Faulting Hardware from Software

Daniel Gruss

2020-09-13

Graz University of Technology


CTV

Test - Mozilla Firefox (on lab02)

Test

file:///home/dgruss/rowhammerjs/rowhammer.html

Search

320: 12
330: 9
340: 1
350: 0
360: 1
370: 2
380: 199
390: 76
400: 72
410: 231
420: 572
1250
[!] Found flip (254 != 255) at array index 340021386 when hammering indices 339881984 and 340156416

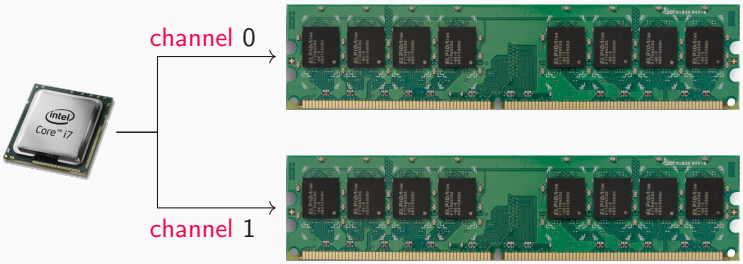
[!] Found flip (239 != 255) at array index 340022176 when hammering indices 339881984 and 340156416

[!] Found flip (191 != 255) at array index 340023138 when hammering indices 339881984 and 340156416

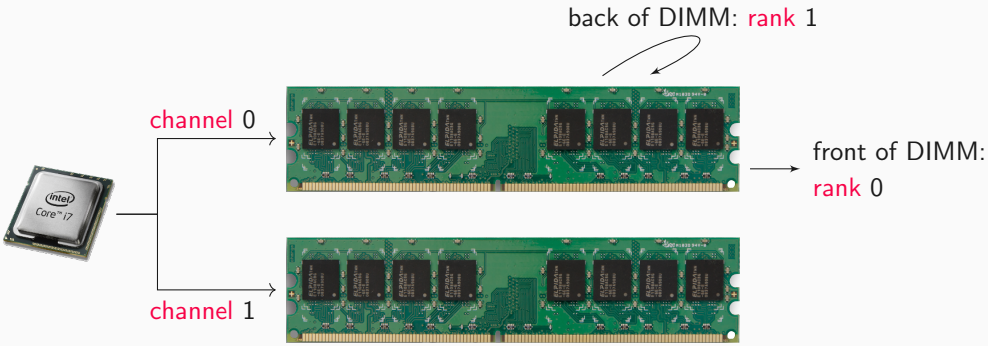
[!] Found flip (254 != 255) at array index 340025146 when hammering indices 339881984 and 340156416

DRAM organization

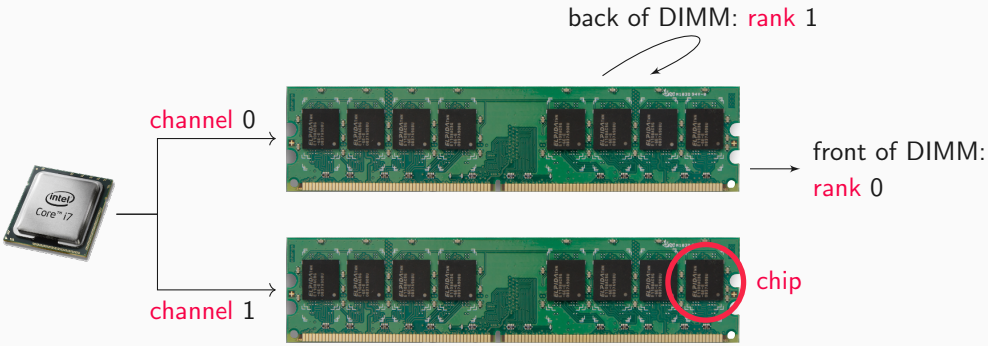
DRAM organization



DRAM organization

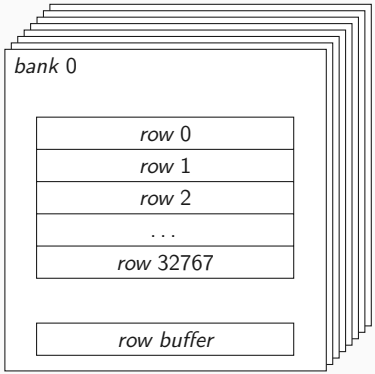


DRAM organization



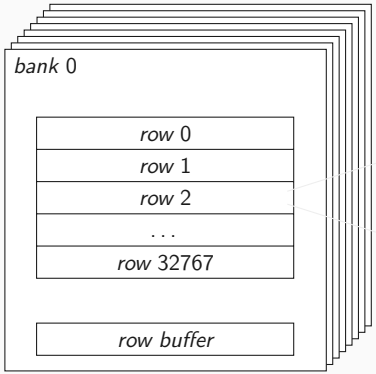
DRAM organization

chip

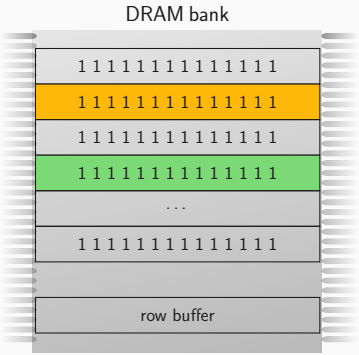


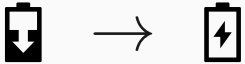
DRAM organization

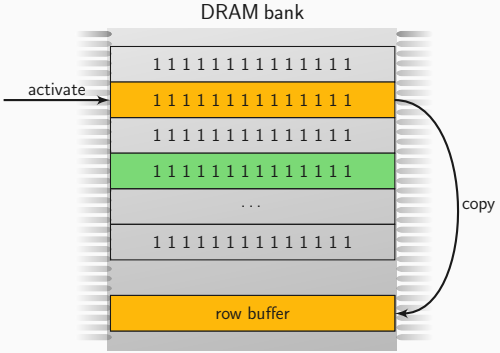
chip

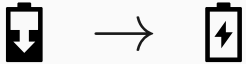


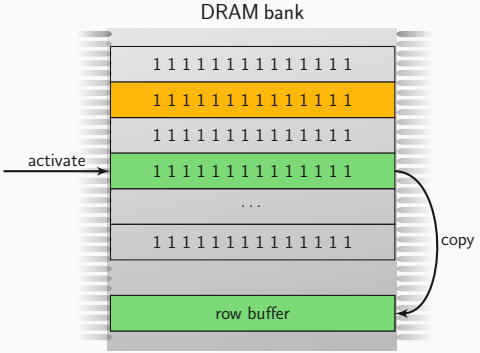
64k cells
1 capacitor,
1 transitor each

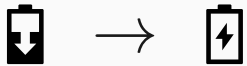

DRAM bank
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
. . .
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
row buffer



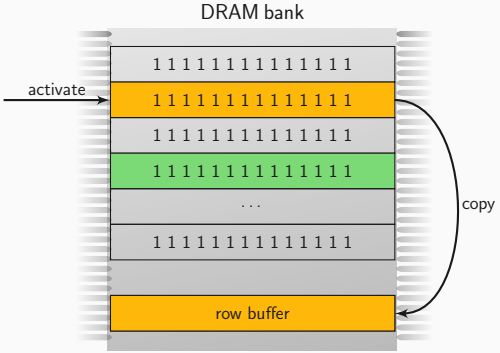

DRAM bank
activate
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
. . .
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
copy
row buffer

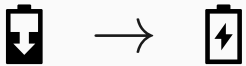




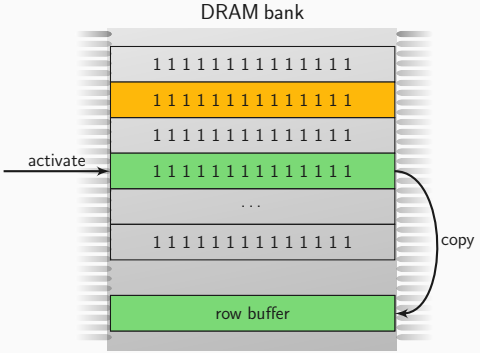


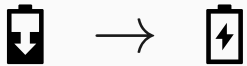
Cells leak faster upon proximate accesses $\rightarrow$ Rowhammer



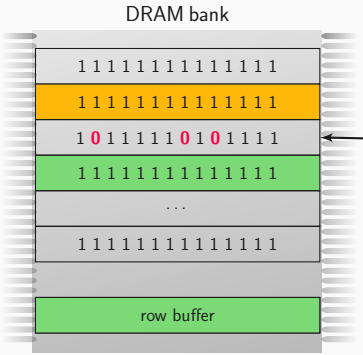


Cells leak faster upon proximate accesses $\rightarrow$ Rowhammer

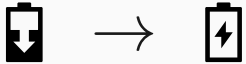




Cells leak faster upon proximate accesses $\rightarrow$ Rowhammer



bit flips in row 2!



Cells leak faster upon proximate accesses → Rowhammer

How widespread is the issue?

How widespread is the issue?

How widespread is the issue?

- 85% affected [Kim+14] (see Figure)

How widespread is the issue?

- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]

How widespread is the issue?

- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]

- First believed to be safe

- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]

- First believed to be safe
- We showed bit flips [Pes+16]

How widespread is the issue?

- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]

- First believed to be safe
- We showed bit flips [Pes+16]
- 67% affected [Lan16]

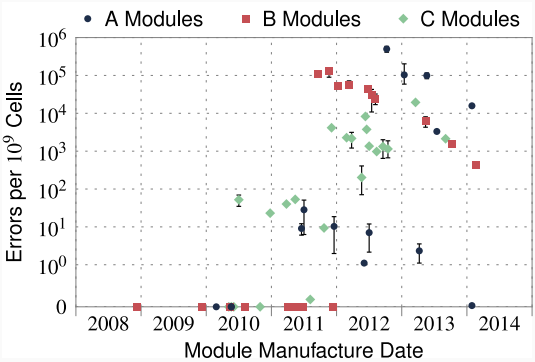
How widespread is the issue?

DDR3

- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]

DDR4

- First believed to be safe
- We showed bit flips [Pes+16]
- 67% affected [Lan16]




BIT FLIPS
BIT FLIPS EVERYWHERE

Requirements

Memory accesses must be

- uncached: reach DRAM
- fast: race against the next row refresh
- targeted: reach specific row

How do we get enough uncached accesses?

- `clflush` instruction → original paper [Kim+14]

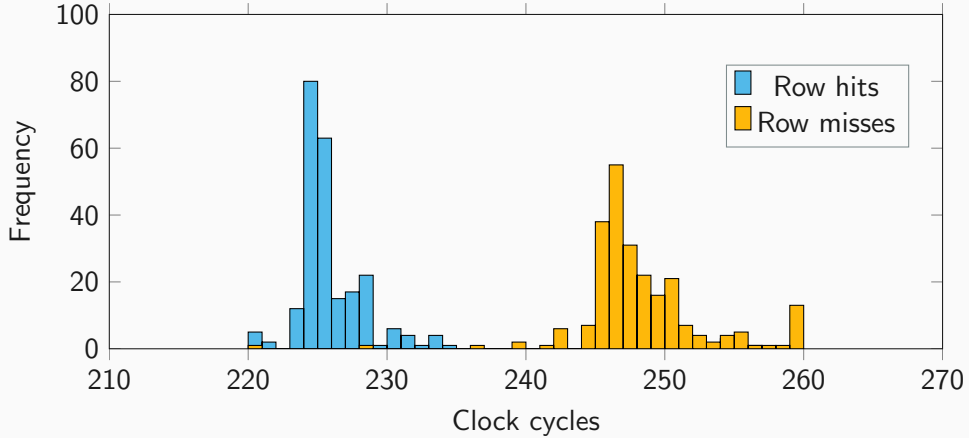
- `clflush` instruction → original paper [Kim+14]
- cache eviction [GMM16; Awe+16]

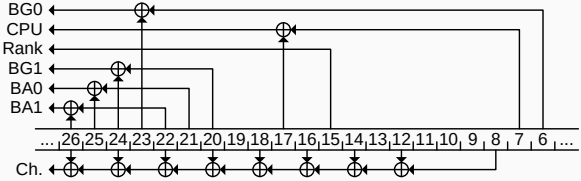
- `clflush` instruction → original paper [Kim+14]
- cache eviction [GMM16; Awe+16]
- non-temporal accesses [QS16]

- `clflush` instruction → original paper [Kim+14]
- cache eviction [GMM16; Awe+16]
- non-temporal accesses [QS16]
- uncached memory [Vee+16]

How do we target accesses?

Timing difference





https://github.com/IAIK/drama

How to exploit random bit flips?

How to exploit random bit flips?

How to exploit random bit flips?

- They are not random → highly reproducible flip pattern!

- They are not random → highly reproducible flip pattern!
  1. Choose a data structure that you can place at arbitrary memory locations

- They are not random → highly reproducible flip pattern!
  1. Choose a data structure that you can place at arbitrary memory locations
  2. Scan for "good" flips

How to exploit random bit flips?

- They are not random → highly reproducible flip pattern!
  1. Choose a data structure that you can place at arbitrary memory locations
  2. Scan for “good” flips
  3. Place data structure there

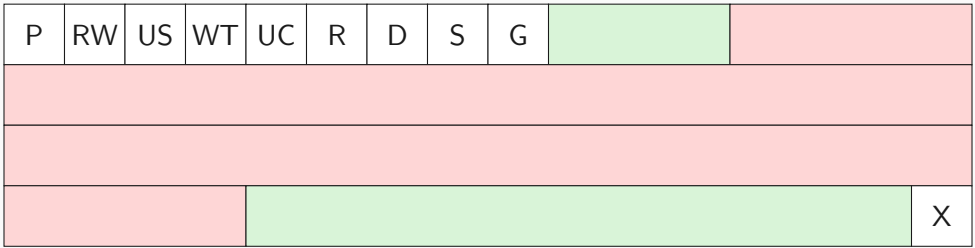
How to exploit random bit flips?

- They are not random → highly reproducible flip pattern!
  1. Choose a data structure that you can place at arbitrary memory locations
  2. Scan for “good” flips
  3. Place data structure there
  4. Trigger bit flip again

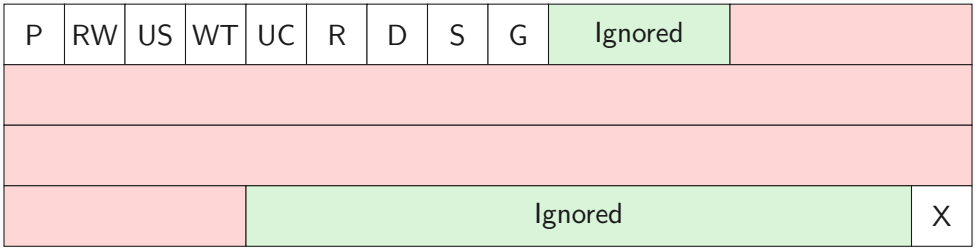
How to exploit random bit flips?

- They are not random → highly reproducible flip pattern!
  1. Choose a data structure that you can place at arbitrary memory locations
  2. Scan for "good" flips
  3. Place data structure there
  4. Trigger bit flip again
- Alternatively: Build a PUF [Ana+18]

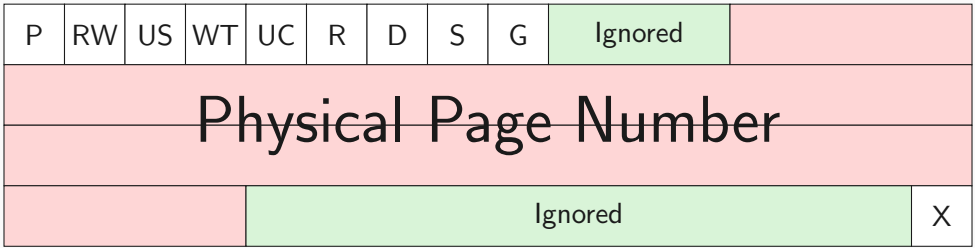
Page Table Entries

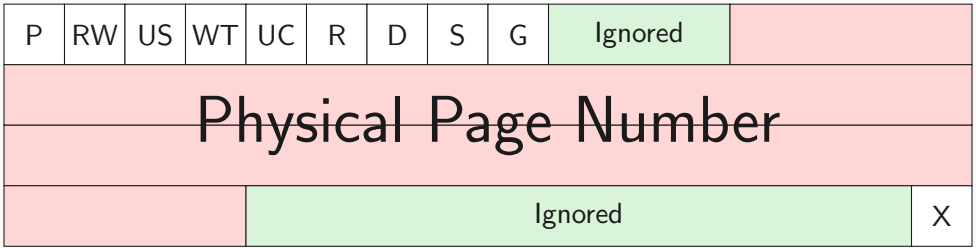


Page Table Entries



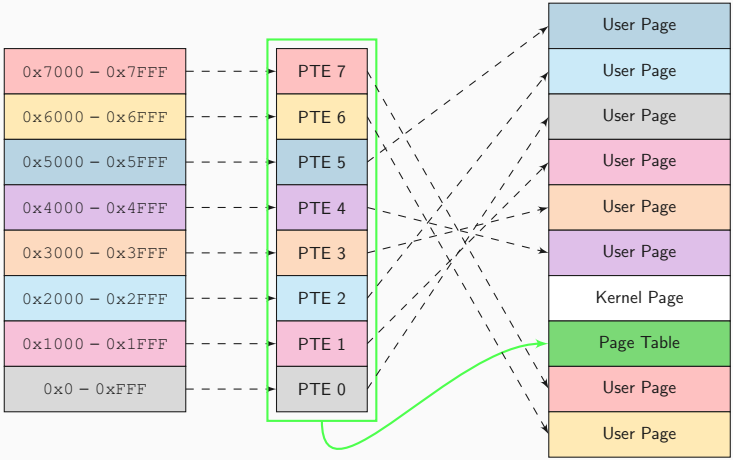
Page Table Entries

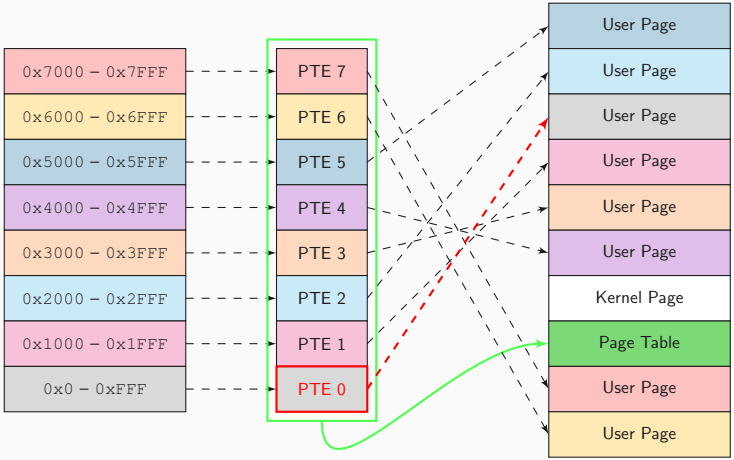




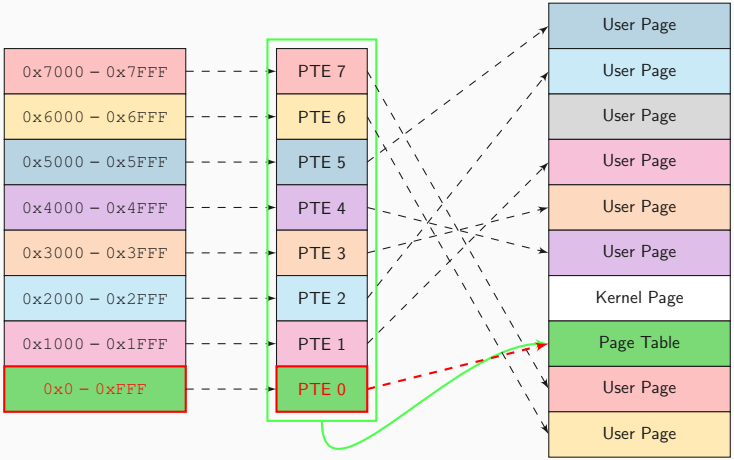
Each 4 KB page table consists of 512 such entries

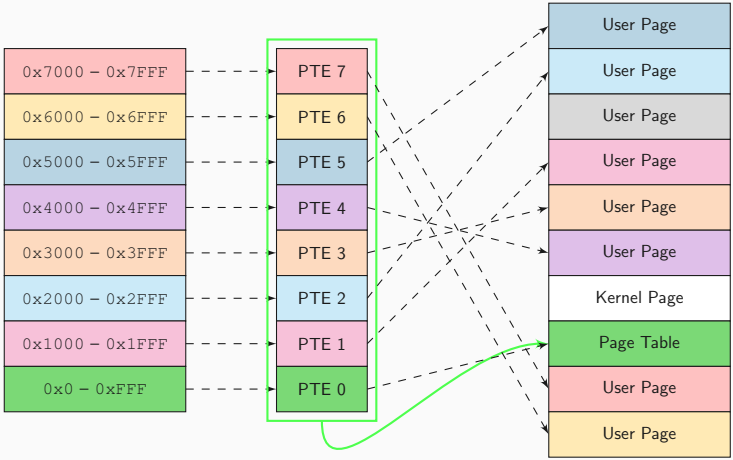
Page Table Manipulation

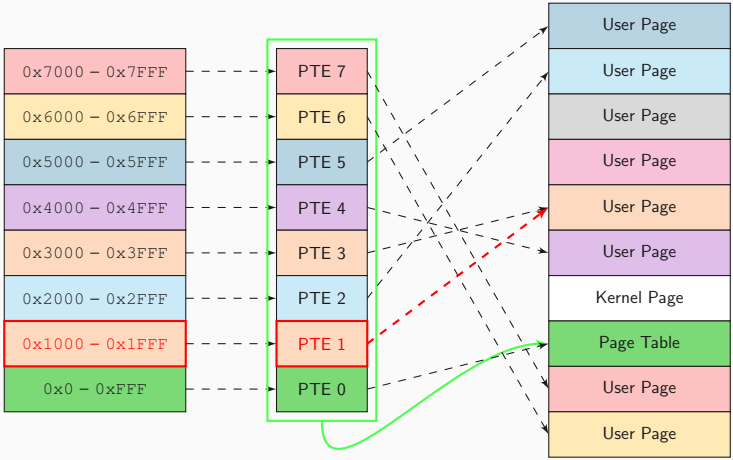



0x7000 – 0x7FFF
0x6000 – 0x6FFF
0x5000 – 0x5FFF
0x4000 – 0x4FFF
0x3000 – 0x3FFF
0x2000 – 0x2FFF
0x1000 – 0x1FFF
0x0 – 0xFFF
PTE 7
PTE 6
PTE 5
PTE 4
PTE 3
PTE 2
PTE 1
PTE 0
User Page
User Page
User Page
User Page
User Page
User Page
Kernel Page
Page Table
User Page
User Page

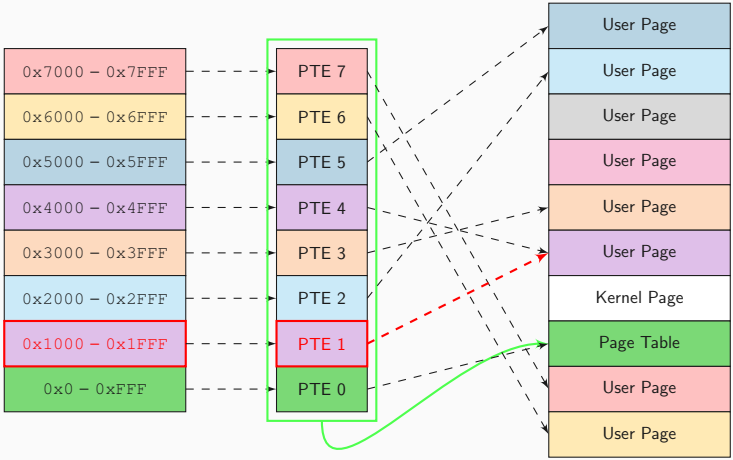
Page Table Manipulation

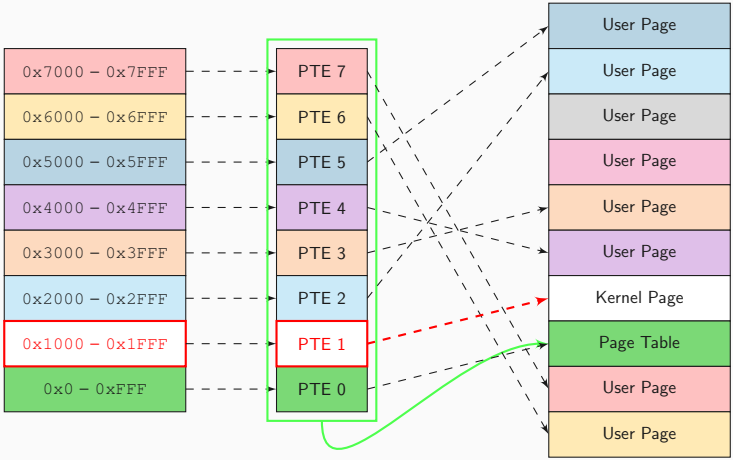


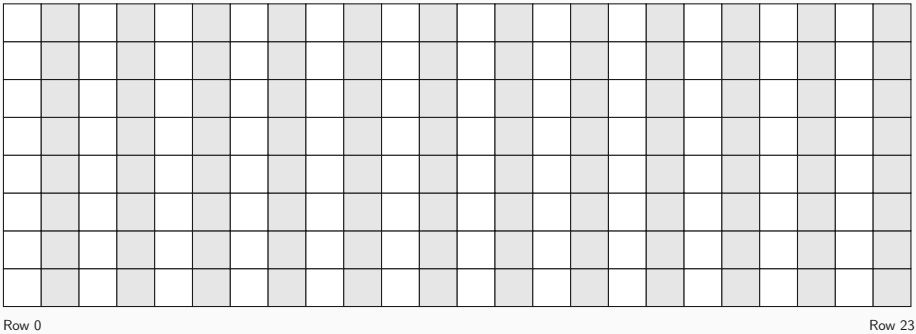

0x7000 – 0x7FFF
0x6000 – 0x6FFF
0x5000 – 0x5FFF
0x4000 – 0x4FFF
0x3000 – 0x3FFF
0x2000 – 0x2FFF
0x1000 – 0x1FFF
0x0 – 0xFFF
PTE 7
PTE 6
PTE 5
PTE 4
PTE 3
PTE 2
PTE 1
PTE 0
User Page
User Page
User Page
User Page
User Page
User Page
Kernel Page
Page Table
User Page
User Page


0x7000 – 0x7FFF
0x6000 – 0x6FFF
0x5000 – 0x5FFF
0x4000 – 0x4FFF
0x3000 – 0x3FFF
0x2000 – 0x2FFF
0x1000 – 0x1FFF
0x0 – 0xFFF
PTE 7
PTE 6
PTE 5
PTE 4
PTE 3
PTE 2
PTE 1
PTE 0
User Page
User Page
User Page
User Page
User Page
User Page
Kernel Page
Page Table
User Page
User Page

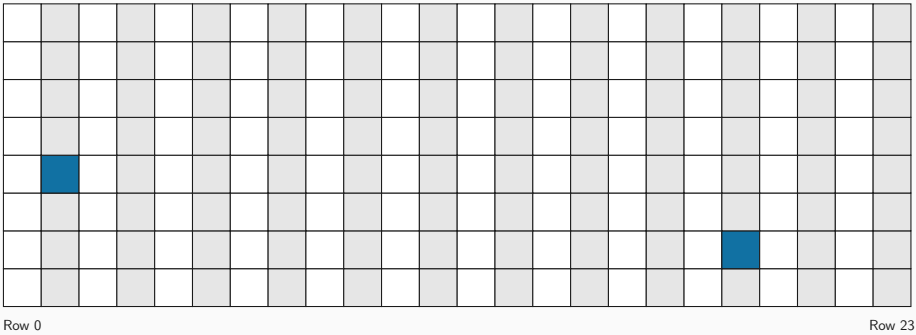
Page Table Manipulation



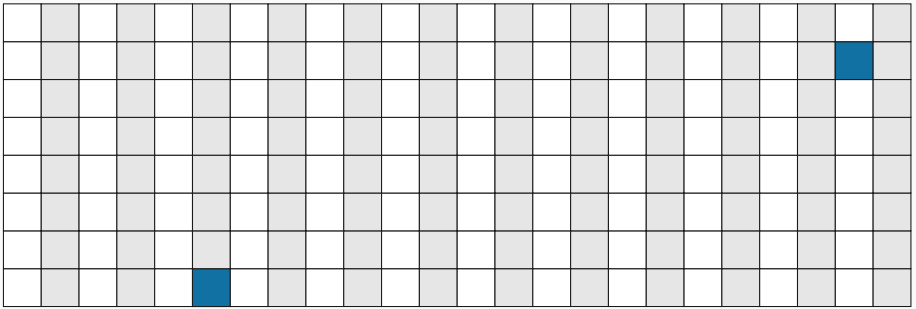

0x7000 – 0x7FFF
0x6000 – 0x6FFF
0x5000 – 0x5FFF
0x4000 – 0x4FFF
0x3000 – 0x3FFF
0x2000 – 0x2FFF
0x1000 – 0x1FFF
0x0 – 0xFFF
PTE 7
PTE 6
PTE 5
PTE 4
PTE 3
PTE 2
PTE 1
PTE 0
User Page
User Page
User Page
User Page
User Page
User Page
Kernel Page
Page Table
User Page
User Page



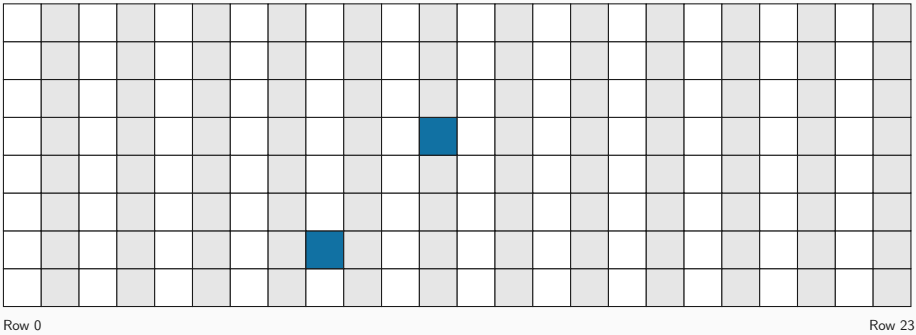
Hammering memory locations in different rows



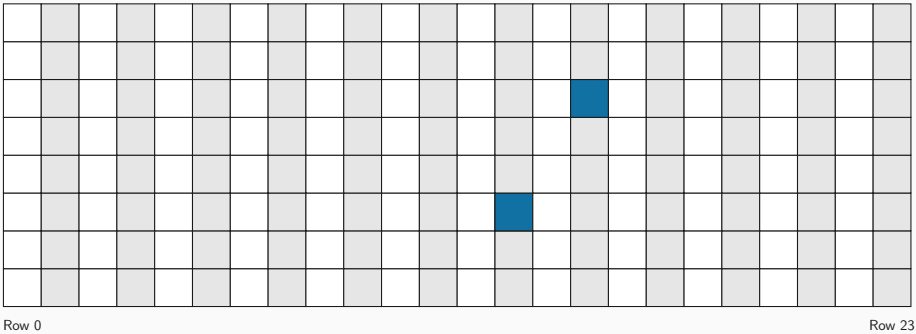
Hammering memory locations in different rows



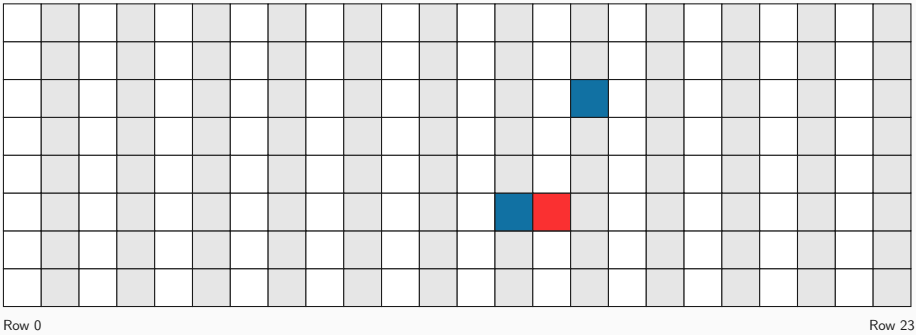
Hammering memory locations in different rows



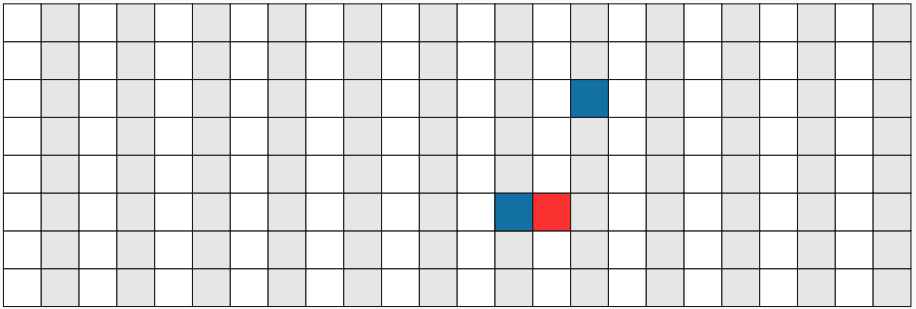
Hammering memory locations in different rows

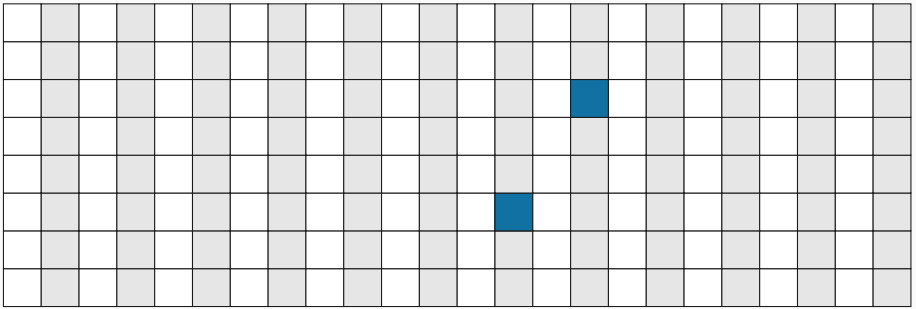


Hammering memory locations in different rows

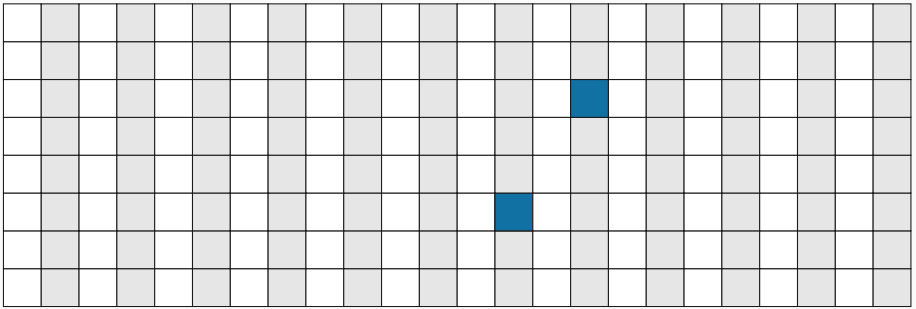


Hammering memory locations in different rows

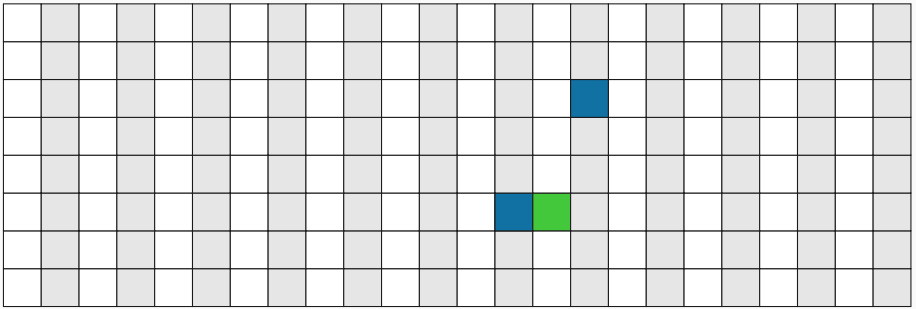

Row 0
Row 23

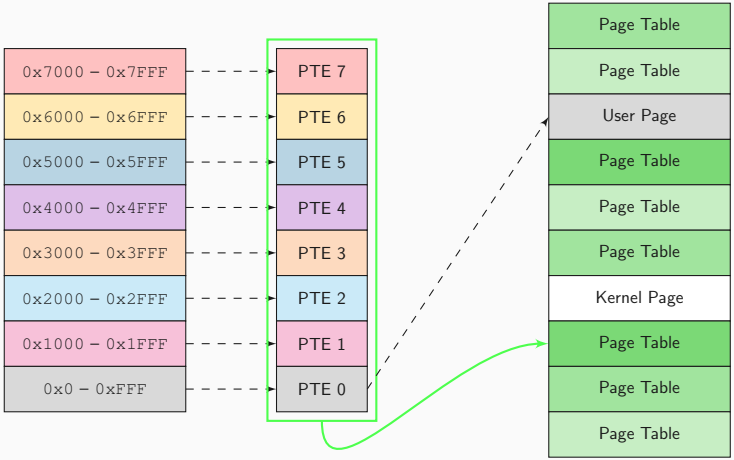

Row 0
Row 23

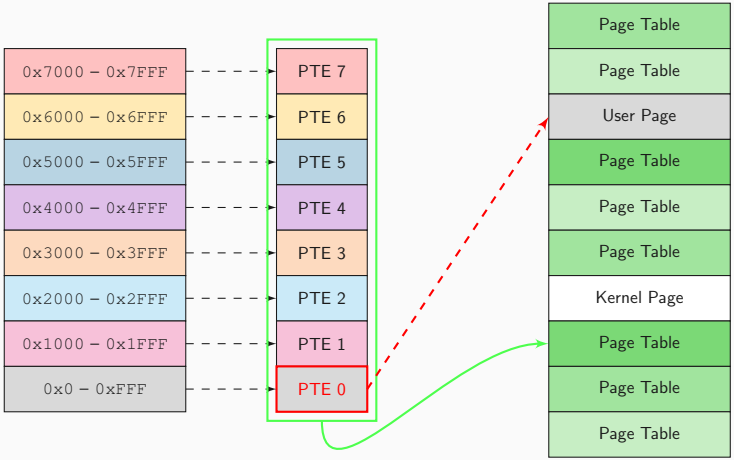
Fill all remaining memory with page tables



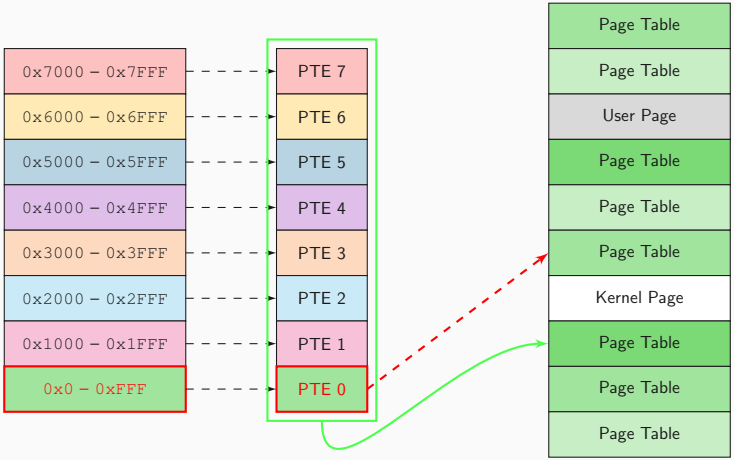
Fill all remaining memory with page tables




0x7000 – 0x7FFF
0x6000 – 0x6FFF
0x5000 – 0x5FFF
0x4000 – 0x4FFF
0x3000 – 0x3FFF
0x2000 – 0x2FFF
0x1000 – 0x1FFF
0x0 – 0xFFF
PTE 7
PTE 6
PTE 5
PTE 4
PTE 3
PTE 2
PTE 1
PTE 0
Page Table
Page Table
User Page
Page Table
Page Table
Page Table
Kernel Page
Page Table
Page Table
Page Table

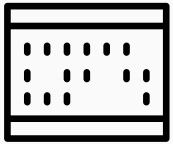

0x7000 – 0x7FFF
0x6000 – 0x6FFF
0x5000 – 0x5FFF
0x4000 – 0x4FFF
0x3000 – 0x3FFF
0x2000 – 0x2FFF
0x1000 – 0x1FFF
0x0 – 0xFFF
PTE 7
PTE 6
PTE 5
PTE 4
PTE 3
PTE 2
PTE 1
PTE 0
Page Table
Page Table
User Page
Page Table
Page Table
Page Table
Kernel Page
Page Table
Page Table
Page Table

Page Table Manipulation

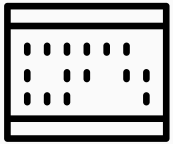


Flipping Page Table PPN bits

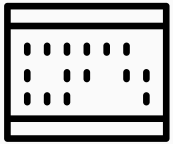
Flipping Page Table PPN bits



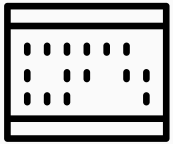
Flipping Page Table PPN bits



Flipping Page Table PPN bits

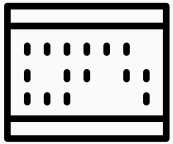


- Idea from [SD15]

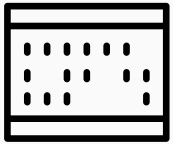


- Idea from [SD15]
- Same idea applied in several other works:

Flipping Page Table PPN bits

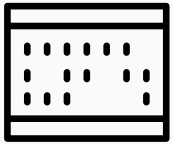


- Idea from [SD15]
- Same idea applied in several other works:
  - Rowhammer.js [GMM16]



- Idea from [SD15]
- Same idea applied in several other works:
  - Rowhammer.js [GMM16]
  - One bit flips, one cloud flops [Xia+16]

Flipping Page Table PPN bits



- Idea from [SD15]
- Same idea applied in several other works:
  - Rowhammer.js [GMM16]
  - One bit flips, one cloud flops [Xia+16]
  - Drammer [Vee+16]

- Scan entire physical memory (very fast) and:

- Scan entire physical memory (very fast) and:
  - Modify binary pages executed in root privileges [Xia+16]

- Scan entire physical memory (very fast) and:
  - Modify binary pages executed in root privileges [Xia+16]
  - Modify credential structs [Vee+16]

- Scan entire physical memory (very fast) and:
  - Modify binary pages executed in root privileges [Xia+16]
  - Modify credential structs [Vee+16]
  - Read keys [Xia+16]

- Scan entire physical memory (very fast) and:
  - Modify binary pages executed in root privileges [Xia+16]
  - Modify credential structs [Vee+16]
  - Read keys [Xia+16]
  - Corrupt signatures [BM16; Pod+18]

- Scan entire physical memory (very fast) and:
  - Modify binary pages executed in root privileges [Xia+16]
  - Modify credential structs [Vee+16]
  - Read keys [Xia+16]
  - Corrupt signatures [BM16; Pod+18]
  - Modify certificates

- Scan entire physical memory (very fast) and:
  - Modify binary pages executed in root privileges [Xia+16]
  - Modify credential structs [Vee+16]
  - Read keys [Xia+16]
  - Corrupt signatures [BM16; Pod+18]
  - Modify certificates
  - Configurations

- Scan entire physical memory (very fast) and:
  - Modify binary pages executed in root privileges [Xia+16]
  - Modify credential structs [Vee+16]
  - Read keys [Xia+16]
  - Corrupt signatures [BM16; Pod+18]
  - Modify certificates
  - Configurations
  - etc.

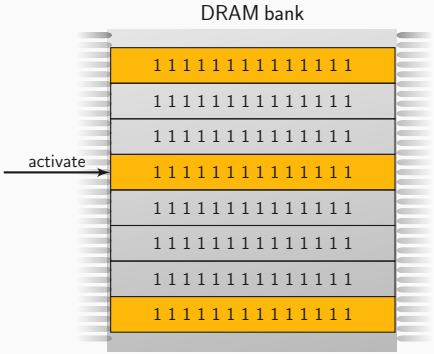
- Scan entire physical memory (very fast) and:
  - Modify binary pages executed in root privileges [Xia+16]
  - Modify credential structs [Vee+16]
  - Read keys [Xia+16]
  - Corrupt signatures [BM16; Pod+18]
  - Modify certificates
  - Configurations
  - etc.
- pages are pretty unique: 32768 bits per page

- There are two different hammering techniques

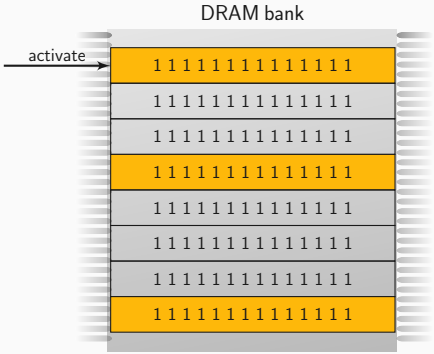
- There are two different hammering techniques
- #1: Hammer one row next to victim row and other random rows

- There are two different hammering techniques
- #1: Hammer one row next to victim row and other random rows
- #2: Hammer two rows neighboring victim row

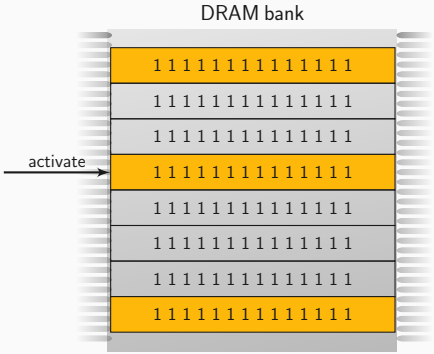
#1 - Single-sided hammering



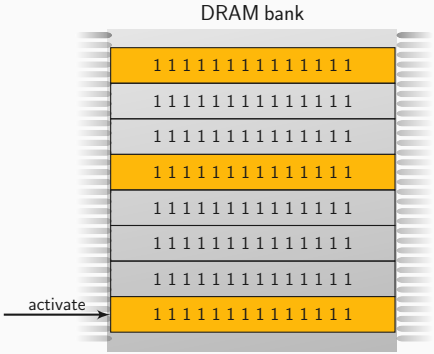
#1 - Single-sided hammering



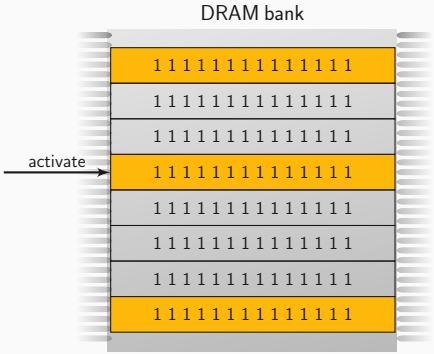
#1 - Single-sided hammering



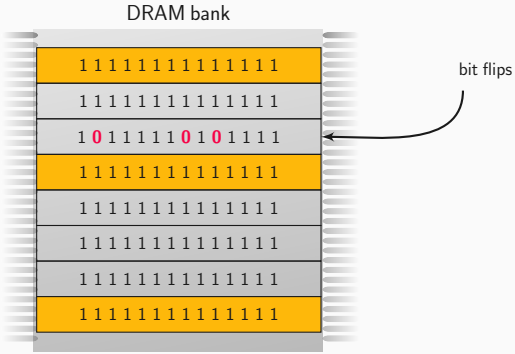
#1 - Single-sided hammering



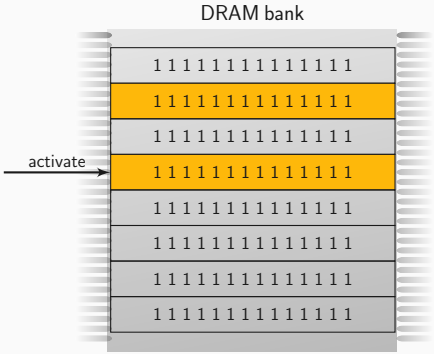
#1 - Single-sided hammering



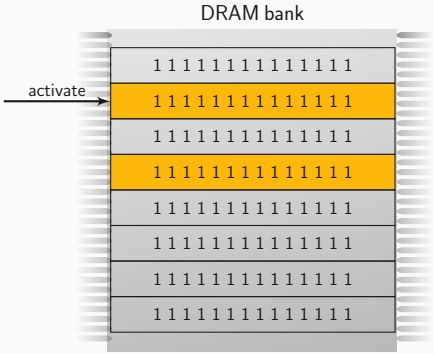
#1 - Single-sided hammering



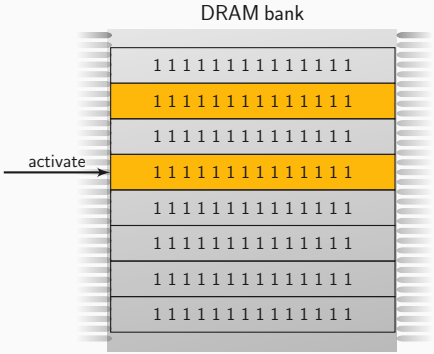
#2 - Double-sided hammering



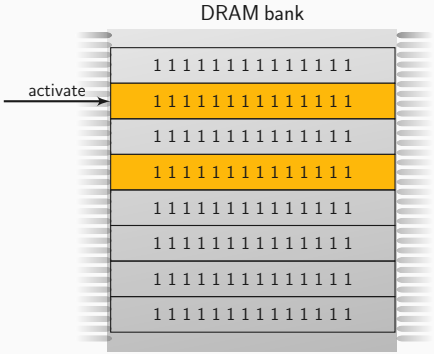
#2 - Double-sided hammering



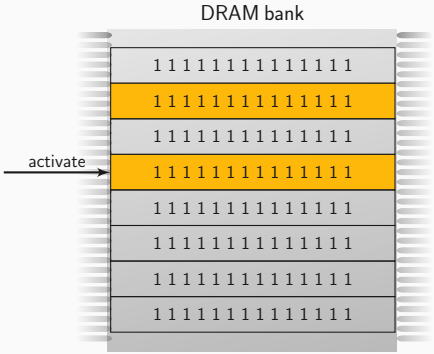
#2 - Double-sided hammering



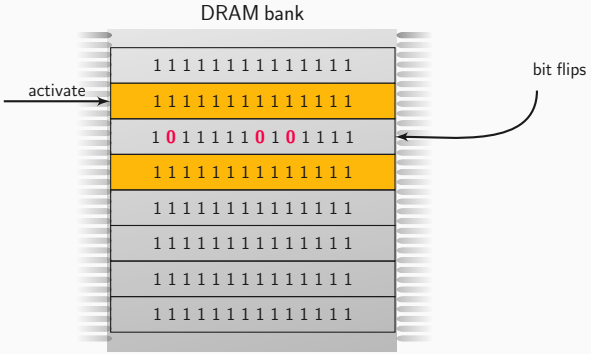
#2 - Double-sided hammering



#2 - Double-sided hammering



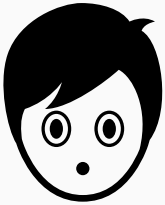
#2 - Double-sided hammering



HAMMERING
TWO ROWS

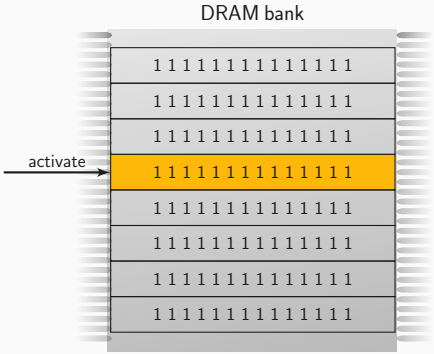
HAMMERING TWO ROWS

HAMMERING A SINGLE ROW

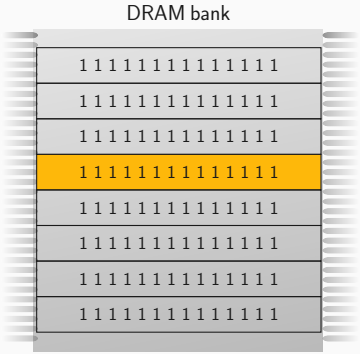


- There are three different hammering techniques
- #1: Hammer one row next to victim row and other random rows
- #2: Hammer two rows neighboring victim row
- #3: Hammer only one row next to victim row

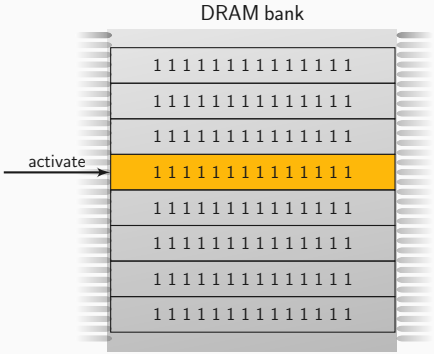
#3 - One-location hammering



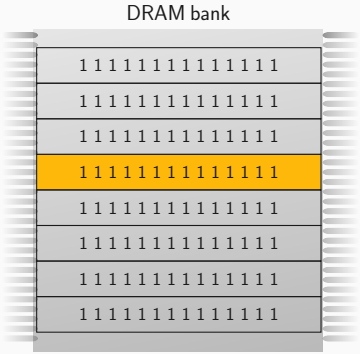
#3 - One-location hammering



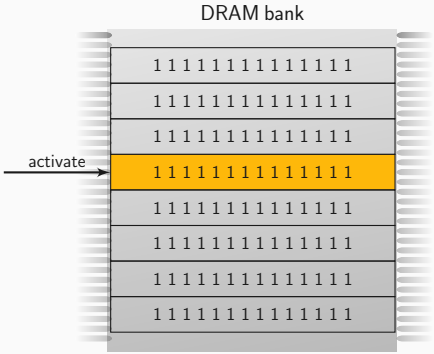
#3 - One-location hammering



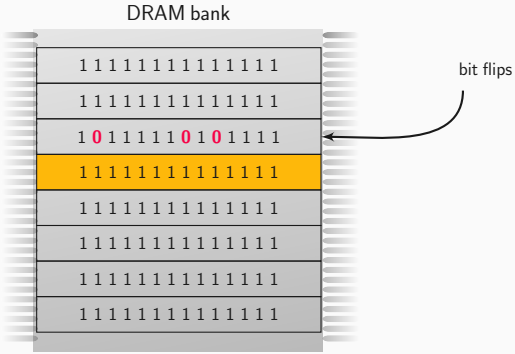
#3 - One-location hammering

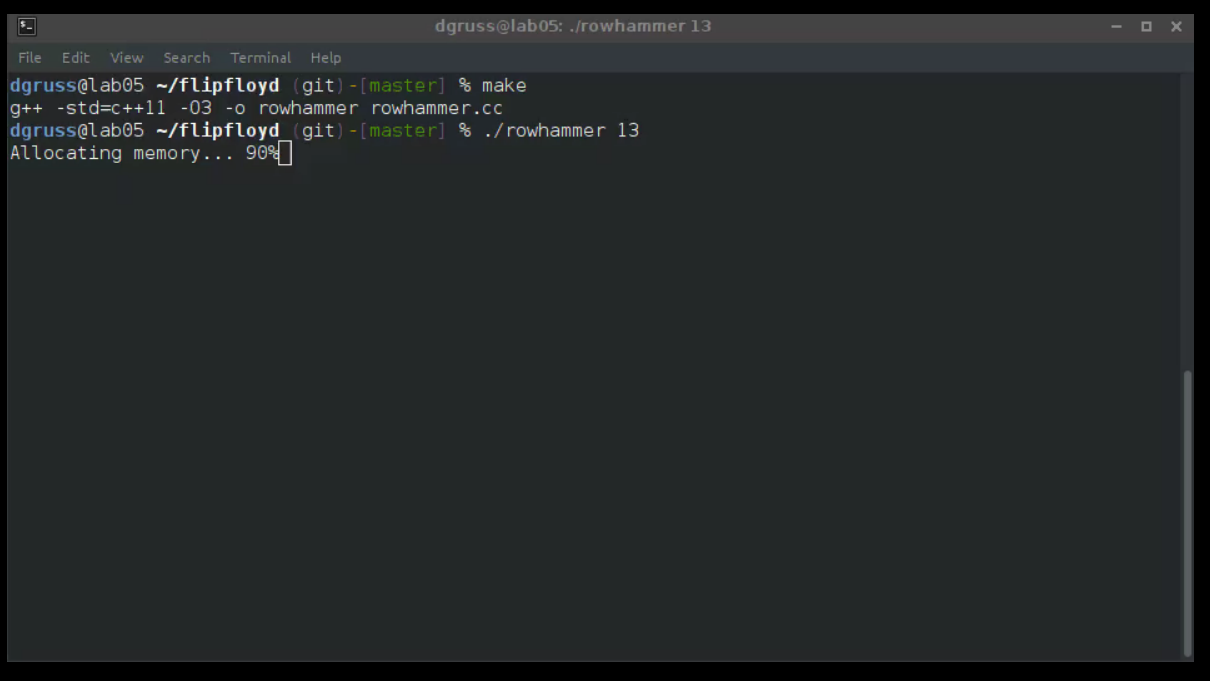


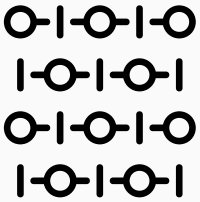
#3 - One-location hammering



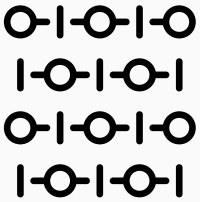
#3 - One-location hammering



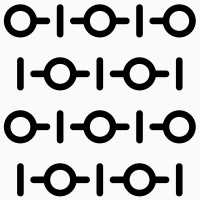

dgruss@lab05: ./rowhammer 13
File Edit View Search Terminal Help
dgruss@lab05 ~/flipfloyd (git)-[master] % make
g++ -std=c++11 -O3 -o rowhammer rowhammer.cc
dgruss@lab05 ~/flipfloyd (git)-[master] % ./rowhammer 13
Allocating memory... 90%



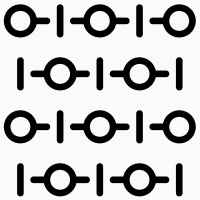
- **Open-page policy**: Keep row opened and buffered



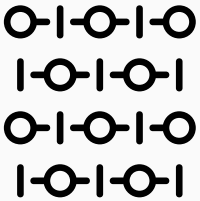
- **Open-page policy**: Keep row opened and buffered
  - Low latency for subsequent accesses to same row
  - High latency for accesses to any other row



- **Open-page policy**: Keep row opened and buffered
  - Low latency for subsequent accesses to same row
  - High latency for accesses to any other row
- **Close-page policy**: Immediately close row, ready to open a new row



- **Open-page policy**: Keep row opened and buffered
  - Low latency for subsequent accesses to same row
  - High latency for accesses to any other row
- **Close-page policy**: Immediately close row, ready to open a new row
  - Medium latency for accesses to any row



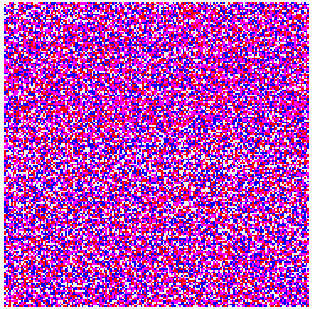
- **Open-page policy**: Keep row opened and buffered
  - Low latency for subsequent accesses to same row
  - High latency for accesses to any other row
- **Close-page policy**: Immediately close row, ready to open a new row
  - Medium latency for accesses to any row
  - Perform better on multi-core systems [Dav+11]

- Policies that preemptively close rows, would allow one-location hammering

- Policies that preemptively close rows, would allow one-location hammering
- We observed close-page policies on desktop computers

- Policies that preemptively close rows, would allow one-location hammering
- We observed close-page policies on desktop computers
- Mobile devices (e.g., laptops) seem to use mostly open-page policies

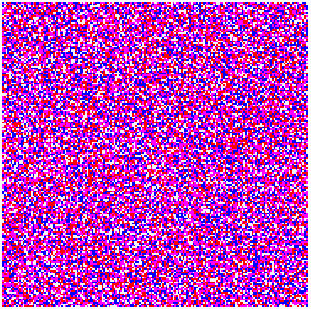
How well does it work?



Double-sided

77.0 % bit offsets

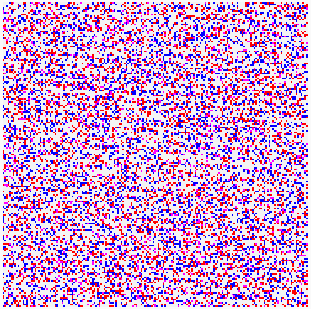
51.7 % 0→1 bit flips



Single-sided

78.5 % bit offsets

54.1 % 0→1 bit flips



One-location

36.5 % bit offsets

51.6 % 0→1 bit flips

What if we cannot target kernel pages?

What if we cannot target kernel pages?

Opcode Flipping

- Many applications perform actions as root

- Many applications perform actions as root
- They can be used by unprivileged users as well

- Many applications perform actions as root
- They can be used by unprivileged users as well
- Implicitly: e.g., `ping` or `mount`

- Many applications perform actions as root
- They can be used by unprivileged users as well
- Implicitly: e.g., `ping` or `mount`
- Explicitly: `sudo`

- Many applications perform actions as root
- They can be used by unprivileged users as well
- Implicitly: e.g., `ping` or `mount`
- Explicitly: `sudo`
- Target `sudo` (easy to exploit)


JE
HLT
0 1 1 1 0 1 0 0
1 1 1 1 0 1 0 0


JE
0 1 1 1 0 1 0 0
XORB
0 0 1 1 0 1 0 0


JE
0 1 1 1 0 1 0 0
PUSHQ
0 1 0 1 0 1 0 0


JE
0 1 1 1 0 1 0 0
<prefix>
0 1 1 0 0 1 0 0


JE
0 1 1 1 0 1 0 0
JL
0 1 1 1 1 1 0 0


JE
0 1 1 1 0 1 0 0
JO
0 1 1 1 0 0 0 0

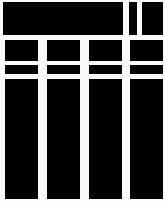
Opcode Flipping - Conditional Jump


JE
0 1 1 1 0 1 0 0
JNE
0 1 1 1 0 1 0 1

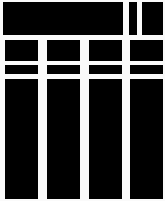
How to get the target virtual page to the target physical location?

How to get the target virtual page to the target physical location?

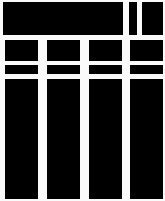
Memory Waylaying



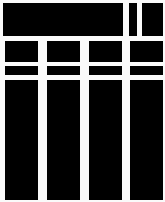
- If a binary is loaded the first time, it is loaded to the memory



- If a binary is loaded the first time, it is loaded to the memory
- It stays in memory (in the page cache) even after execution



- If a binary is loaded the first time, it is loaded to the memory
- It stays in memory (in the page cache) even after execution
- Only evicted if page cache is full

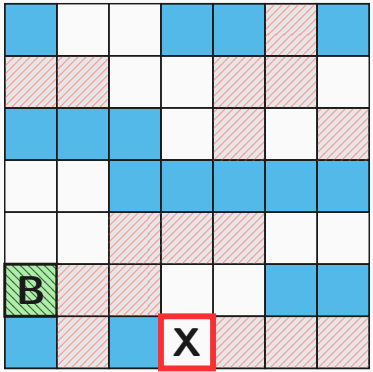


- If a binary is loaded the first time, it is loaded to the memory
- It stays in memory (in the page cache) even after execution
- Only evicted if page cache is full
- Page cache is huge - usually all unused memory

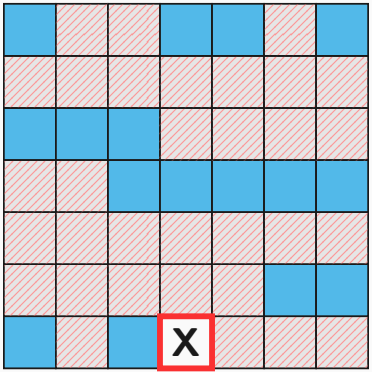
MEMORY WAYLAYING

Wait for the right moment, and then hit it with a bit flip!

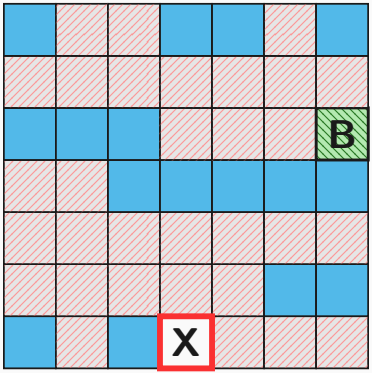
(1) Start



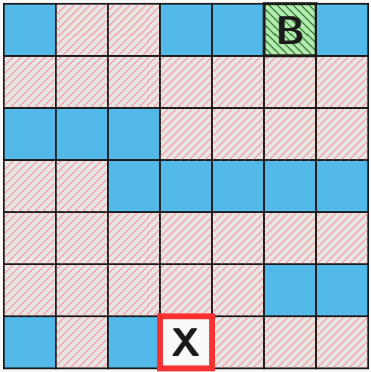
(2) Evict Page Cache



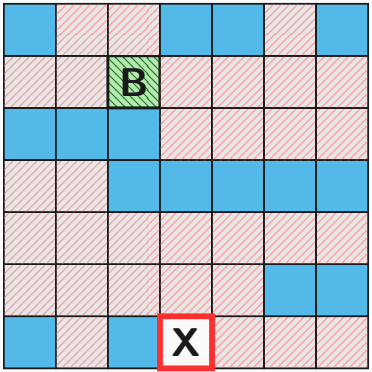
(3) Access Binary



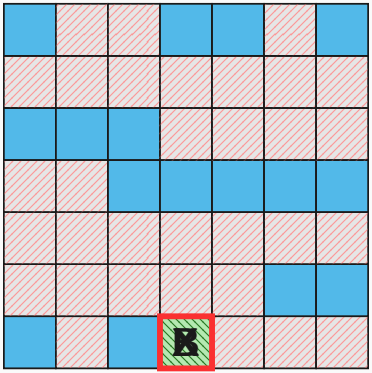
(4) Evict + Access



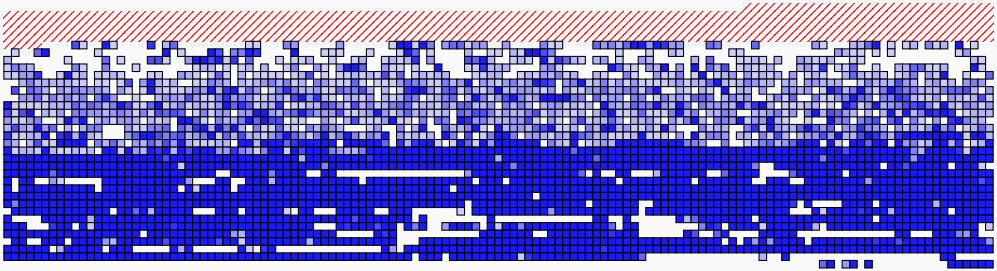
(5) Evict + Access



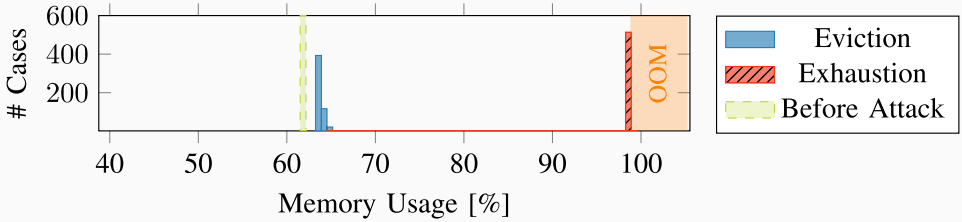
(6) Stop if target reached



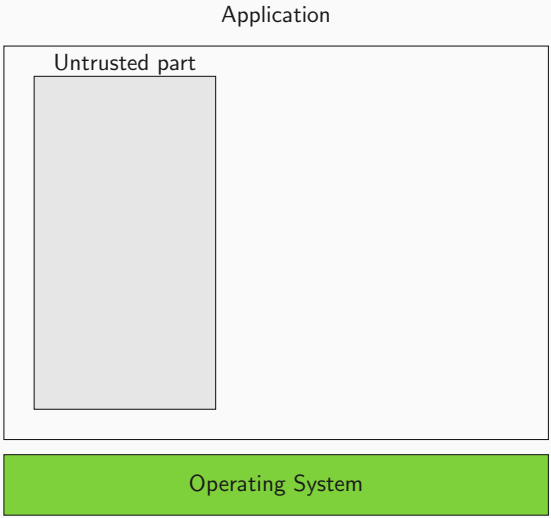
- New pages cover most of the physical memory

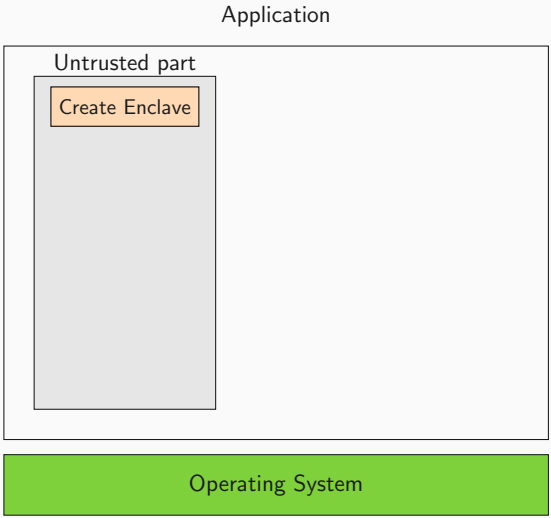


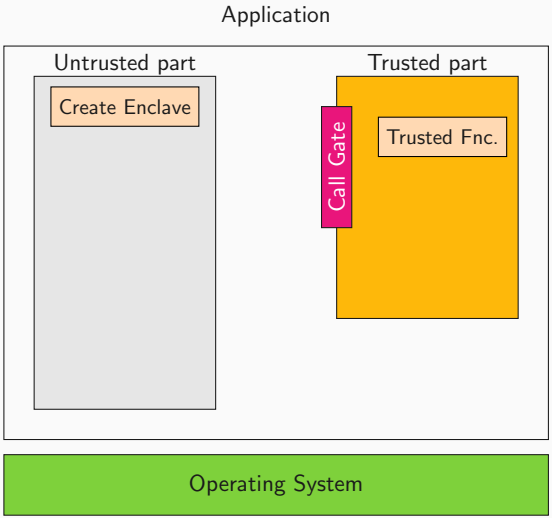
- Great advantage over memory massaging: only negligible memory footprint

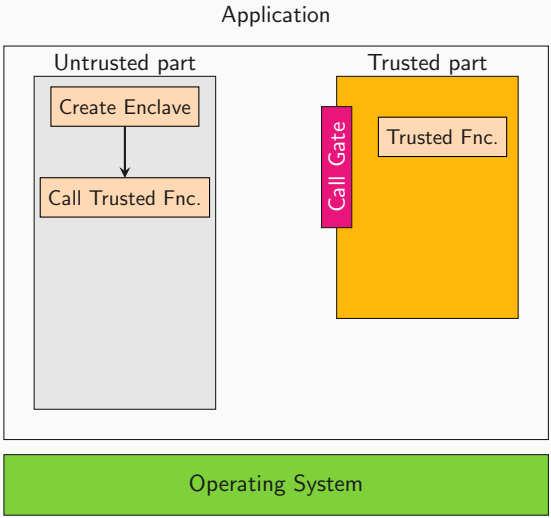


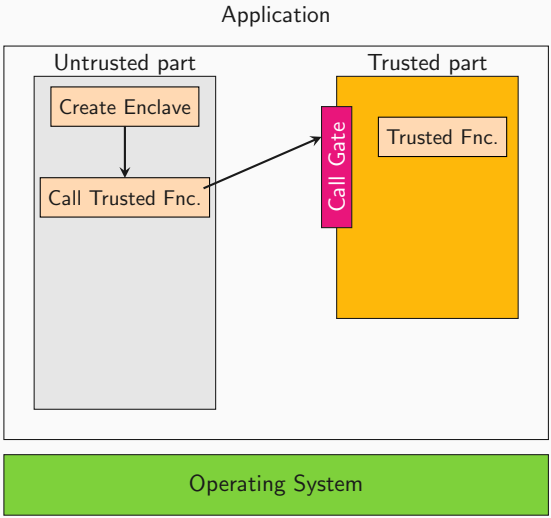
Rowhammer + SGX = Cheap Denial of Service

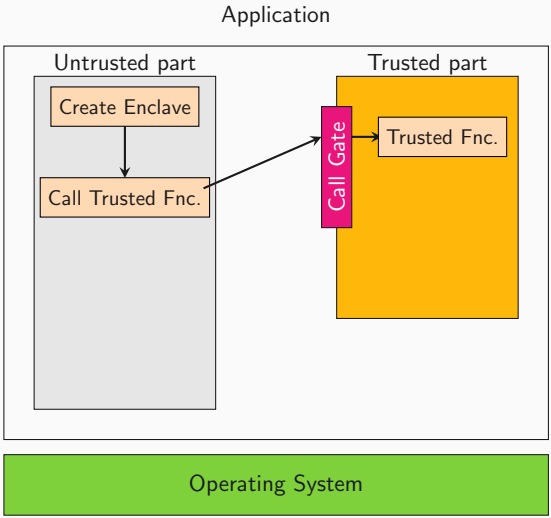

Application
Untrusted part
Operating System

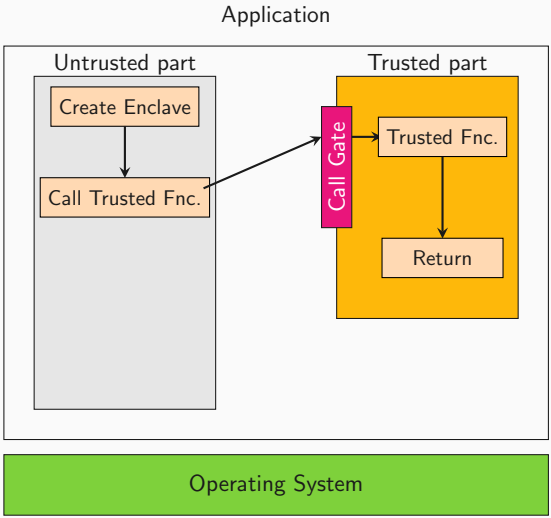

Application
Untrusted part
Create Enclave
Operating System

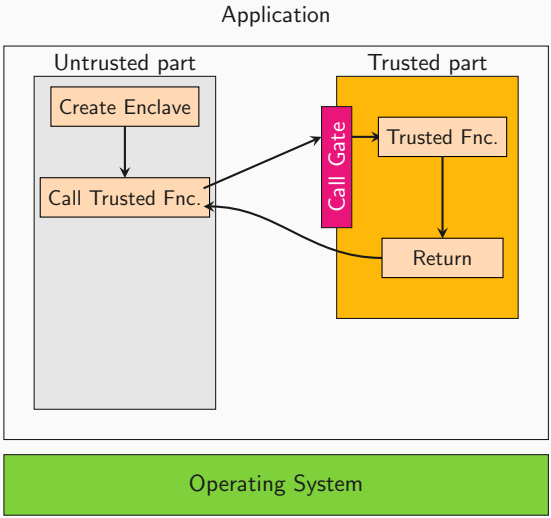

Application
Untrusted part
Create Enclave
Trusted part
Call Gate
Trusted Fnc.
Operating System

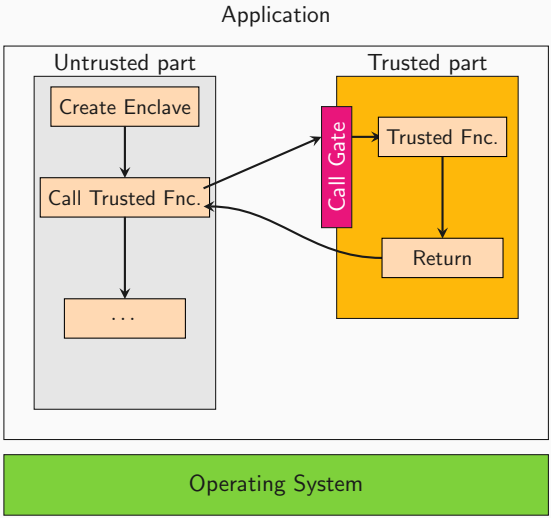

Application
Untrusted part
Create Enclave
Call Trusted Fnc.
Trusted part
Call Gate
Trusted Fnc.
Operating System

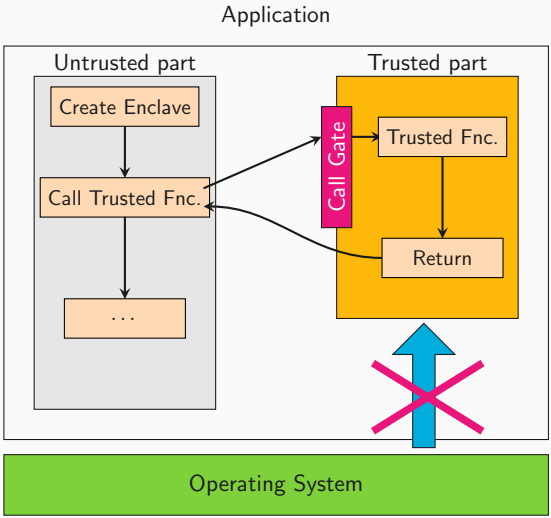

Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
Call Gate
Trusted Fnc.
Operating System


Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
Call Gate
Trusted Fnc.
Operating System

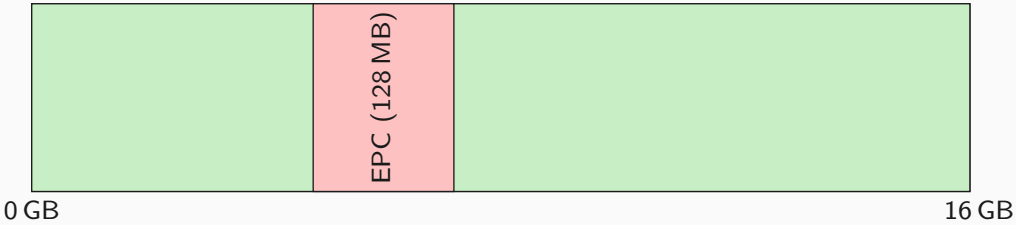

Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
Call Gate
Trusted Fnc.
Return
Operating System

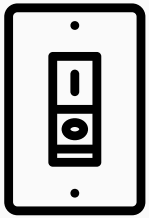

Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
Call Gate
Trusted Fnc.
Return
Operating System


Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
...
Call Gate
Trusted Fnc.
Return
Operating System


Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
...
Call Gate
Trusted Fnc.
Return
Operating System

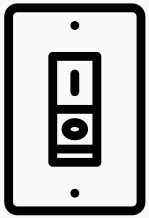

0 GB
16 GB


EPC (128 MB)
0 GB
16 GB

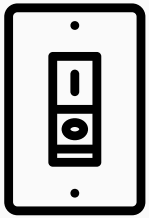


- What happens if a bit flips in the EPC?

Bit Flips in the EPC

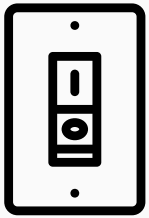


- What happens if a bit flips in the EPC?
- Integrity check will fail!



- What happens if a bit flips in the EPC?
- Integrity check will fail!

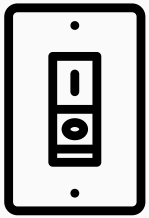
→ Locks up the memory controller



- What happens if a bit flips in the EPC?
- Integrity check will fail!

→ Locks up the memory controller

→ Not a single further memory access!

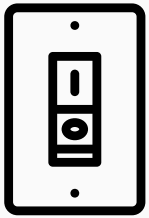


- What happens if a bit flips in the EPC?
- Integrity check will fail!

→ Locks up the memory controller

→ Not a single further memory access!

→ System halts immediately



- What happens if a bit flips in the EPC?
- Integrity check will fail!

→ Locks up the memory controller

→ Not a single further memory access!

→ System halts immediately


SOUNDS UNSAFE?
IT IS UNSAFE!
imgflip.com

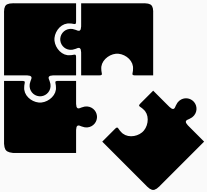
- If a malicious enclave induces a bit flip, ...

- If a malicious enclave induces a bit flip, ...
- ...the entire machine halts

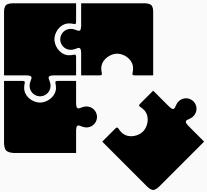
- If a malicious enclave induces a bit flip, ...
- ...the entire machine halts
- ...including co-located tenants

- If a malicious enclave induces a bit flip, ...
- ...the entire machine halts
- ...including co-located tenants
- Denial-of-Service Attacks in the Cloud [Gru+18; Jan+17]

SGX + One-location Hammering + Opcode Flipping = Undetectable Exploit

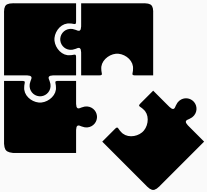


- SGX protects software from malicious environments



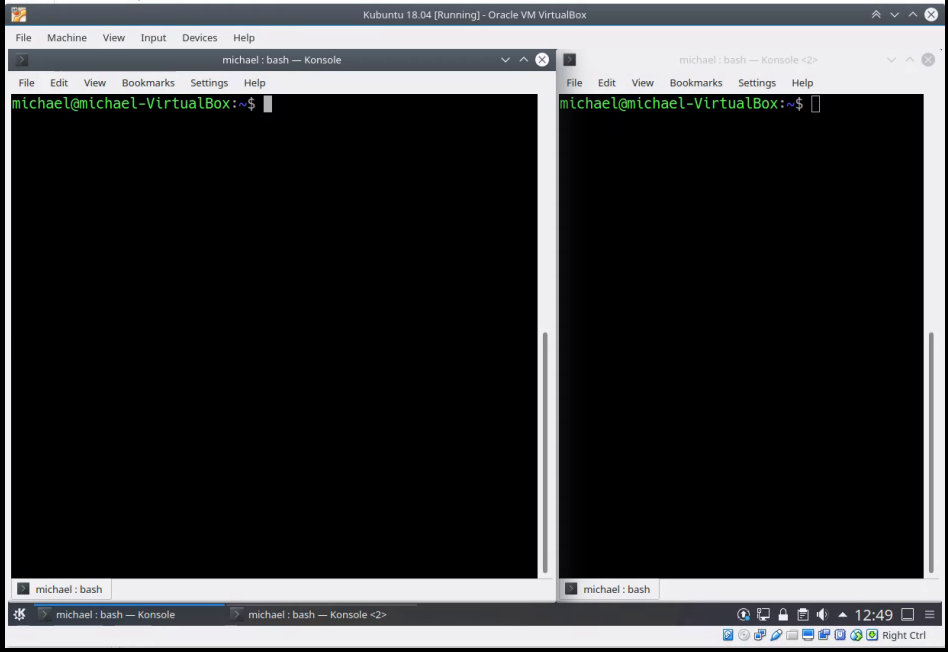
- SGX protects software from malicious environments
- Thwarts static and dynamic (= performance counters) analysis

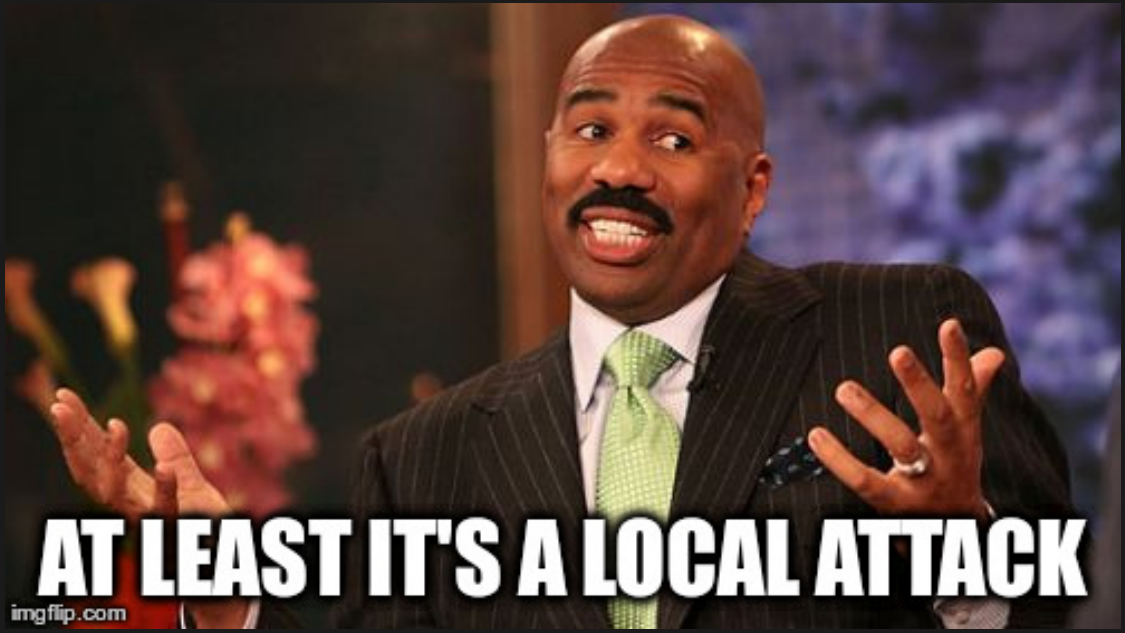
(Ab)using SGX Protection



- SGX protects software from malicious environments
- Thwarts static and dynamic (= performance counters) analysis
- Hammering from SGX defeats countermeasures relying on this


STEALTH LEVEL: EXPERT


Kubuntu 18.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
michael : bash — Konsole
File Edit View Bookmarks Settings Help
michael@michael-VirtualBox:~$
michael : bash — Konsole <2>
File Edit View Bookmarks Settings Help
michael@michael-VirtualBox:~$
michael : bash
michael : bash
michael : bash — Konsole
michael : bash — Konsole <2>
12:49
Right Ctrl


AT LEAST IT'S A LOCAL ATTACK
imgflip.com

Just comparing some performance numbers...

Just comparing some performance numbers...

Just comparing some performance numbers...

- $\geq$ 43 000 hammering attempts (within 64 ms) for a bit flip [GMM16]

Just comparing some performance numbers...

- $\geq$ 43 000 hammering attempts (within 64 ms) for a bit flip [GMM16]

$=$ 671 875 accesses per second

Just comparing some performance numbers...

- $\geq$ 43 000 hammering attempts (within 64 ms) for a bit flip [GMM16]

$=$ 671 875 accesses per second

- Network packets access memory location up to 6 times (depending on kernel)

Just comparing some performance numbers...

- $\geq$ 43 000 hammering attempts (within 64 ms) for a bit flip [GMM16]

$=$ 671 875 accesses per second

- Network packets access memory location up to 6 times (depending on kernel)

$\rightarrow$ 111 979 packets per second

Just comparing some performance numbers...

- $\geq$ 43 000 hammering attempts (within 64 ms) for a bit flip [GMM16]

$=$ 671 875 accesses per second

- Network packets access memory location up to 6 times (depending on kernel)

$\rightarrow$ 111 979 packets per second

- Network packets are a least 64 B

Just comparing some performance numbers...

- $\geq$ 43 000 hammering attempts (within 64 ms) for a bit flip [GMM16]

$=$ 671 875 accesses per second

- Network packets access memory location up to 6 times (depending on kernel)

$\rightarrow$ 111 979 packets per second

- Network packets are a least 64 B

$=$ 7 166 656 B/s $=$ 7 MB/s $=$ 57 Mb/s

Just comparing some performance numbers...

- $\geq$ 43 000 hammering attempts (within 64 ms) for a bit flip [GMM16]

$=$ 671 875 accesses per second

- Network packets access memory location up to 6 times (depending on kernel)

$\rightarrow$ 111 979 packets per second

- Network packets are a least 64 B

$=$ 7 166 656 B/s $=$ 7 MB/s $=$ 57 Mb/s

$\rightarrow$ Doable on "modern" networks [Lip+17]

- $\geq$ 43 000 hammering attempts (within 64 ms) for a bit flip [GMM16]

$=$ 671 875 accesses per second

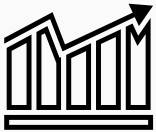
- Network packets access memory location up to 6 times (depending on kernel)

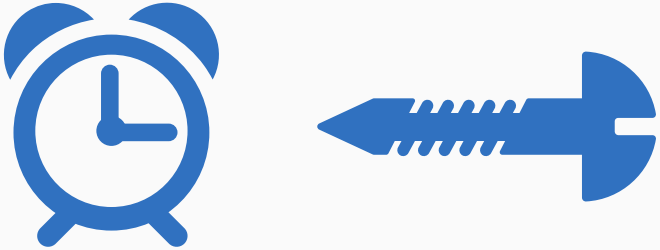
$\rightarrow$ 111 979 packets per second

- Network packets are a least 64 B

$=$ 7 166 656 B/s $=$ 7 MB/s $=$ 57 Mb/s

$\rightarrow$ Doable on "modern" networks [Lip+17]



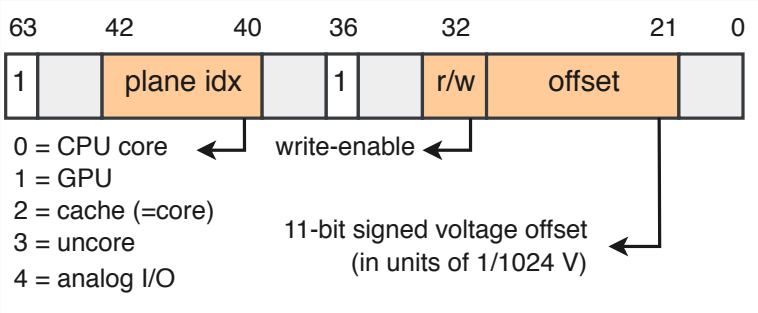


- RMClock

- RMClock
- Throttlestop

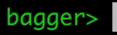
- RMClock
- Throttlestop
- linux-intel-undervolt

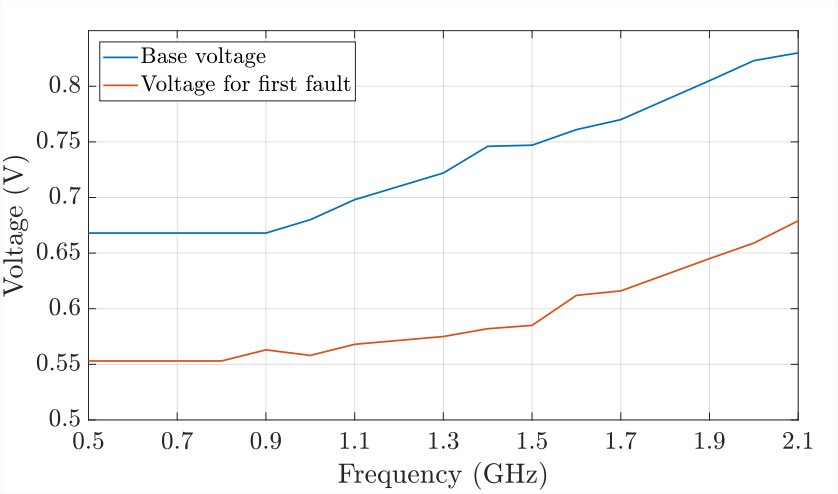
- RMClock
- Throttlestop
- linux-intel-undervolt
- Adrian Tang's PhD dissertation


63
42
40
36
32
21
0
1
plane idx
1
r/w
offset
0 = CPU core
1 = GPU
2 = cache (=core)
3 = uncore
4 = analog I/O
write-enable
11-bit signed voltage offset
(in units of 1/1024 V)

Will it fault?

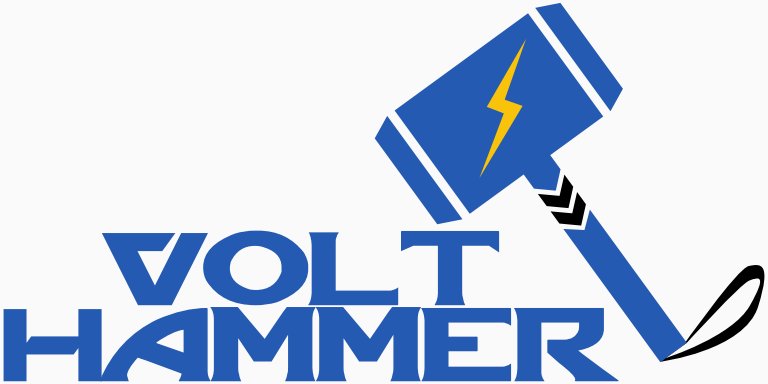
```
uint64_t multiplier = 0x1122334455667788;
uint64_t correct    = 0xdeadbeef * multiplier;
uint64_t var        = 0xdeadbeef * multiplier;

while (var == correct)
{
    var = 0xdeadbeef * multiplier;
}
            uint64_t flipped_bits = var ^ correct;
```


bagger>


Base voltage
Voltage for first fault
Voltage (V)
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.5
0.7
0.9
1.1
1.3
1.5
1.7
1.9
2.1
Frequency (GHz)

Aristotle Tzafalias
@Aristot73

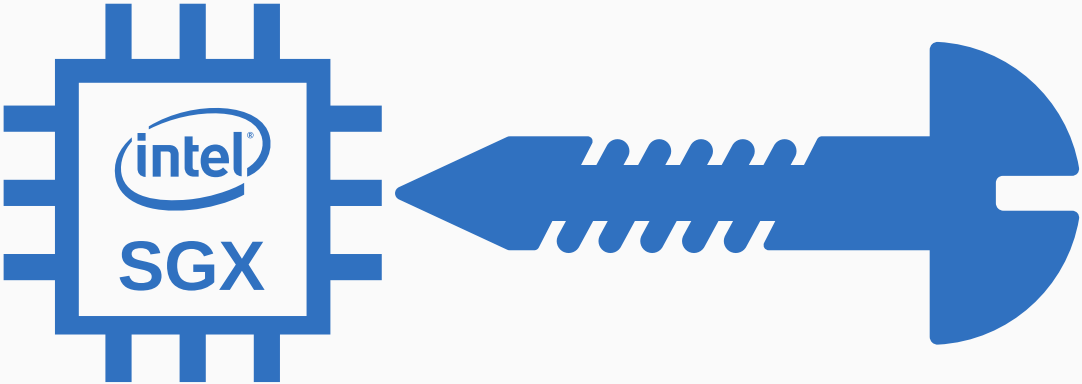
thesis: given a cool name, researchers will be able to find a new vulnerability to match it, in a finite period of time; cool name-logo combination requirements have no measurable effect on aforementioned period.


VOLT
HAMMER

What about the integrity-protected SGX EPC?

[Userspace] Voltage: -261 mV

```
[Enclave] Multiplying 0xdeadbeef 0x1122334455667788
[Enclave] Multiply_it. result: 0.
```


intel
SGX

- Public Key Crypto
- Encrypt/Sign messages
- Untrusted channel
- Encrypt/Verify messages with public key
- Decrypt/Sign messages with private key

$$n = \mathrm{p} \times q$$

$$S = H^{d} \bmod n$$

$$n = \mathrm{p} \times q$$

$$S = H^{d} \bmod n$$

$$d_p = d \bmod p - 1$$

$$d_q = d \bmod q - 1$$

$$s_1 = H^{d_p} \bmod p$$

$$s_2 = H^{d_q} \bmod q$$

$$n = \mathrm{p} \times q$$

$$S = ((q^{-1} \bmod p)(s_1 - s_2) \bmod p) \times q + s_2$$

$$d_p = d \bmod p - 1$$

$$d_q = d \bmod q - 1$$

$$s_1 = H^{d_p} \bmod p$$

$$s_2 = H^{d_q} \bmod q$$

$$n = \mathrm{p} \times q$$

$$S = ((q^{-1} \bmod p)(s_1 - s_2) \bmod p) \times q + s_2$$

$$d_p = d \bmod p - 1$$

$$d_q = d \bmod q - 1$$

$$s_1 = H^{d_p} \bmod p$$

$$s_2 = H^{d_q} \bmod q$$

$$n = \mathrm{p} \times q$$

$$S = ((q^{-1} \bmod p)(s_1 - s_2) \bmod p) \times q + s_2$$

$$S' = ((q^{-1} \bmod p)(s_1 - s_2) \bmod p) \times q + s_2$$

$$n = p \times q$$

$$S = \left(\boxed{\text{something}} \right) \times q + s_2$$

$$S' = \left(\boxed{\text{something else}} \right) \times q + s_2$$

$$n = \mathrm{p} \times q$$

$$S = (\text{something}) \times q + s_2$$

$$S' = (\text{something else}) \times q + s_2$$

$$S - S' = \text{something} \times q$$

$$n = p \times q$$

$$S = (\text{something}) \times q + s_2$$

$$S' = (\text{something else}) \times q + s_2$$

$$S - S' = \text{something} \times q$$

$$q = \gcd(S - S', n)$$

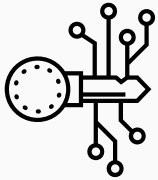
RSA decryption

```
uint8_t rsa_dec_ecall(int iterations)
{
    // Wait for first fault
    trigger_fault(iterations);

    // Actual decryption
    ippsRSA_Decrypt(ct, dec, pPrv, scratchBuffer);
}
```

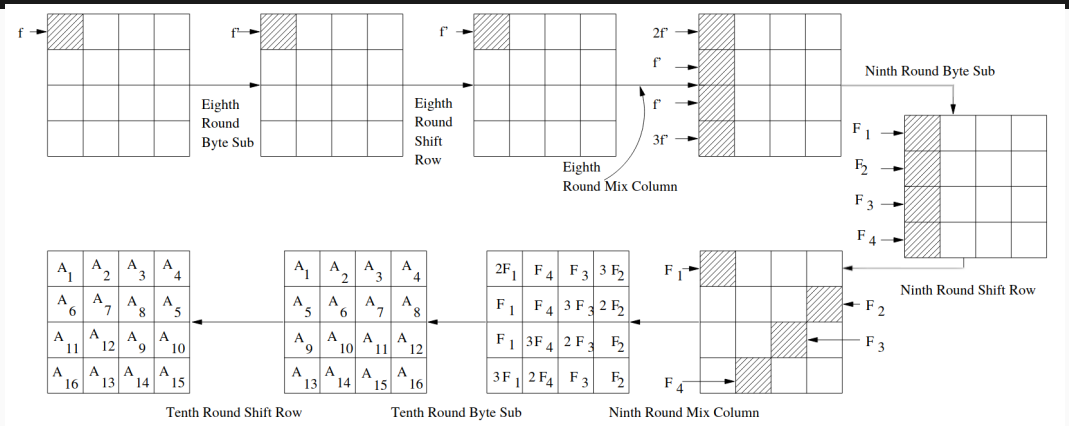
```
bagger> dog Enclave/encl
```


MUFARSA
My Undervolting
Fault Attack on RSA
imgflip.com

WHAT ELSE CAN WE BREAK?!



- Symmetric Key Crypto
- Encrypt messages for transfer over public channel
- Encrypt data for (untrusted) storage
- 4×4 byte state
- 10 rounds: S-Box, ShiftRows, MixColumns, AddRoundKey



Michael Tunstall et al. Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. In: International Workshop on Information Security Theory and Practices. 2011 [TMA11]

AES-NI (New Instructions)

Instruction	Description
`AESENC`	Perform one round of an AES encryption flow
`AESENCLAST`	Perform the last round of an AES encryption flow
`AESDEC`	Perform one round of an AES decryption flow
`AESDECLAST`	Perform the last round of an AES decryption flow
`AESKEYGENASSIST`	Assist in AES round key generation
`AESIMC`	Assist in AES Inverse Mix Columns
`PCLMULQDQ`	Carryless multiply (CLMUL)

```
do
{
        i++;
        plaintext = <randomly generated>

        result1 = aes128_enc(plaintext);
        result2 = aes128_enc(plaintext);
} while (vec_equal_128(result1,result2) && i<iterations);
```

```
bagger> sudo ./aes-encrypt 100000 -262
```


CRYPTO
VAULT
SCREW
HAMMER

But what about memory corruption?

```
struct_foo_t *foo = &arr[offset];
foo->foo = enclave_secret;
```

But what about memory corruption?

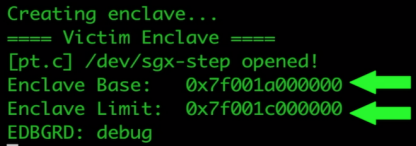
```
struct_foo_t *foo = &arr[offset];
foo->foo = encrypted_secret;
```

But what about memory corruption?


```
foo = arr + offset * 0x24
```

But what about memory corruption?

```
foo = arr + offset * 0x24
```



Voltage
0.584V

Undervolting
-235mV

- Should be related to undervolting

- Should be related to undervolting
- From protected TEE vaults

- Should be related to undervolting
- From protected TEE vaults
- Steal

- Should be related to undervolting
- From protected TEE vaults
- Steal, corrupt

- Should be related to undervolting
- From protected TEE vaults
- Steal, corrupt, plunder, ...


PLUNDER
VOLT

Faulting Hardware from Software

Daniel Gruss

2020-09-13

Graz University of Technology