite loop: compiler removes epilogue! :o
```

prologue

stack guard

/wasm

(on V8 v7.4.0)

clock.wat

```
;; load-store increment loop
(func $x
  (loop $ccc
    (i64.store
      (i32.const 0)
      (i64.add
        (i64.load (i32.const 0))
        (i64.const 1)))
    (br $ccc))
)
```

clock.wat

```
;; atomic increment loop
(func $x
  (loop $ccc
    (i64.atomic.rmw.add
      (i32.const 0)
      (i64.const 1))
    (br $ccc))
)
```

TurboFan x86_64 output

```
0   55               push rbp
1   4889e5           movq rbp,rsp
4   6a0a             push 0xa
6   56               push rsi
7   4883ec08         subq rsp,0x8
b   488b9ea7000000   movq rbx,[rsi+0xa7] ;; memory
12  ba01000000       movl rdx,0x1
17  e914000000       jmp <+0x30>
1c  0f1f4000         nop
20  488b03           movq rax,[rbx]
23  488bc8           movq rcx,rax
26  4803ca           addq rcx,rdx
29  f0480fb10b       lock cmpxchgq [rbx],rcx
2e  75f0             jnz <+0x20>
30  488b86c7000000   movq rax,[rsi+0xc7]
37  483920           cmpq [rax],rsp
3a  72e4             jc <+0x20>

;; infinite loop: compiler removes epilogue! :o
```

prologue

stack guard

/wasm

(on V8 v7.4.0)

traverse.wat

```
(func $x (param $ptr i64)
  (loop $iter
    (set_local $ptr
      (i64.load
        (i32.wrap/i64 (get_local $ptr))))
    (br_if $iter
      (i32.eqz (i64.eqz (get_local $ptr)))))
)
```

TurboFan x86_64 output

```
0   55                           push rbp
1   4889e5                       movq rbp,rsp
4   6a0a                         push 0xa                 prologue
6   56                           push rsi
7   4883ec10                     subq rsp,0x10
b   488b9ea7000000               movq rbx,[rsi+0xa7]
12  6666660f1f840000000000       nop
1d  0f1f00                       nop
20  488b96c7000000               movq rdx,[rsi+0xc7]
27  483922                       cmpq [rdx],rsp           stack guard
2a  0f8340000000                 jnc <+0x70>
30  8bc0                         movl rax,rax
32  49ba0000000001000000         movq r10,0x100000000     sign check
3c  4c3bd0                       cmpq r10,rax
3f  7320                         jnc <+0x61>
                                 ...
61  488b0403                     movq rax,[rbx+rax*1]
65  4883f800                     cmpq rax,0x0
69  75b5                         jnz <+0x20>
6b  488be5                       movq rsp,rbp
6e  5d               epilogue    pop rbp
6f  c3                           retl
```

/wasm

(on V8 v7.4.0)

hit.wat

```
(func $x (param $victim i32) (result i64)
  (i64.const 0)
  ;; access victim
  (drop (i64.load (i32.and
    (i32.const 0xffffffff)
    (get_local $victim))))
  ;; t_start
  (i64.load (i32.const 0));; shared counter
  ;; re-access victim
  (drop (i64.load (i32.and
    (i32.const 0xffffffff)
    (get_local $victim)))))
  ;; t_end
  (i64.load (i32.const 0));; shared counter
  (i64.sub)
  (i64.sub)
  (return)
)
```

TurboFan x86_64 output

```
0   55              push rbp
1   4889e5          movq rbp,rsp
4   6a0a            push 0xa
6   56              push rsi
7   488b9ea7000000  movq rbx,[rsi+0xa7]   ;; memory
e   83e0ff          andl rax,0xff
11   488b1403       movq rdx,[rbx+rax*1]  ;; access()
15   488b13         movq rdx,[rbx]        ;; t_start
18   488b0c03       movq rcx,[rbx+rax*1]  ;; access()
1c   488b1b         movq rbx,[rbx]        ;; t_end
1f   482bd3         subq rdx,rbx
22   48f7da         negq rdx
25   488bc2         movq rax,rdx
28   488be5         movq rsp,rbp
2b   5d             pop rbp
2c   c3             retl
```

prologue

epilogue

Problem: x86 looks good, but we have no guarantees that memory accesses are done in order…

In practice we get a lot of zeros :(

Solutions: Repeat measure many times, do internal calls instead of inlining accesses, or create a data dependency

/wasm

(on V8 v7.4.0)

hit_opt.wat

```
(func $x (param $victim i32) (result i64)
  (local $t0 i64) (local $t1 i64) (local $td i64)
  ;; acces victim
  (set_local $td (i64.load (i32.and
    (i32.const 0xffffffff)
    (get_local $victim))))
  ;; t_start
  (set_local $t0 (i64.load (i32.and
     (i32.const 0xffffffff)
     ((i32.eqz (i64.eqz (get_local $td))))))
  ;; re-access victim
  (set_local $td (i64.load (i32.and
    (i32.const 0xffffffff)
    (i32.or (get_local $victim) (i64.eqz
(get_local $t0))))))
  ;; t_end
  (set_local $t1 (i64.load (i32.and
    (i32.const 0xffffffff)
    ((i32.eqz (i64.eqz (get_local $td))))))
  ;; ret time
  (i64.sub (get_local $t1) (get_local $t0))
  (return)
)
```

TurboFan x86_64 output

Offset	Bytes	Instruction	Section
0	55	push rbp	prologue
1	4889e5	movq rbp,rsp	
4	6a0a	push 0xa	
6	56	push rsi	
7	488b9ea7000000	movq rbx,[rsi+0xa7] ;; memory	
e	488bd0	movq rdx,rax	victim
11	83e2ff	andl rdx,0xff	
14	488b1413	movq rdx,[rbx+rdx*1]	
18	4883fa00	cmpq rdx,0x0	t_start
1c	0f94c2	setzl dl	
1f	0fb6d2	movzxbl rdx,rdx	
22	83e2ff	andl rdx,0xff	
25	488b1413	movq rdx,[rbx+rdx*1]	
29	4883fa00	cmpq rdx,0x0	victim
2d	0f94c1	setzl cl	
30	0fb6c9	movzxbl rcx,rcx	
33	0bc8	orl rcx,rax	
35	83e1ff	andl rcx,0xff	
38	488b0c0b	movq rcx,[rbx+rcx*1]	
3c	4883f900	cmpq rcx,0x0	t_end
40	0f94c1	setzl cl	
43	0fb6c9	movzxbl rcx,rcx	
46	83e1ff	andl rcx,0xff	
49	488b1c0b	movq rbx,[rbx+rcx*1]	
4d	482bda	subq rbx,rdx	
50	488bc3	movq rax,rbx	
53	488be5	movq rsp,rbp	epilogue
56	5d	pop rbp	
57	c3	retl	

/demo


Oingo Boingo
Weird Science
MCA Records
Promo Only
By Ichnos71