Systems Programming

Security Attacks

Byoungyoung Lee

Seoul National University

byoungyoung@snu.ac.kr

https://lifeasageek.github.io

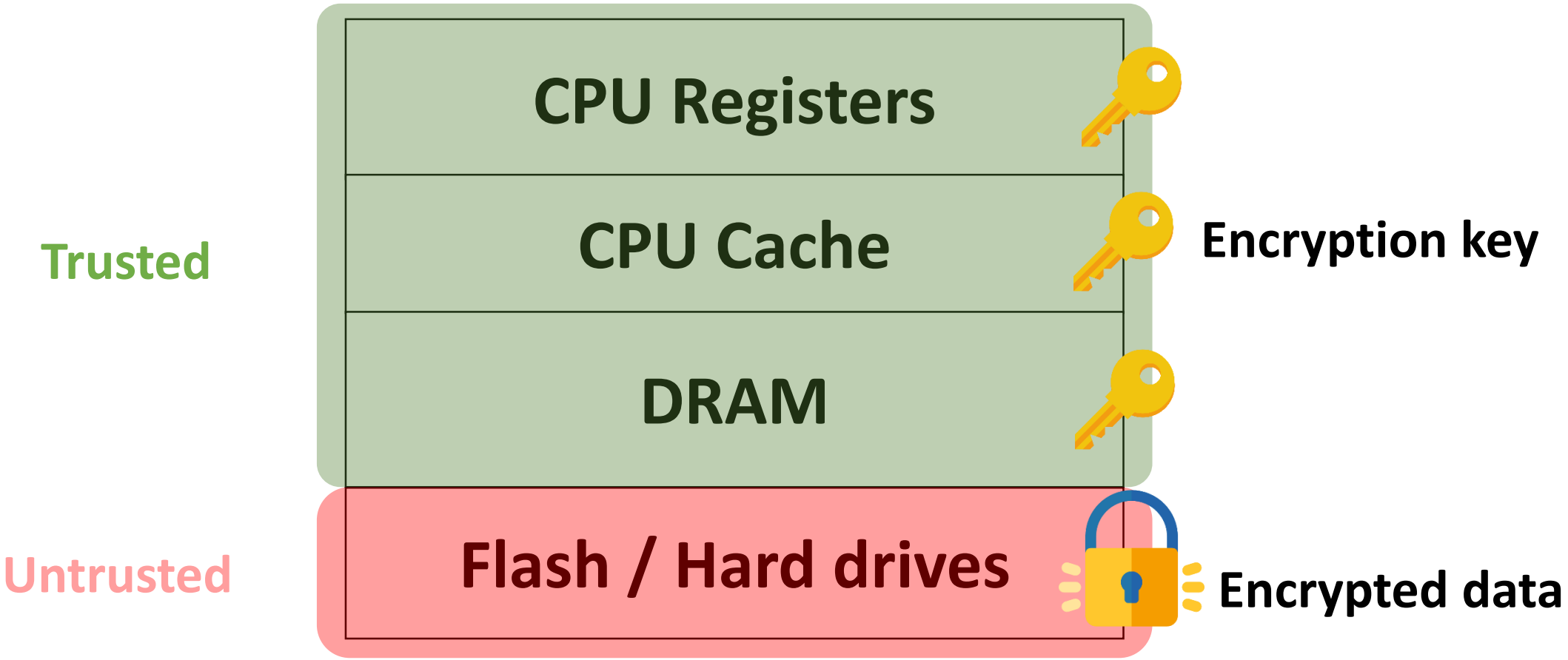
Outline

- Cold boot attacks
- Side-channels
- Cache Side-channels and Meltdown
- Row hammer

Disk Encryption (Encrypted File Systems)

Trust Models in Encrypted FS

- Memory hierarchy with trust models (when OS is trusted)

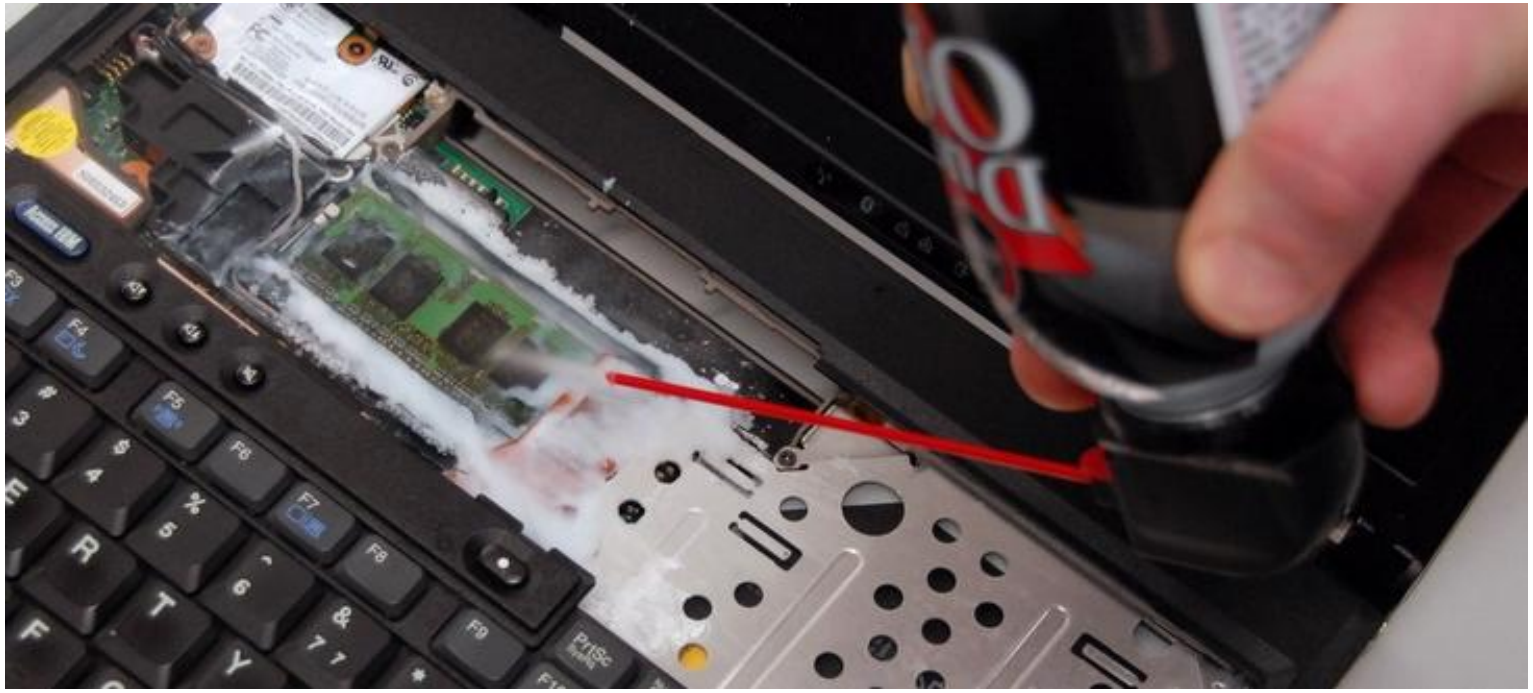


Attacking DRAM

- Physical attacks against DRAM
  - What would happen if DRAMs are detached from DIMM slots?
  - Should data be retained? Probably not.
  - DRAM cells have to be refreshed
    - If detached, a data value in a capacitor decays over time

- Can we slowdown decay?

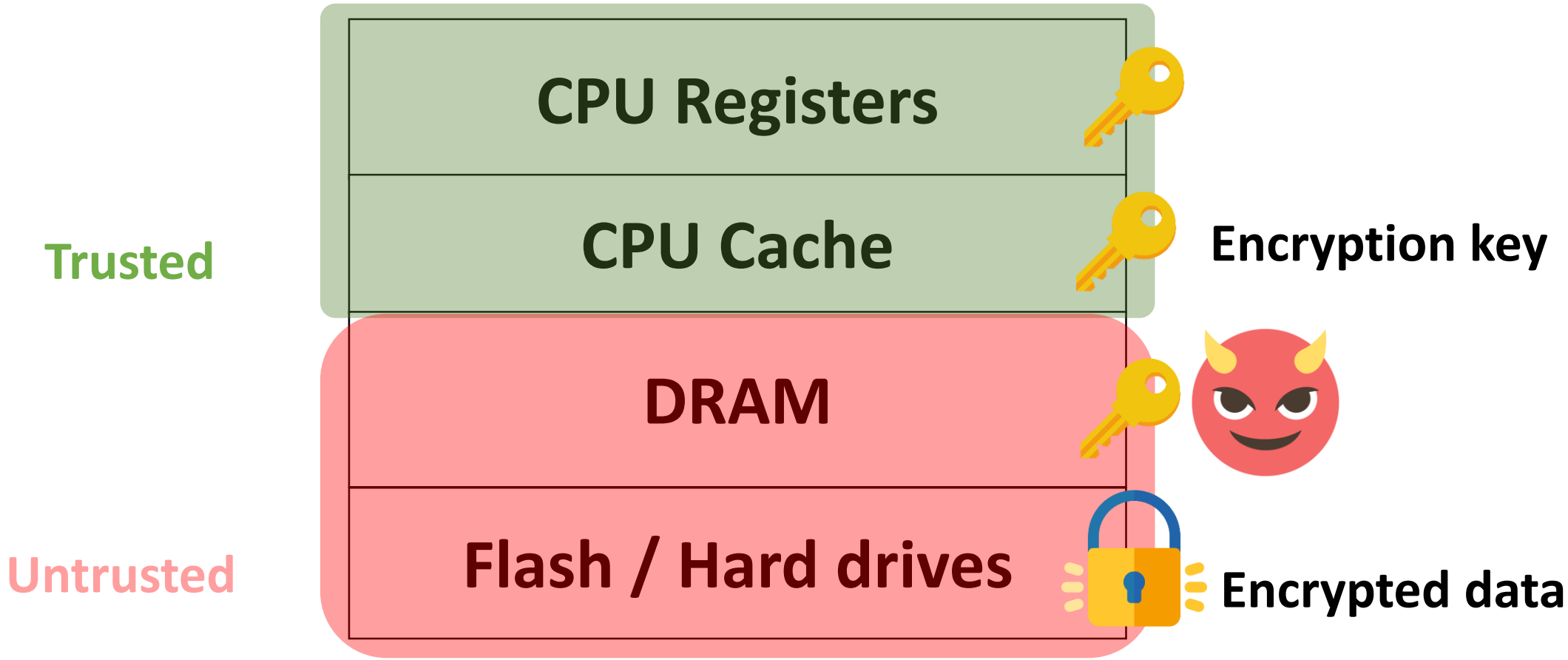
Cold Boot Attack: Slowing Decay by Cooling



-50˚C: less than 0.2% decay after 1 minute

“Lest We Remember: Cold Boot Attacks on Encryption Keys [USENIX Security 08]”

Security Implications of Cold Boot Attack

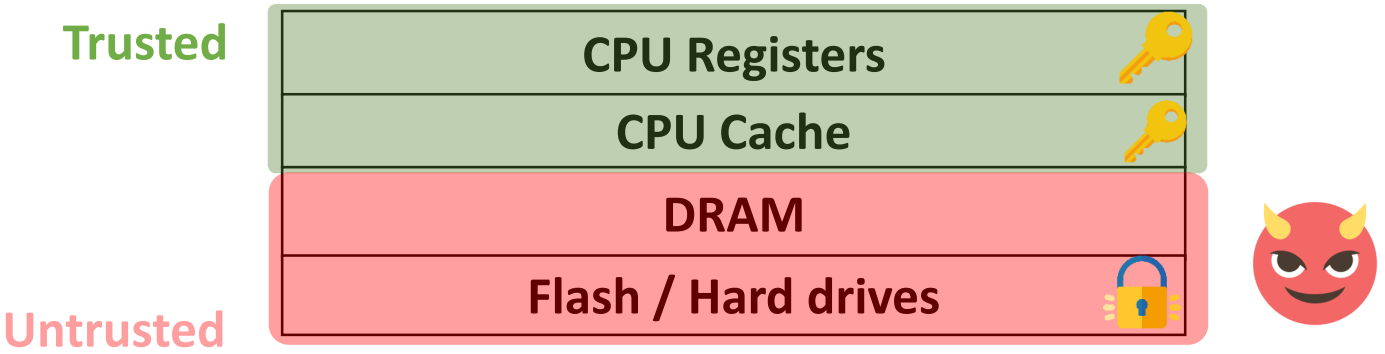


Security Implications of Cold Boot Attack

- Encryption keys stored in DRAM can be leaked
- Demonstrated attacks in Windows BitLocker
  - MacOS FileVault
  - Linux dm-crypt
  - Linux LoopAES
  - TrueCrypt

Countermeasures against Cold Boot Attack

- Encryption key and states only present in registers/cache
  - TRESOR [USENIX Security 11]
    - Linux kernel patch
    - The AES encryption algorithm and its key management solely on CPU



Outline

- Cold boot attacks
- Side-channels
- Cache Side-channels and Meltdown
- Row hammer

Side-Channels

- Definition from Wikipedia

"Any attack based on information **gained from the implementation** of a computer system, rather than weaknesses in the implemented algorithm itself
(e.g. cryptanalysis and software bugs)"

"Timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information, which can be exploited."

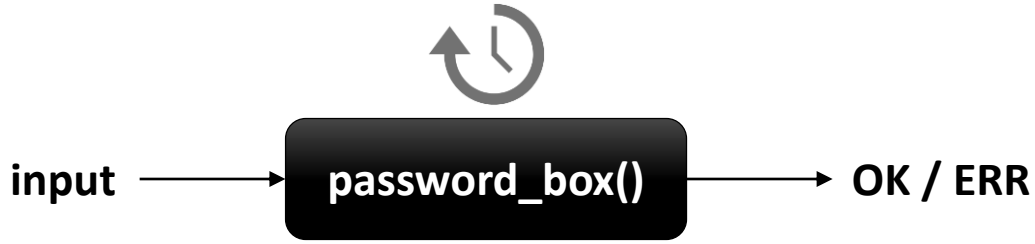
Timing Side-Channels: Example

```
void password_box(char* input) {
  const char *password = “SnuEngEce”;

  if (strcmp(password, input) {
     // password is incorrect
     return ERR;
  }
  // password is correct
  // do something security sensitive here
 return OK;
}
```

- **Attacker’s goal**
  - **Learn the password**
- **Random guessing takes about 52^9**

Timing Side-Channels: Idea



You cannot see the inside of password_box().
But, you can **measure the time it takes to return.**

Timing Side-Channels: Attack

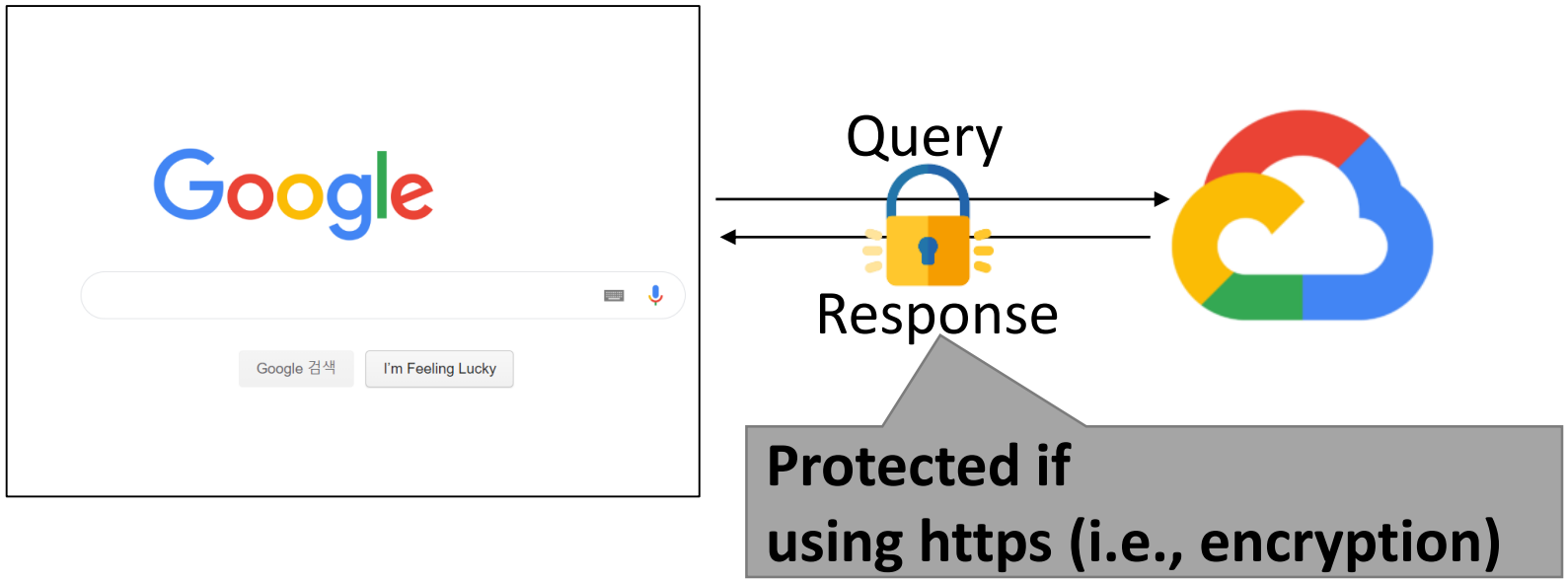
Let time(input) be the time password_box() takes to check input.

- time(“**aa**”) == time(“**ba**”) == … == time(“**Ra**”) < **time(“Sa”)**

If any character matches ➔ takes a more time to match

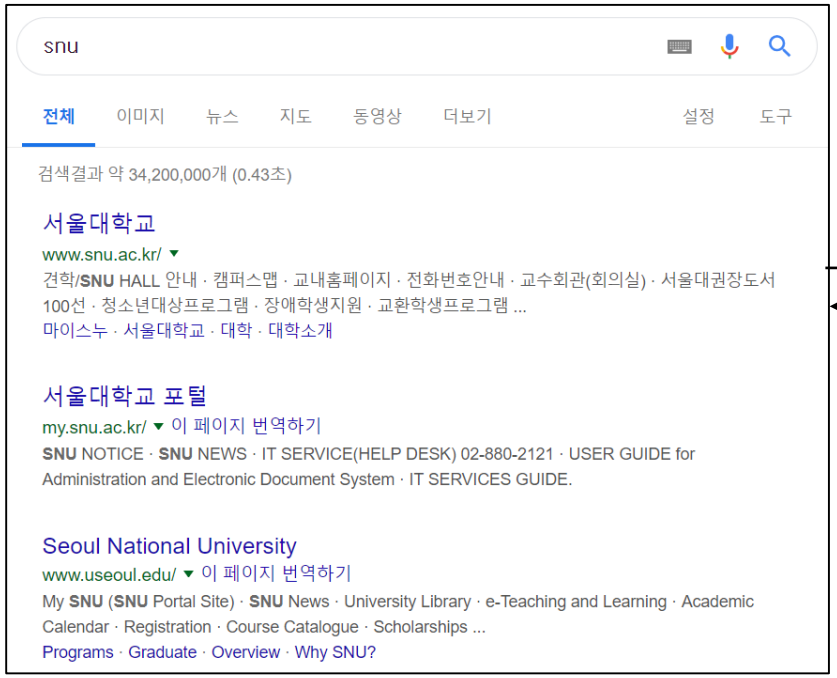
Complexity to break the password through timing attacks
➔ **52 * 9**

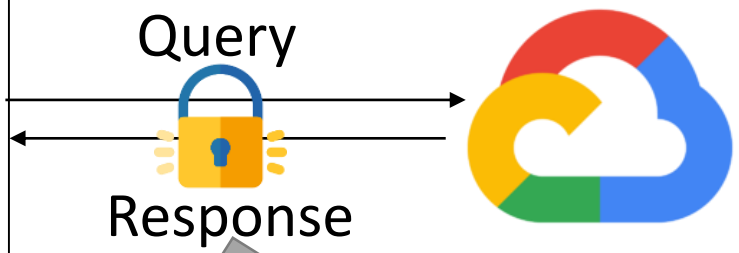
Side-Channels in Google's Instance Search



Attacker's goal: Learn the user's query

Side-Channels in Google's Instance Search

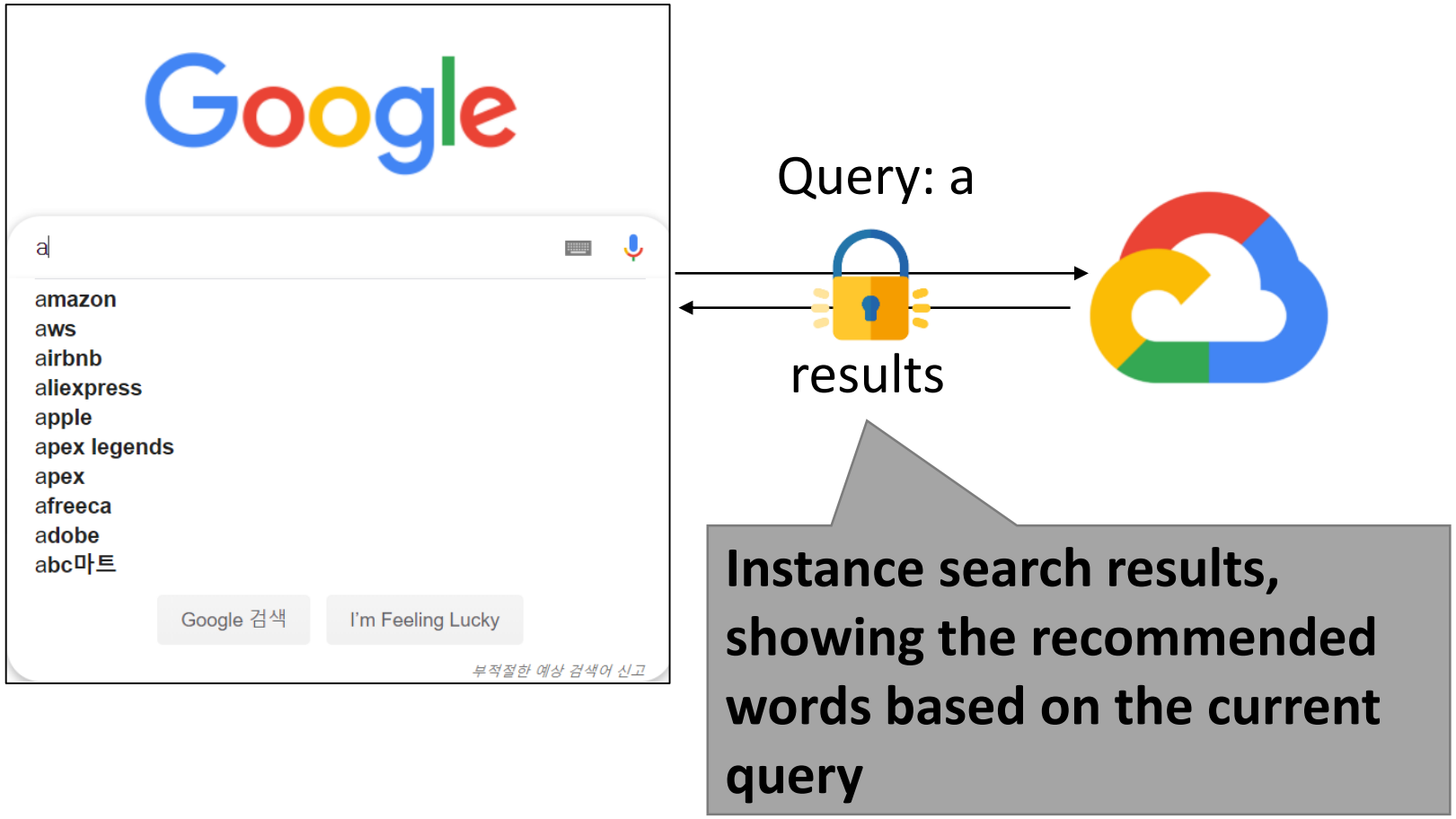




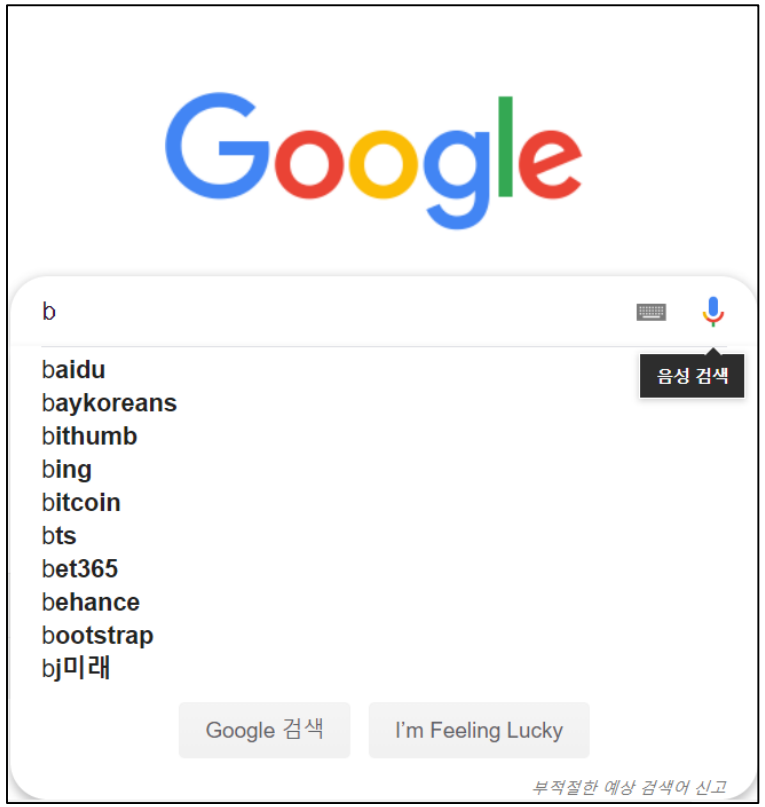
Protected if using https (i.e., encryption)

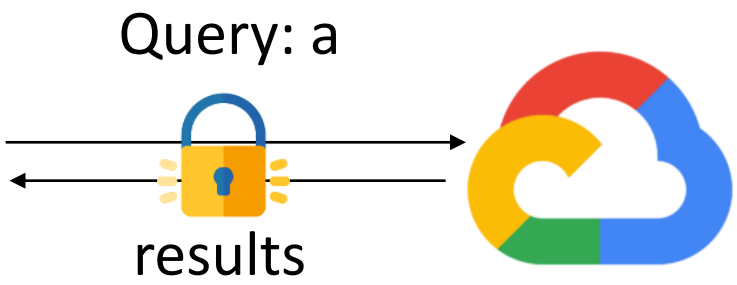
Attacker's goal: Learn the user's query

Side-Channels in Google's Instance Search



Side-Channels in Google's Instance Search





Instance search results, showing the recommended words based on the current query

Side-Channel Attacks: VOIP

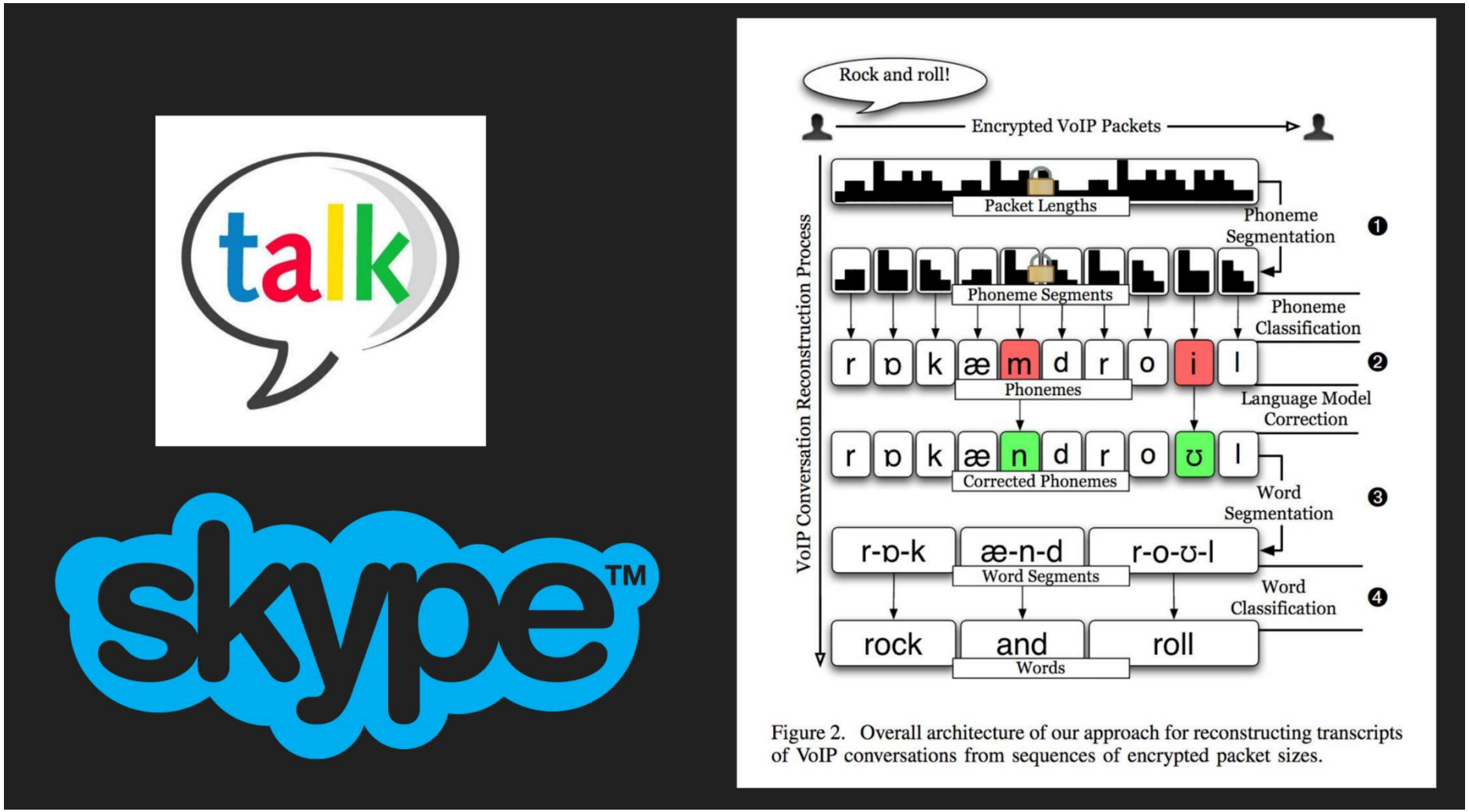
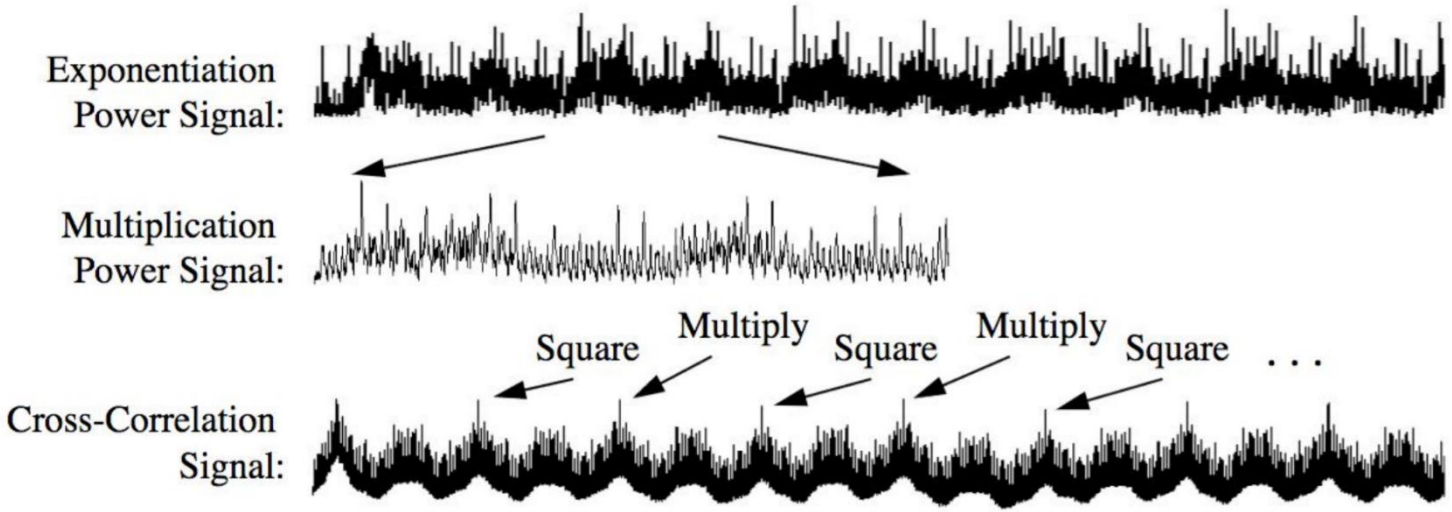


Figure 2. Overall architecture of our approach for reconstructing transcripts of VoIP conversations from sequences of encrypted packet sizes.

Sound Side-channel: Acoustic Sound from CPU

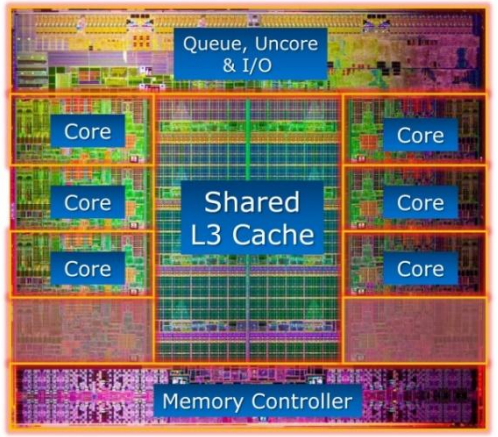


Outline

- Cold boot attacks
- Side-channels
- Cache Side-channels and Meltdown
- Row hammer

Cache Side-channel

- CPU Cache
  - Reduce the latency to fetch the data in DRAM
  - Cache hit
    - The accessing data presents in DRAM
    - Fast access
  - Cache miss
    - the accessing data does not present in DRAM
    - Slow access

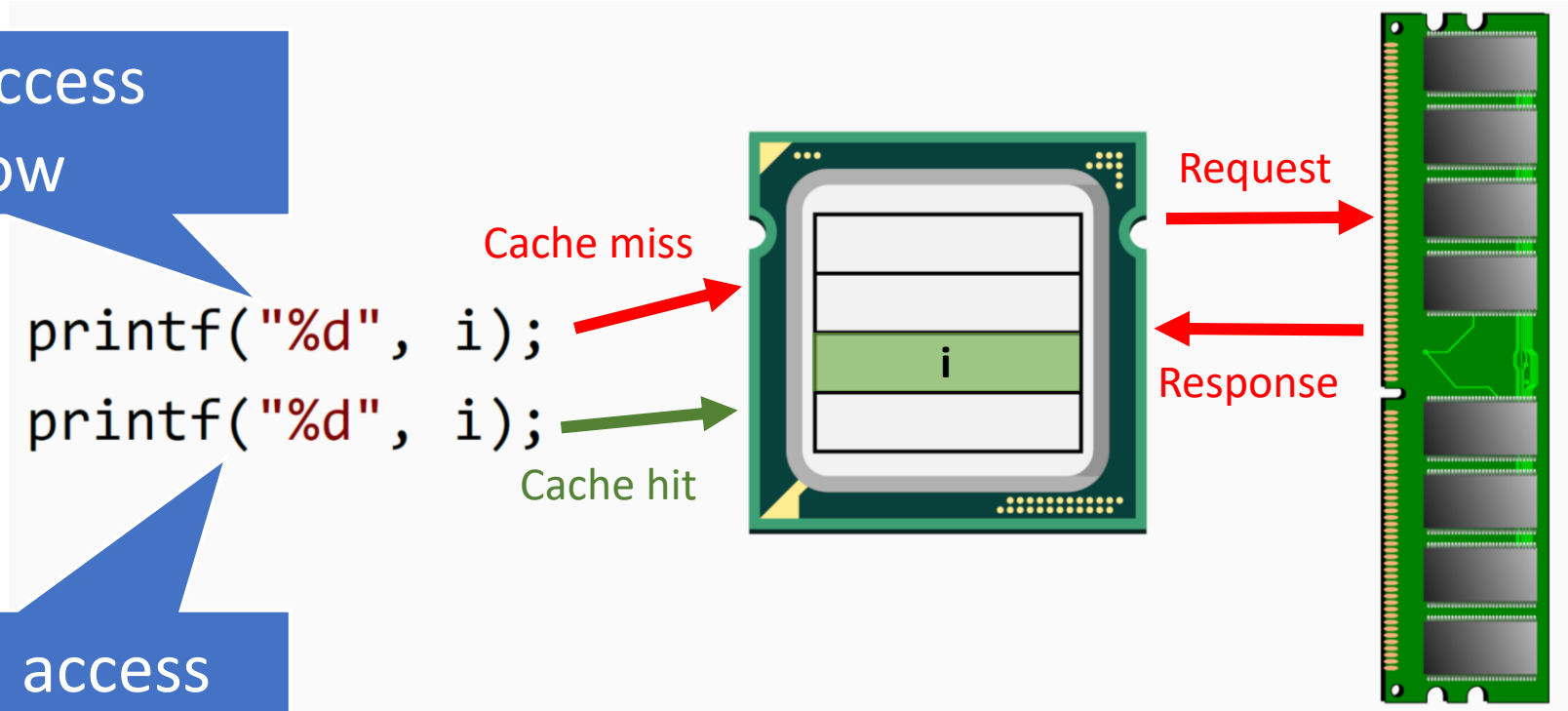


https://www.pcper.com/files/imagecache/article_max_width/review/2011-12-14/die.jpg

Cache Side-Channel

- Timing channels in CPU Cache

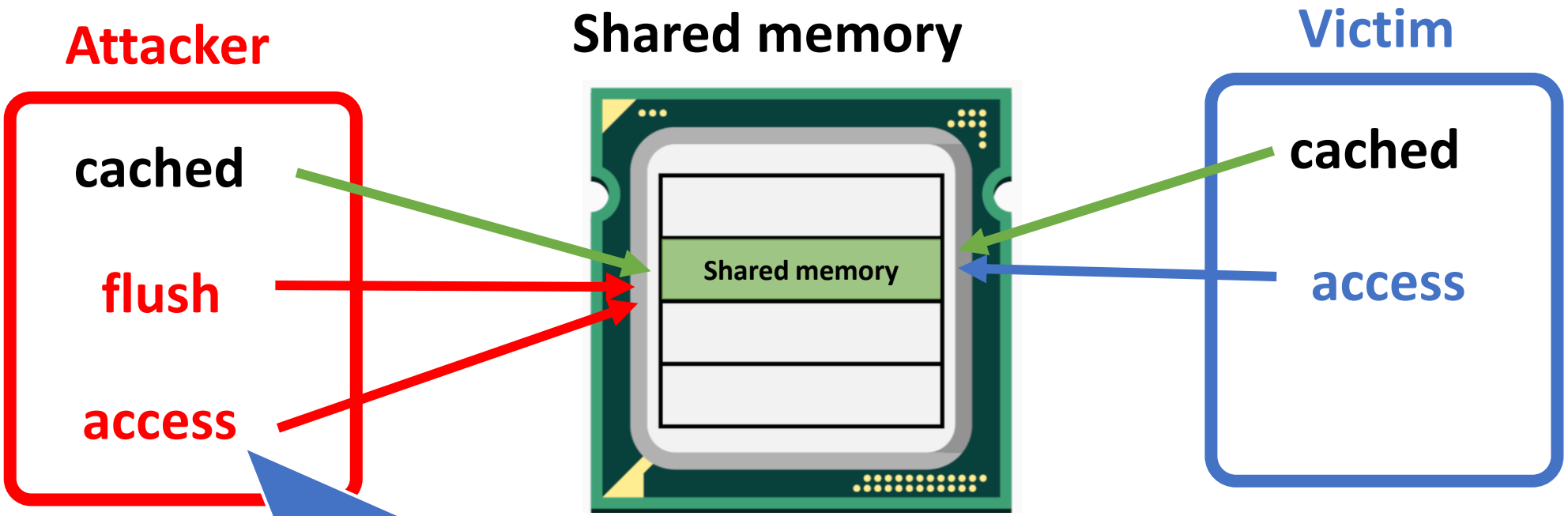
DRAM access
➔ Slow



No DRAM access
➔ Much faster

Cache Side-Channel

- Flush+Reload attack



Fast if victim accessed data, slow otherwise

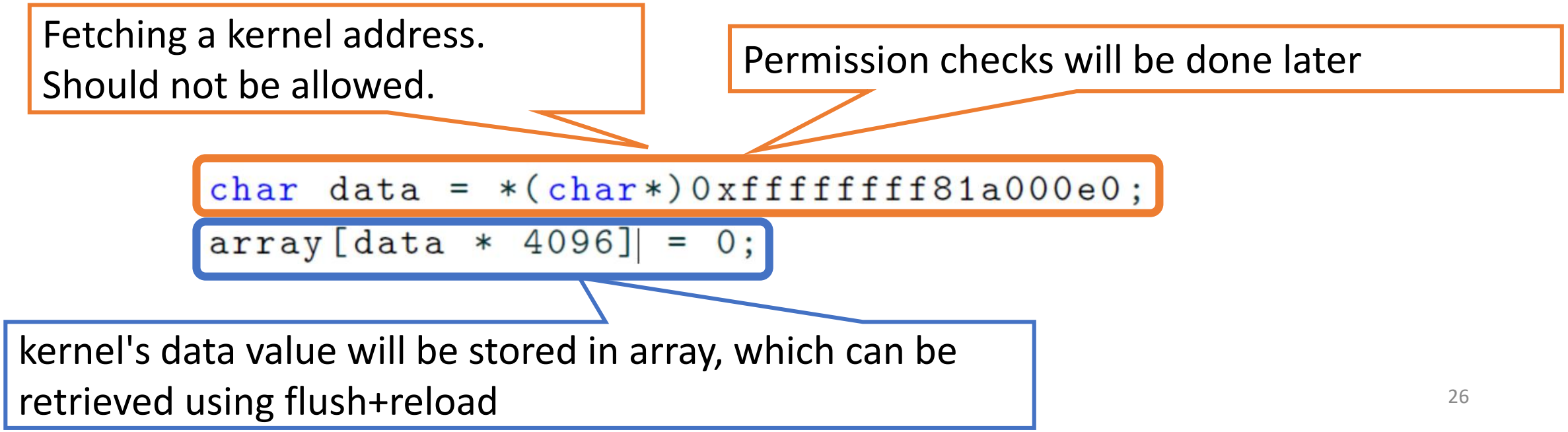
Speculative Execution and Transient Instructions

- CPU speculatively executes code, which won't be committed in the end
- If CPU executes such code, it should be reverted back
  - Some architectural states may not be reverted by the design of the architecture
  - **Transient instructions**: An instruction which raises such a behavior

Meltdown

- Permission check for transient instructions is only done
  - when committing them

- Suppose we are running a user-level program below



Meltdown: Transient execution

Step#1. Transient execution

```
// user's program. Simplified Meltdown PoC.
int main(void) {
    char user_array[255 * 4096];
     // Flushing all cache lines of user_array

    char kernel_data = *(char *)0xffffffff81a000e0;
    maccess(user_array[kernel_data * 4096]);

    for (uint index=0; index<255; i++) {
        addr = &user_array[index*4096];
        time = get_access_time(addr);
    }
}
```

Step#1A. Accessing secrets

Step#1B. Send secrets

Step#2. Receive Secrets

Check the working PoC from https://github.com/paboldin/meltdown-exploit

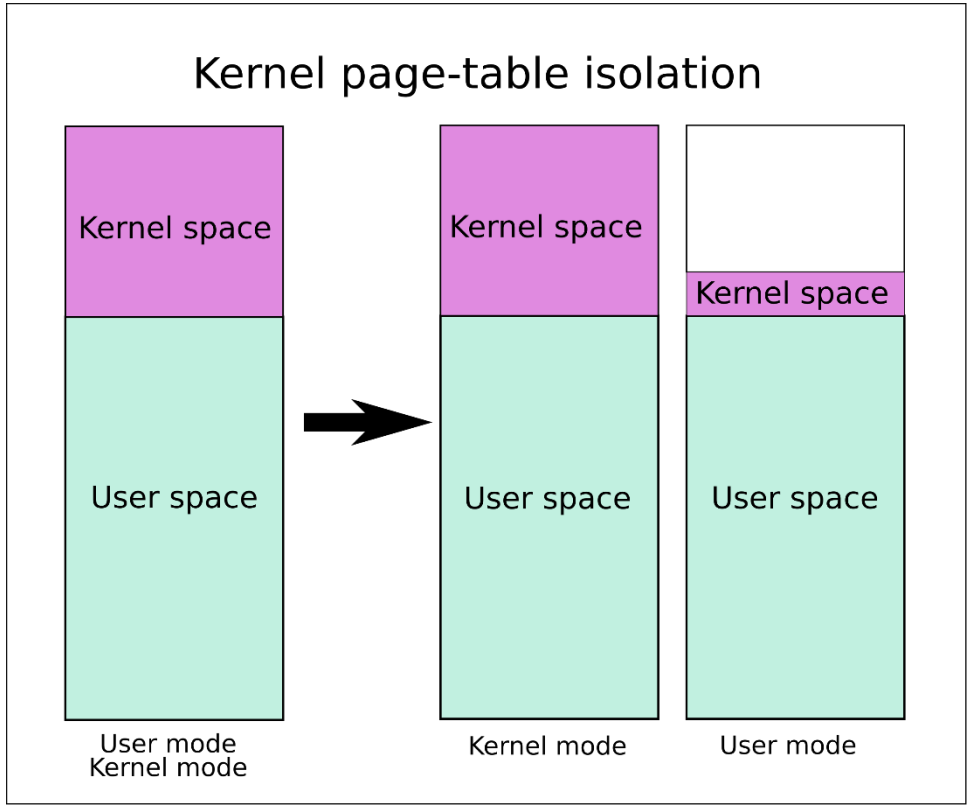
Meltdown: Receiving secrets

- Receiving through cache covert-channel
  - Flush+Reload over all pages of the user_array
  - i.e., user_array[**index***4096], where 0 <= **index** < 255;

  - Index of cache hit reveals the value of data
    - kernel_data == **index**, for which index shows the minimum access time for all index

Meltdown: Mitigation

- Kernel Page Table Isolation (KPTI)
  - Separate PTs for kernel and user modes
  - Performance limitation
    - Raise Context-switching overheads
      - TLB flush, page-table swapping, etc.
    - 5%-20% overheads
  - Merged to the mainline kernel in 2017
    - Note: KASLR was adopted to Linux in 2014



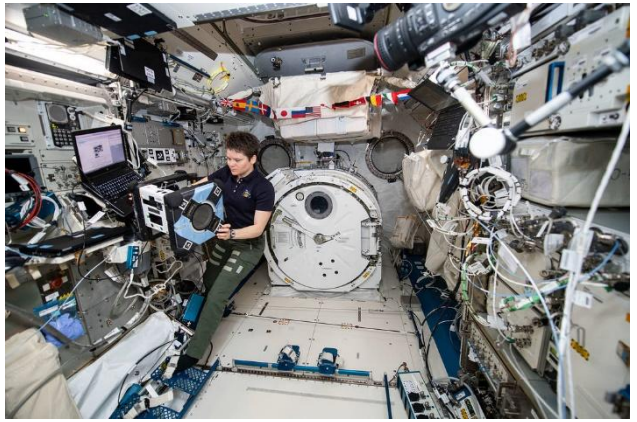
https://en.wikipedia.org/wiki/Kernel_page-table_isolation

Outline

- Cold boot attacks
- Side-channels
- Cache Side-channels and Meltdown
- Row hammer

Random Bit Flips in DRAM

- Random bit flips in DRAM can be observed
  - DRAM in a space station
  - Laser induced heating



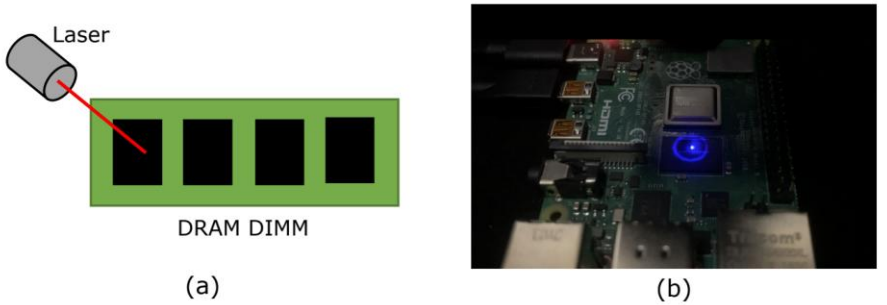


Fig. 1. (a) DRAM Testing setup schematic (b) Setup used to induce the faults experimentally on Raspberry-Pi4

Exploiting Random Bit Flips

- Using memory Errors to Attack a Virtual Machine [SP 03]
  - Identify data structure, which offers escalated privilege if randomly bit-flipped
  - Fill memory as much as possible with this data structure (similar to heap spray)
  - Launch bit-flip attacks and wait until the bit-flip occurs

  - https://www.cs.princeton.edu/~appel/papers/memerr.pdf

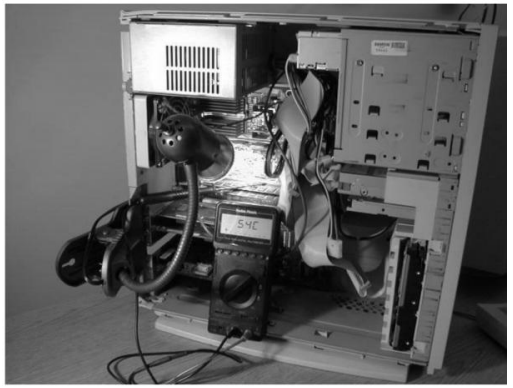


Figure 3. Experimental setup to induce memory errors, showing a PC built from surplus components, clip-on gooseneck lamp, 50-watt spotlight bulb, and digital thermometer. Not shown is the variable AC power supply for the lamp.

DRAM Essentials

- Cells are capacitors, so it should refresh contents (e.g., every 64ms)
- Accessed by row
  - Each rank has multiple banks
  - Each bank has multiple rows

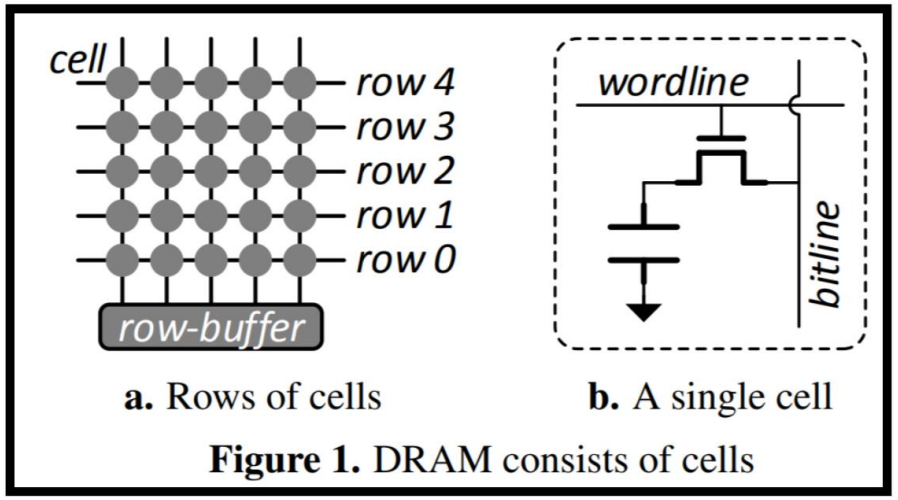


Figure 1. DRAM consists of cells

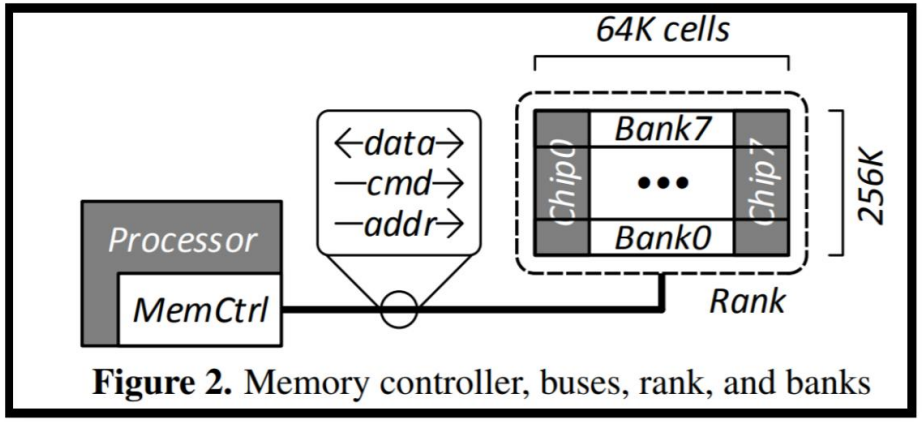
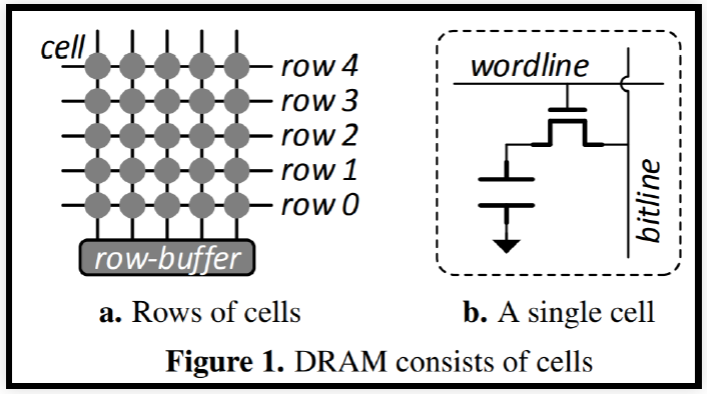


Figure 2. Memory controller, buses, rank, and banks

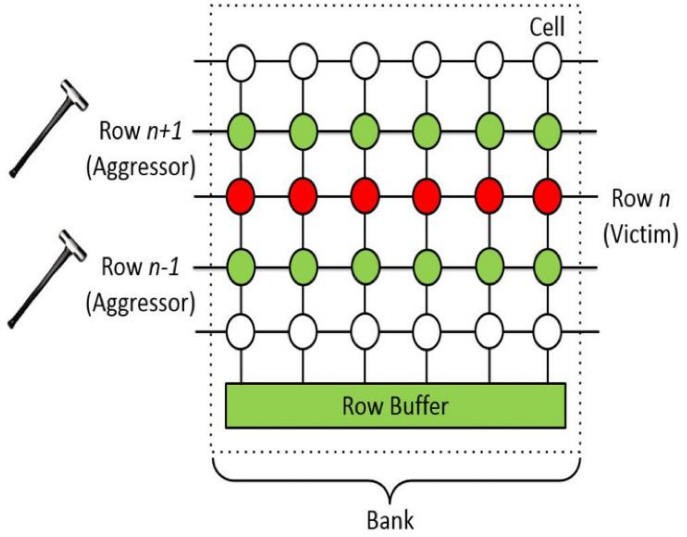
Random Bit-flips in DRAM

- **Repeat row activation** may cause bit-flips in adjacent rows
  - The issue was somehow known, and many DRAMs had the same issue
    - Three major DRAM manufacturers had this random bit-flip problems
  - Before no one knew how to exploit, it was a reliability problem
    - Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors [ISCA 14]

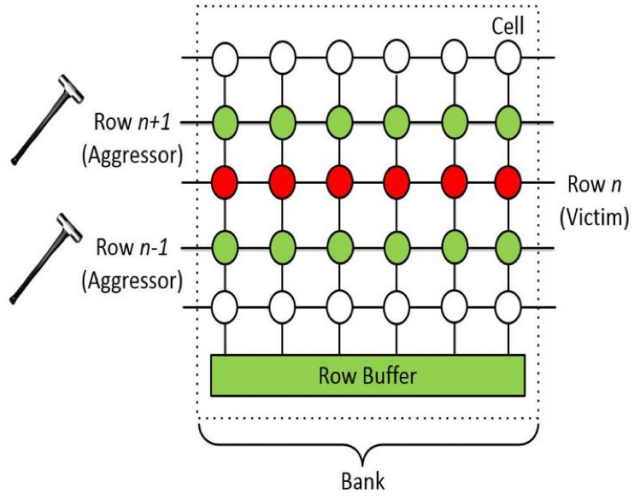


Attack Model of Rowhammer:

- Rowhammer
  - Attacker: un-privileged user
  - Attacker's goal
    - Privilege escalation
    - Flip the bit in the privileged memory
  - Attacker's strategy
    - Keep accessing two memory addresses (aggressor row)
    - In hopes some bit flips in a controlled way



Rowhammer Challenges



- Rowhammer Challenges
  - #1. Cache bypass
    - The attacker's memory access should reach DRAM
    - Can you keep accessing the DRAM, which bypasses L1/L2/L3 caches?
  - #2. Address information
    - The attacker should know two addresses, mapped to neighboring rows in the same bank
    - How do you know "virtual addresses" of "Row n+1" and "Row n-1"?

Rowhammer Ideas

- #1. Cache bypass
  - Keep flushing the cache using the cache-line flush instruction (i.e., clflush)
- #2. Address information
  - The last 12 bits of virtual addresses are the same that of physical addresses
  - Reverse-engineer to infer how the physical address is mapped into DRAM's row

```
code1a:
   mov (X), %eax // Read from address X
   mov (Y), %ebx // Read from address Y
   clflush (X) // Flush cache for address X
   clflush (Y) // Flush cache for address Y
   jmp code1a
```

Rowhammer Exploitation

- Two original rowhammer exploitation
  - 1) NaCl Sandbox escape
    - Bit-flip the validated code of NaCl
      - The attacker should not be able to modify validated code
    - Easier: the attacker can read code to see if bit-flip occurs
  - 2) Linux kernel privilege escalation
    - Bit-flip the page table entries
      - The attacker should not be able to modify the page table (which is maintained by the kernel)
    - Gain RW access to a page table

NaCl Sandbox Escape

- NaCl runs a safe subset of x86: Software Fault Isolation
  - Executable (NEXE) is checked by x86 validator
  - NEXE cannot execute privileged x86 instructions
    - But it allows CLFLUSH
- Validated instruction sequence

```
andl $~31, %eax // Truncate address to 32 bits and mask to be 32-byte-a
addq %r15, %rax // Add %r15, the sandbox base address.
jmp *%rax       // Indirect jump.
```

NaCl Sandbox Escape: Idea

- NaCl sandbox model
  - Prevent jumping into the middle of an x86 instruction
  - Indirect jumps can only target 32-byte-aligned addresses
- Idea: Bit-flips make instruction sequence unsafe
  - e.g., %eax ➔ %ecx
  - Allow jumping into a non-32-byte-aligned address
- Exploitation steps
  - (1) Spray many copies of such validate sequences in NEXE
  - (2) Rowhammer: trigger the bit-flip
  - (3) Check if bit-flip occured (code is completely readable)
  - (4) If not flipped as expected, go back to (2)

Linux Kernel Privilege Escalation

- x86 page tables entries (PTEs) are trusted
  - They control access to physical memory
  - A bit-flip in a PTE's physical page number can give an unprivileged process access to a different (privileged) physical page
- Exploitation goal: Get a write access to a page table
  - Gives access to all of physical memory
- Maximize chances that a bit-flip is useful
  - Spray physical memory with page tables (note, this is your own memory space)
  - Check for useful, repeatable bit-flip first

x86-64 Page Table Entries (PTEs)

- Page table is a 4k page containing array of 512 PTEs
- Each PTE is 64 bits:

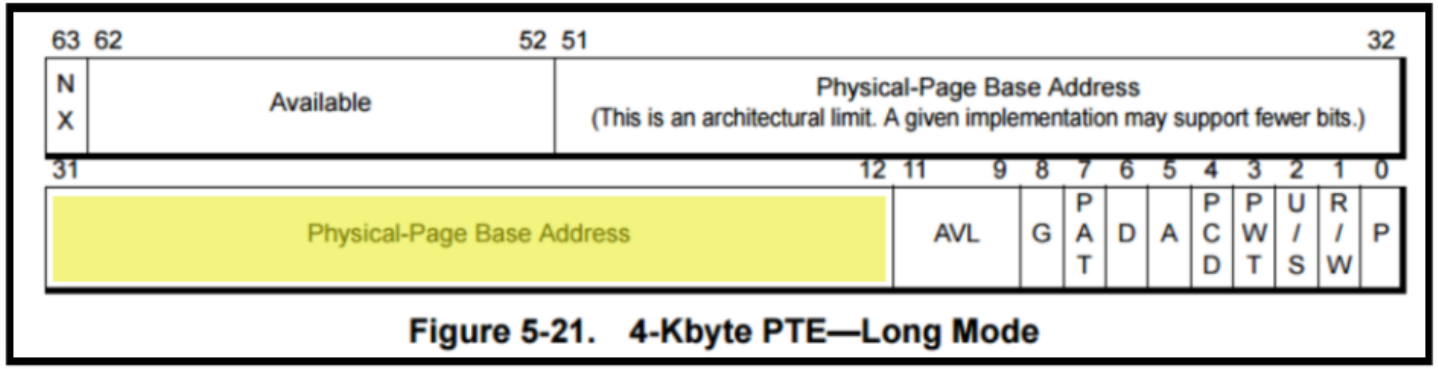


Figure 5-21. 4-Kbyte PTE—Long Mode

- Useful bit-flips in PTE
  - Writable permission bit (RW): 1 bit ➔ 2% chance (1/64)
  - Physical page number: 20 bits on 4GB system ➔31% chance (20/64)

Exploiting Linux Kernel Privilege Escalation

- Exploitation steps
  - #1. Allocate a large chunk of memory
  - #2. Search for locations prone to flipping (i.e., keep rowhammering)
  - #3. Return the memory to the kernel
  - #4. Force the kernel to use the returned memory for page table entries
    - By mapping massive quantities of address space
  - #5. Raise the bitflip, which modifies the page table entry
  - #6. Gain the access to the kernel memory