Security Engineering

Hardware Security 2: More on side channels and enclaves. Spectre, Rowhammer, Plundervolt. Codesign for security e.g. CHERI, MTE

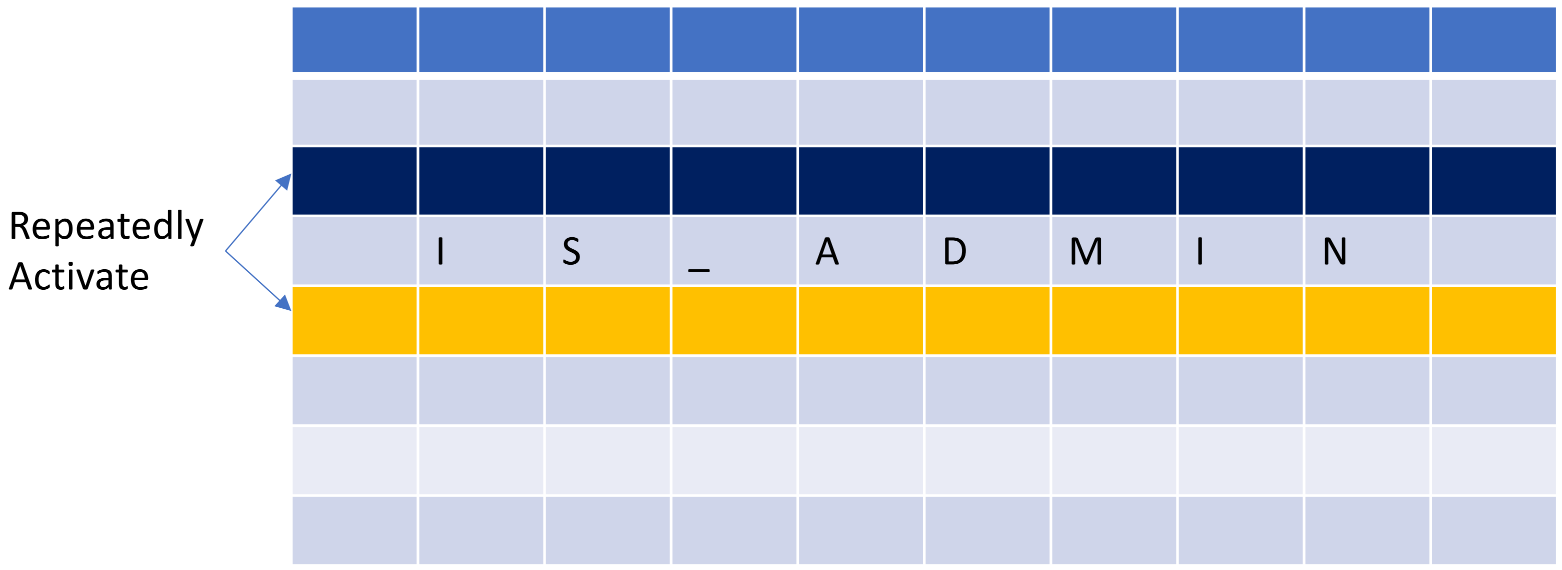
Hardware Security 2

- Today we’ll focus more on more complex systems, e.g. your phone or laptop, or a data centre
- Many of these can be triggered without physical access!
- Attacks we’ll look at: Rowhammer, Cache side channels incl. Spectre/Meltdown, Plundervolt
- We’ll also look (briefly) at hardware defence mechanisms: TPMs, Enclaves, Physically Unclonable Function.
- We’ll also look at techniques to make correct software easier to write/debug: CHERI and MTE

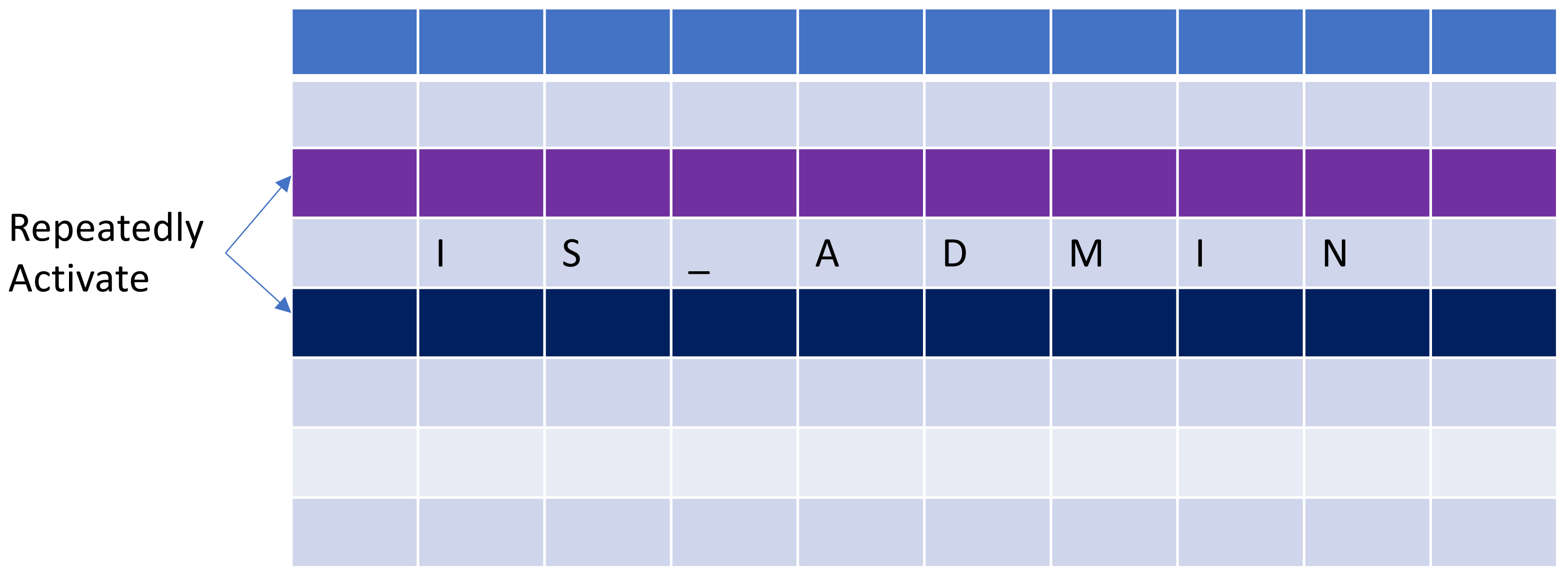
Rowhammer

	I	S	_	A	D	M	I	N	

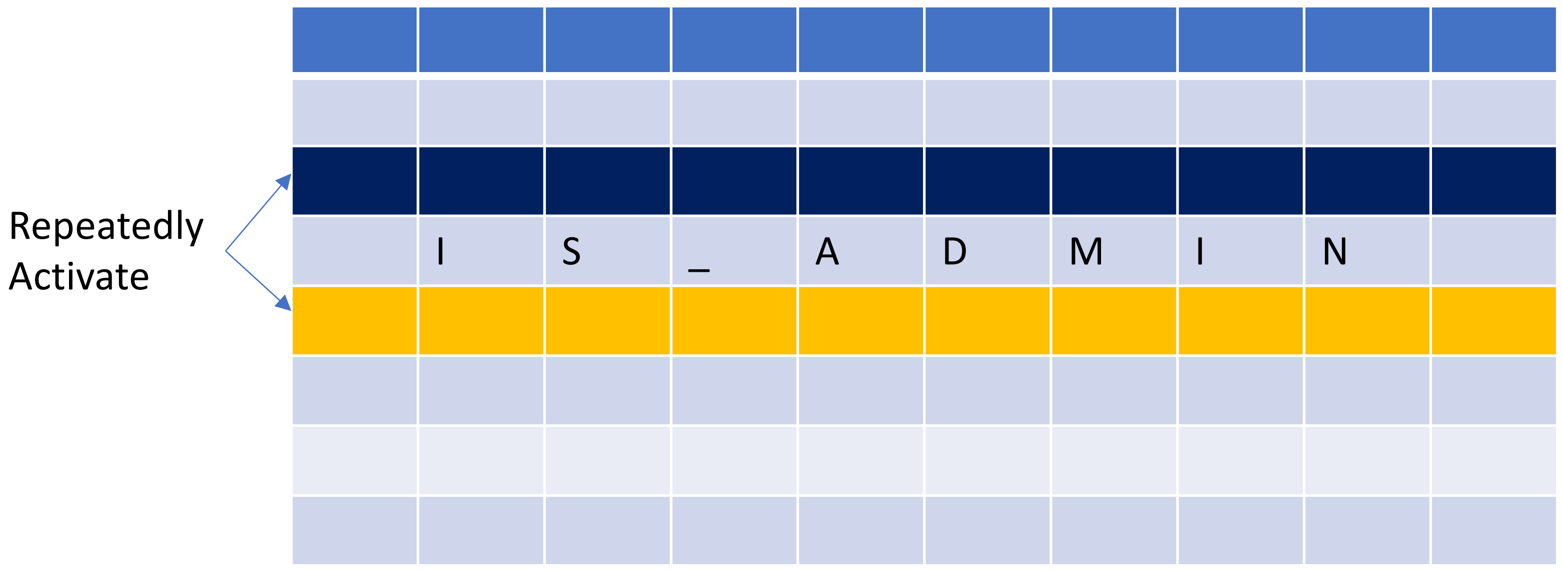
Rowhammer



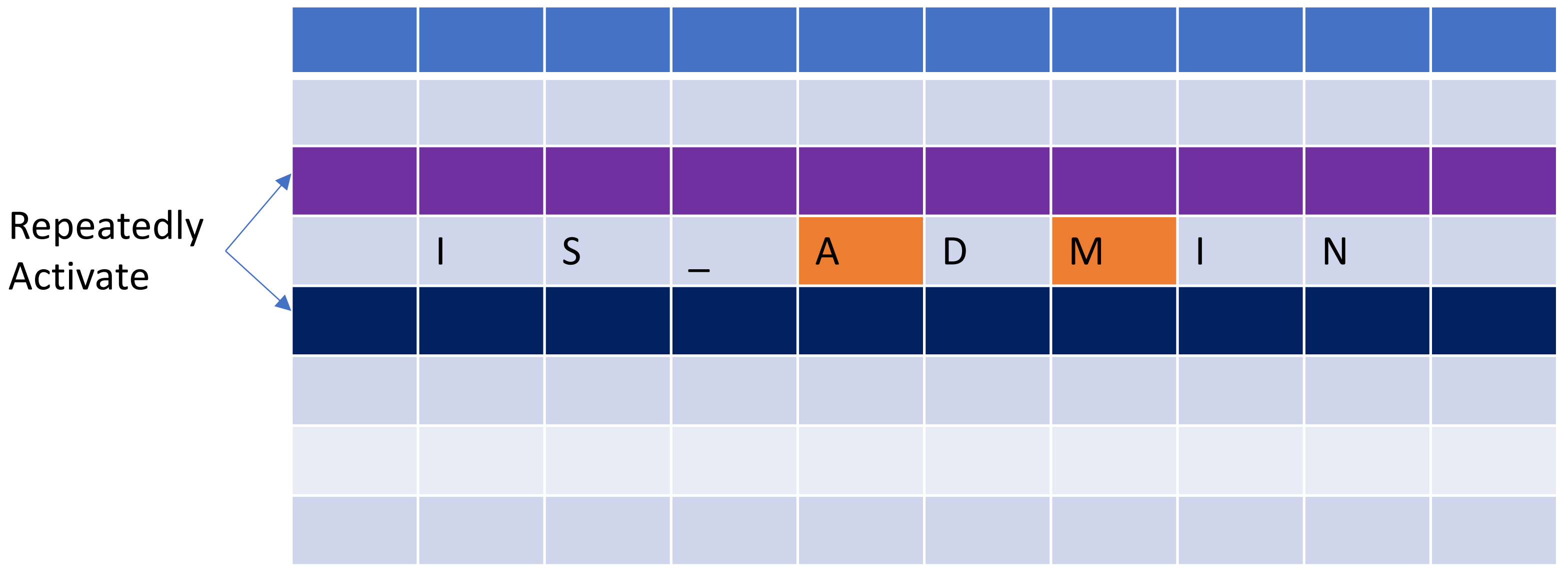
Rowhammer



Rowhammer



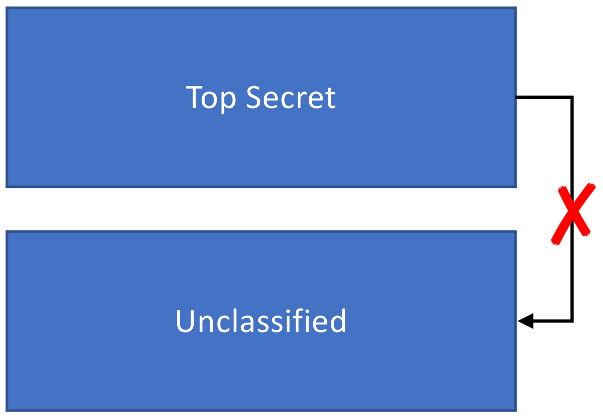
Rowhammer



Side Channels, Spectre and Meltdown

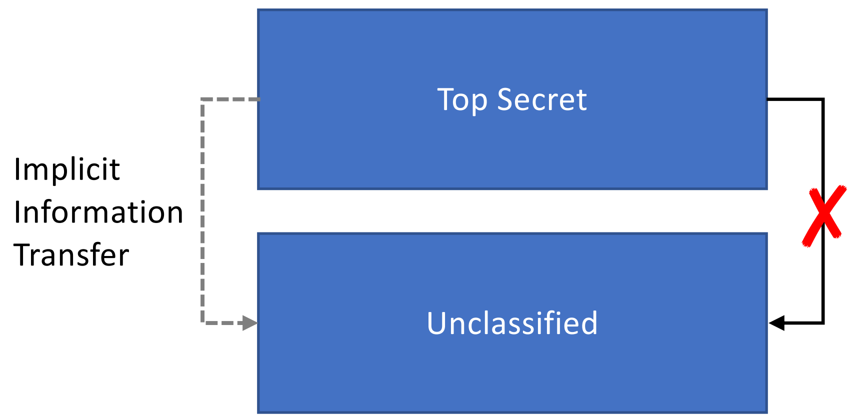
Covert Channels

- E.g. Bell LaPadula

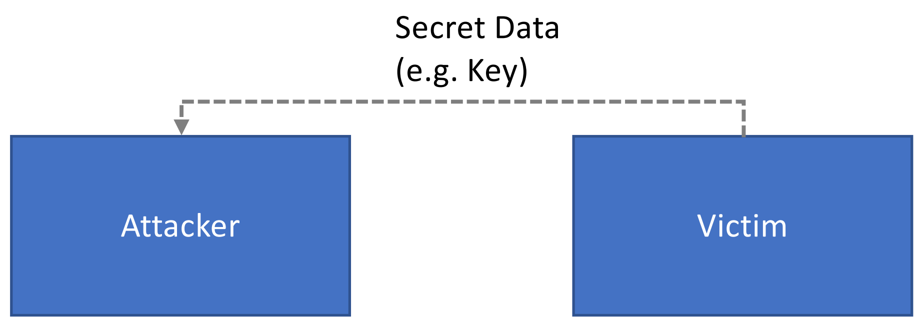


Covert Channels

- E.g. Bell LaPadula



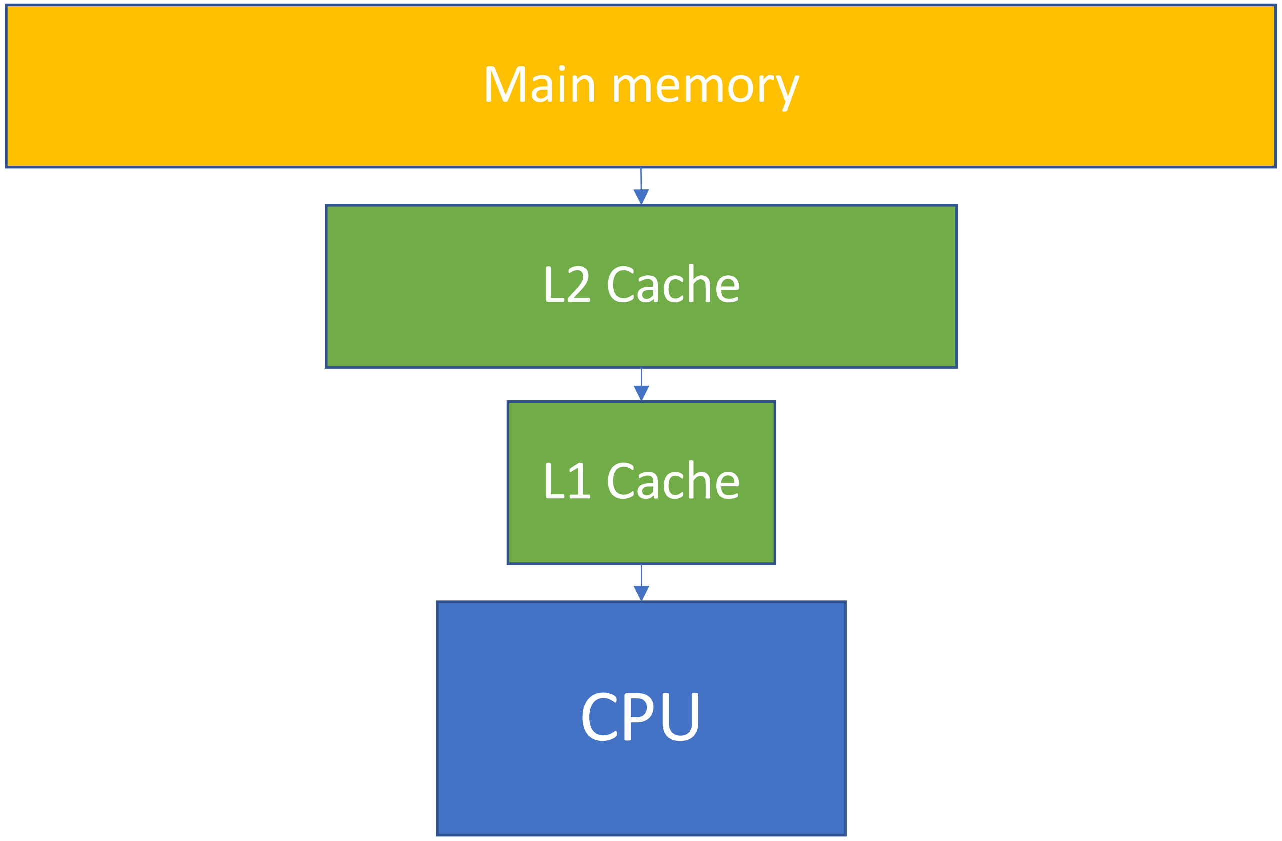
Side Channels



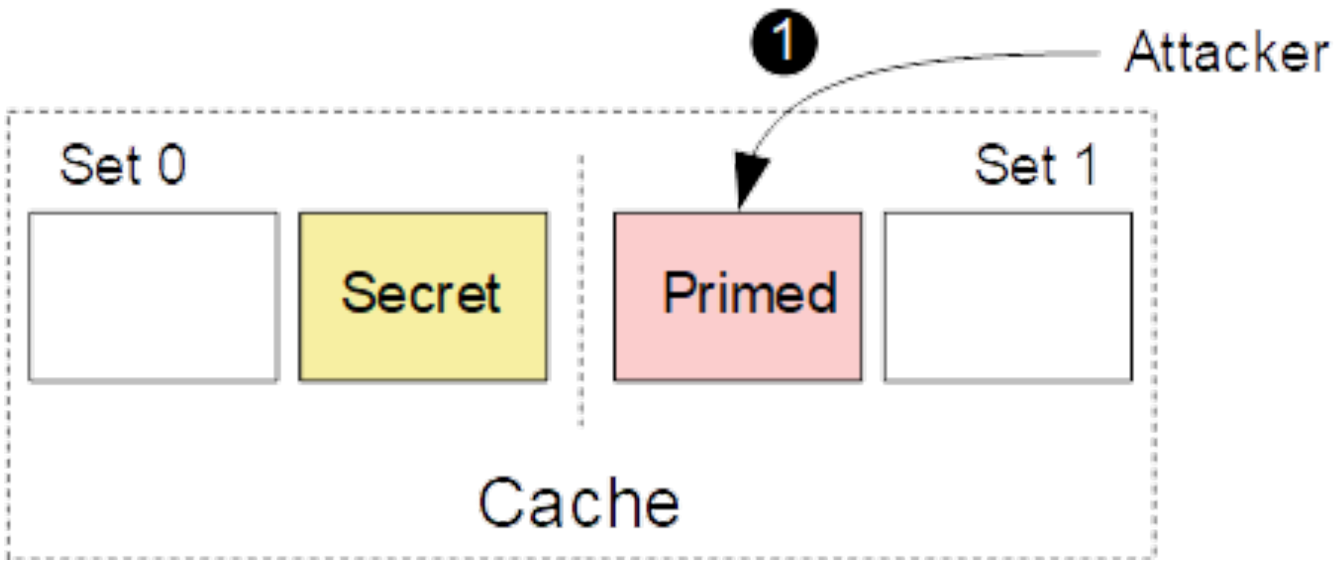
Example Channels

- Cache
- Disk timing
- Instruction timing
- CPU utilisation
- Clock frequency
- Power consumption
- Even erroneous behaviour e.g. differential analysis
- RF emissions (e.g. from monitors)

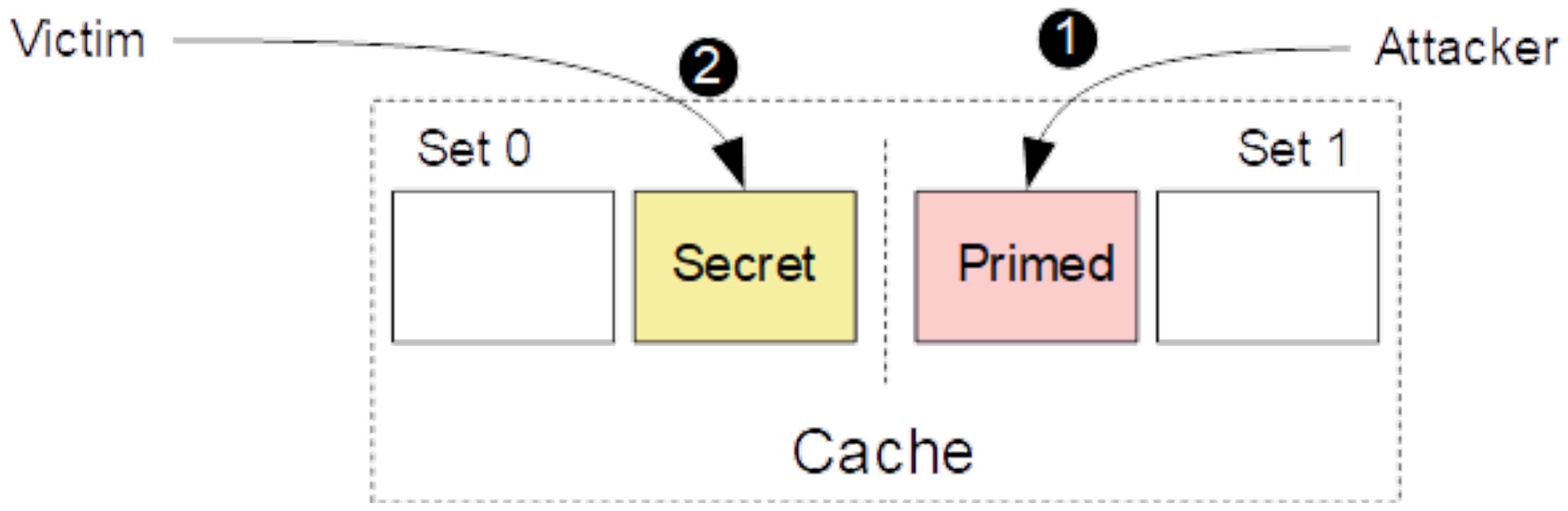
Cache Hierarchy



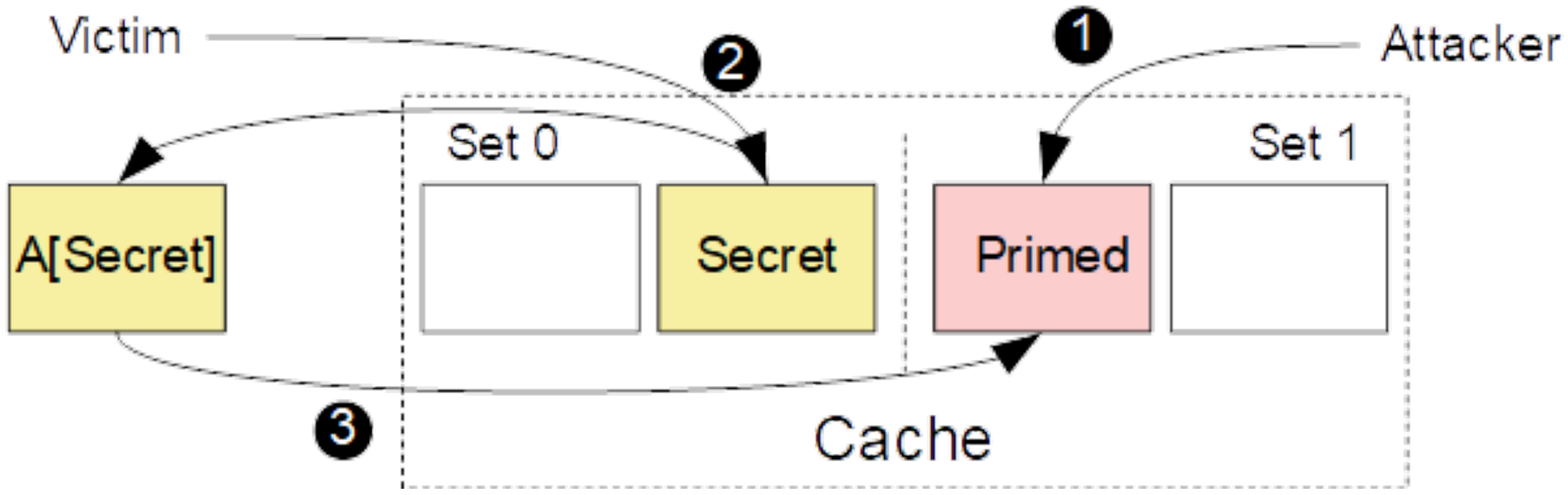
Prime and Probe Cache Attack



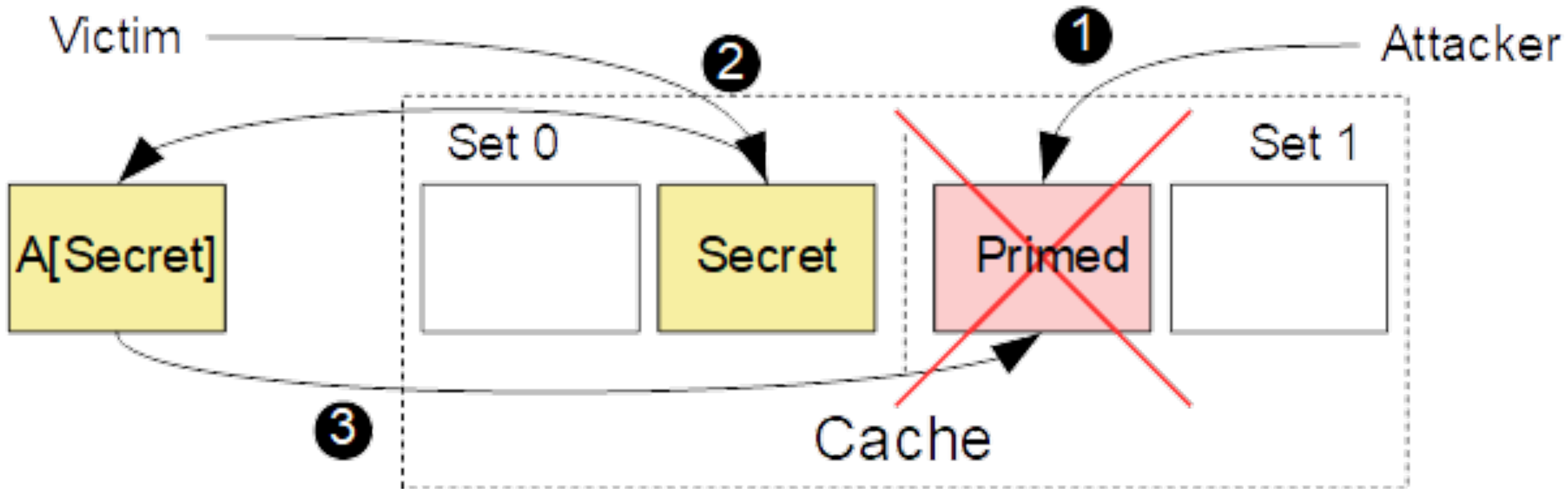
Prime and Probe Cache Attack



Prime and Probe Cache Attack



Prime and Probe Cache Attack



Example: AES Attack

- Read in S-Box from Memory

AES S-Box

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	Xa	Xb	Xc	Xd	Xe	Xf
0X	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1X	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2X	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3X	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4X	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5X	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6X	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7X	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8X	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9X	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
aX	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
bX	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
cX	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
dX	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
eX	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
fX	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Instructions can leak as well!

- E.g. 1 * 1 vs 2338572 * 2314908
- Arm v8.4-A Data Independent Timing flag
- No branching on secrets, either

Optimising Compilers

```
{
Load Sbox[0] – Sbox[128];
Return Sbox[Secret];
}
```

Optimising Compilers

```
{
Load Sbox[0] – Sbox[128];
Return Sbox[Secret];
}
```

Optimising Compilers

```
{
Return Sbox[Secret];
}
```

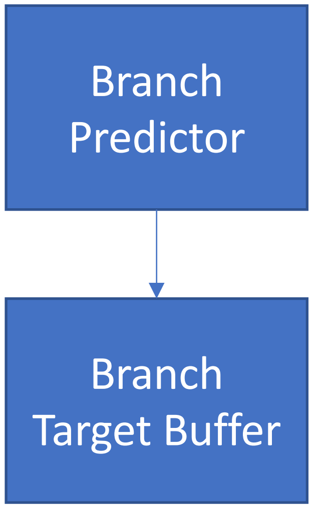
Hardware AES

- x86: Advanced Encryption Standard New Instructions – AES-NI
- Arm: Cryptographic Extensions in Aarch64, and/or separate crypto accelerators.

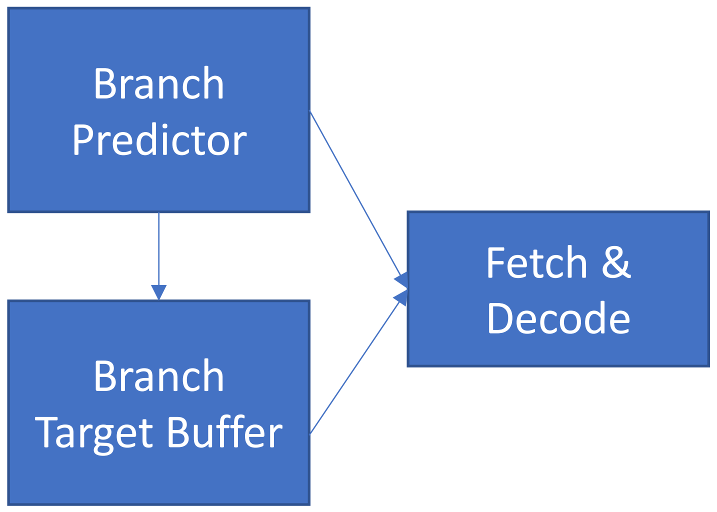
Speculative Side Channel Attacks

- E.g. Meltdown / Spectre
- Not the program that leaks anymore – it’s the speculative execution the processor does!

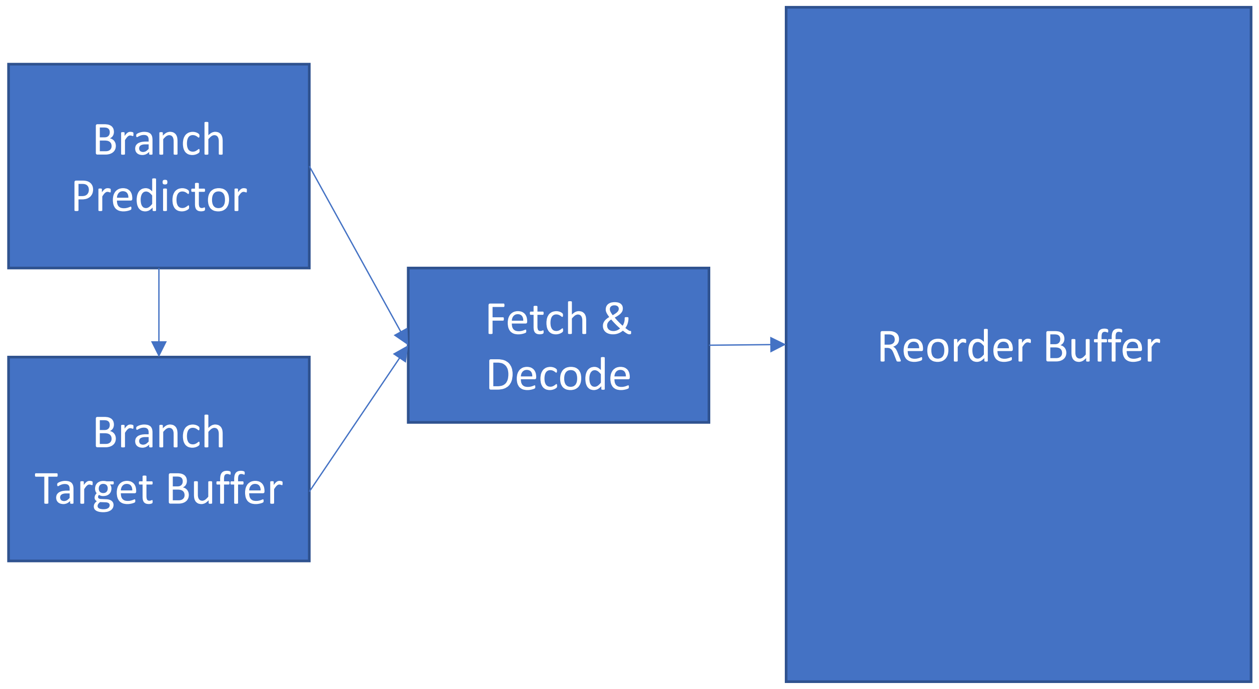
Speculative Execution



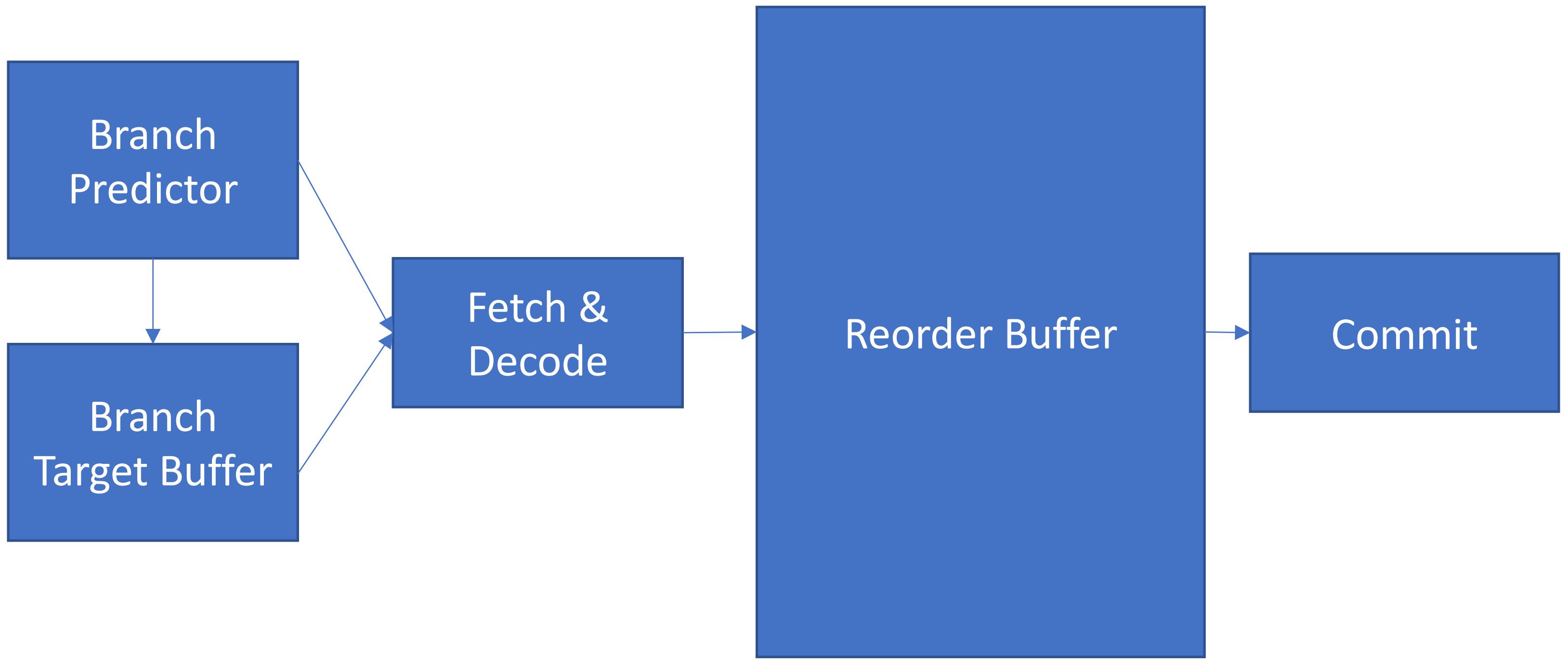
Speculative Execution



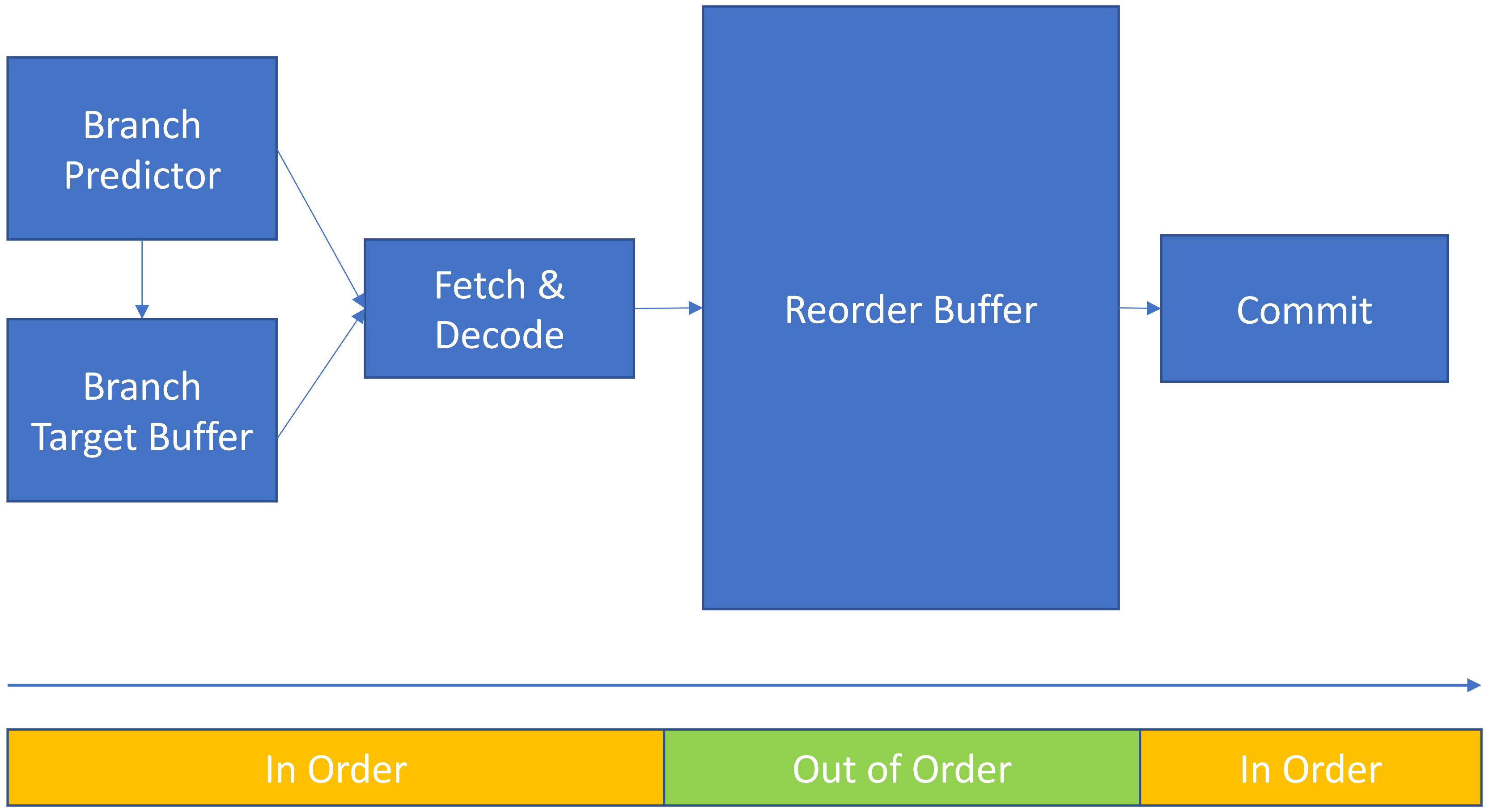
Speculative Execution



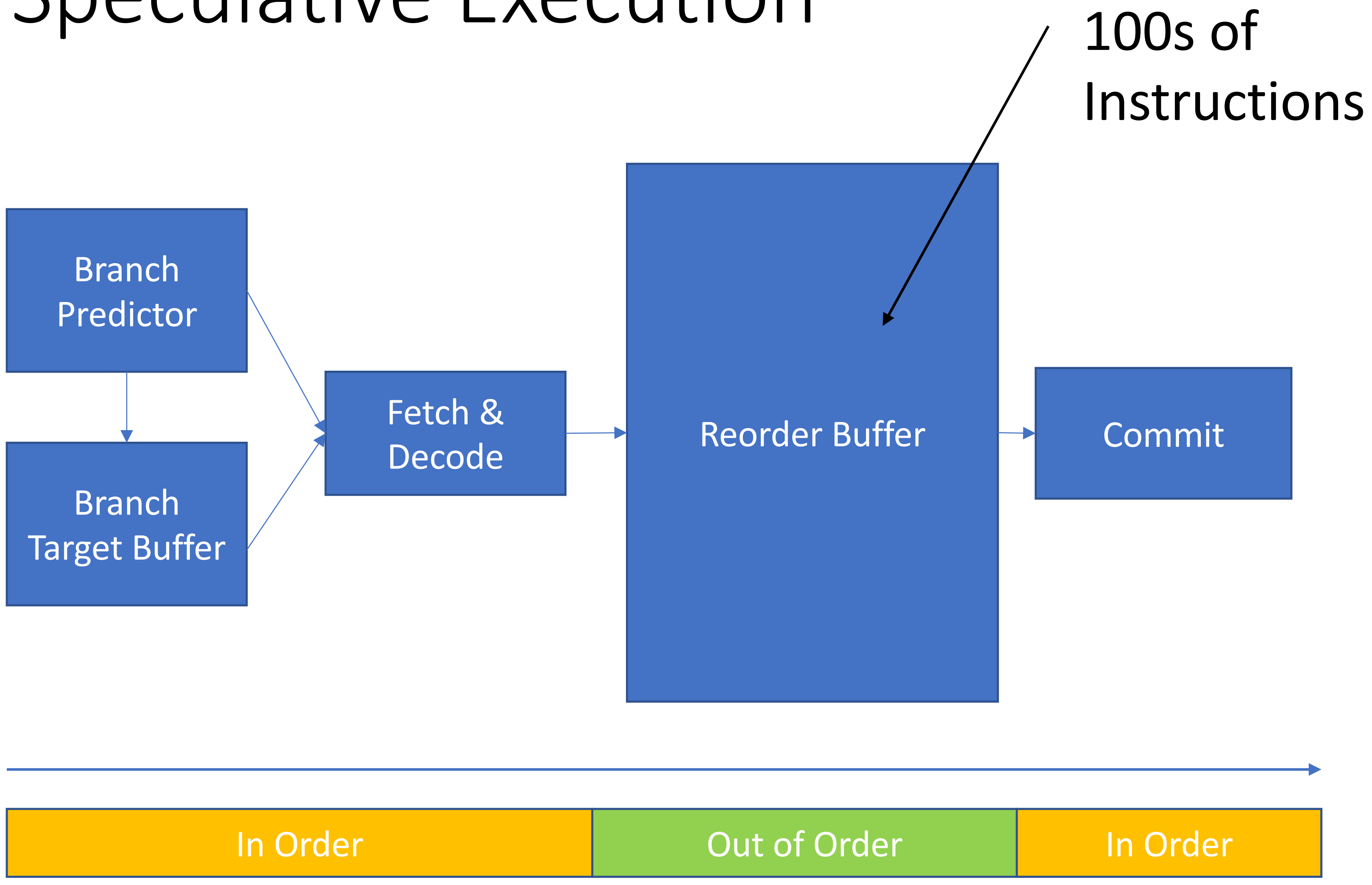
Speculative Execution



Speculative Execution



Speculative Execution



Meltdown

```
Flush (array);
```

Meltdown

```
Flush (array);
Try {
  Int x = *secret_banned_data;
  Int y = array[x];
} Catch (Exception E) {
  printf(“the above never happened, right?”);
}
```

Meltdown

```
Flush (array);
Try {
  Int x = *secret_banned_data;
  Int y = array[x];
} Catch (Exception E) {
  printf("the above never happened, right?");
}
```

Programmer Model

Meltdown

```
Flush (array);
Try {
  Int x = *secret_banned_data;
  ~~Int y = array[x];~~
} Catch (Exception E) {
  printf(“the above never happened, right?”);
}
```

Actual

Meltdown

```
Flush (array);
Try {
  Int x = *secret_banned_data;
  ~~Int y = array[x];~~
} Catch (Exception E) {
  printf(“the above never happened, right?”);
}
```

Actual

Meltdown

Flush (array);

Try {

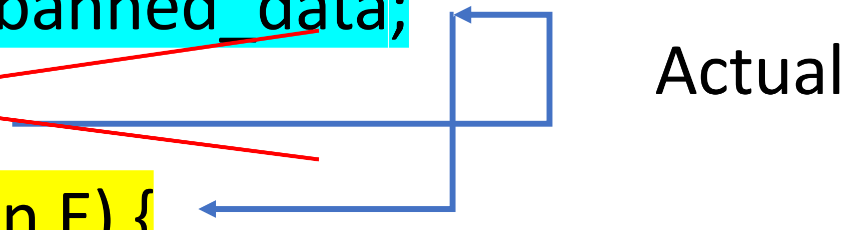
  Int x = *secret_banned_data;

  ~~Int y = array[x];~~

} Catch (Exception E) {

  printf("the above never happened, right?");

}



Meltdown

```
Flush (array);
Try {
  Int x = *secret_banned_data;
  Int y = array[x];
} Catch (Exception E) {
  printf("the above never happened, right?");
}
 for(int z=0; z<size; z++) {
 time(array[z]);
}
```

Meltdown

- Effectively a bug
- Fixed using Kernel Page Table Isolation

Spectre v1

```
Int x = index_of_secret_out_of_bounds_data;
If(x < array_size) {
  y = array[x];
  z = array2[y];
}
```

Spectre v1

```
Int x = index_of_secret_out_of_bounds_data;
If(x < array_size) {
  y = array[x];
  z = array2[y];
}
```

X = 928309183902

Array_size = 100

Spectre v1

```
Int x = index_of_secret_out_of_bounds_data;
If(x < array_size) {
  y = array[x];
  z = array2[y];
}
```

True execution: No

X = 928309183902

Array_size = 100

Spectre v1

```
Int x = index_of_secret_out_of_bounds_data;
If(x < array_size) {
  y = array[x];
  z = array2[y];
}
```

True execution: No

Branch prediction: Yes

X = 928309183902

Array_size = 100

Spectre v1

```
Int x = index_of_secret_out_of_bounds_data;
If(x < array_size) {
  y = array[x];
  z = array2[y];
}
```

True execution: No

Branch prediction: Yes

X = 928309183902

Array_size = 100

Leaks (Partial) Contents of Y

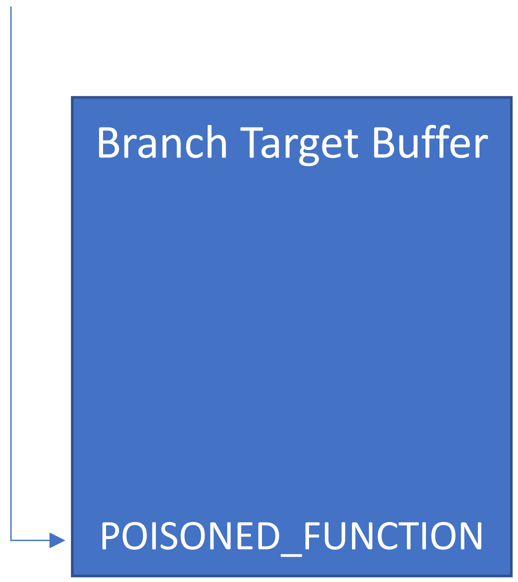
Spectre v2

```
Int x = index_of_secret_out_of_bounds_data;
Call_safe_function();
```

Spectre v2

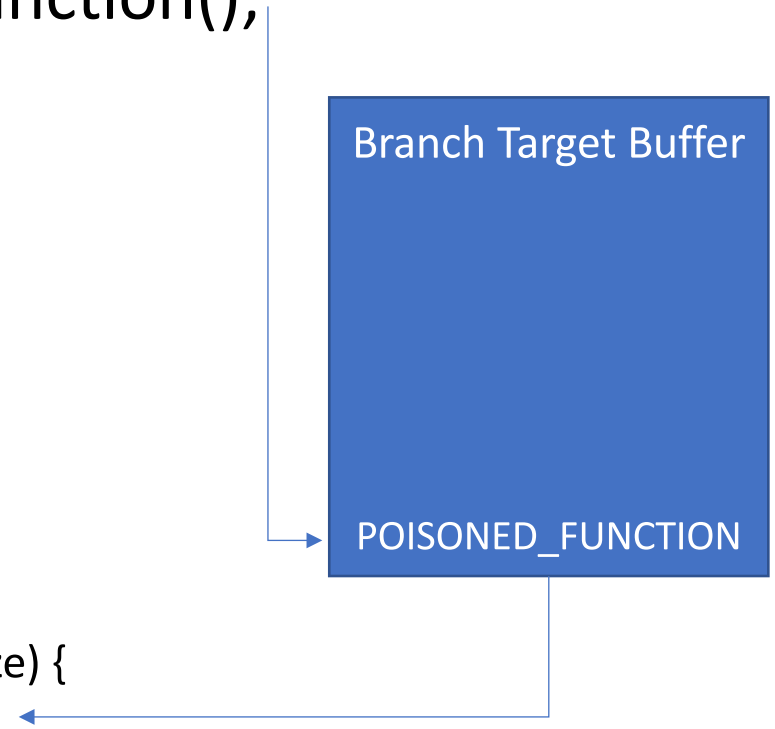
Int x = index_of_secret_out_of_bounds_data;

Call_safe_function();



Spectre v2

```
Int x = index_of_secret_out_of_bounds_data;
Call_safe_function();
```



```
If(x < array_size) {
  y = array[x];
  z = array2[y];
}
```

How can we use Spectre?

- Sandbox Escape
- Inter Process Communication

Conclusions

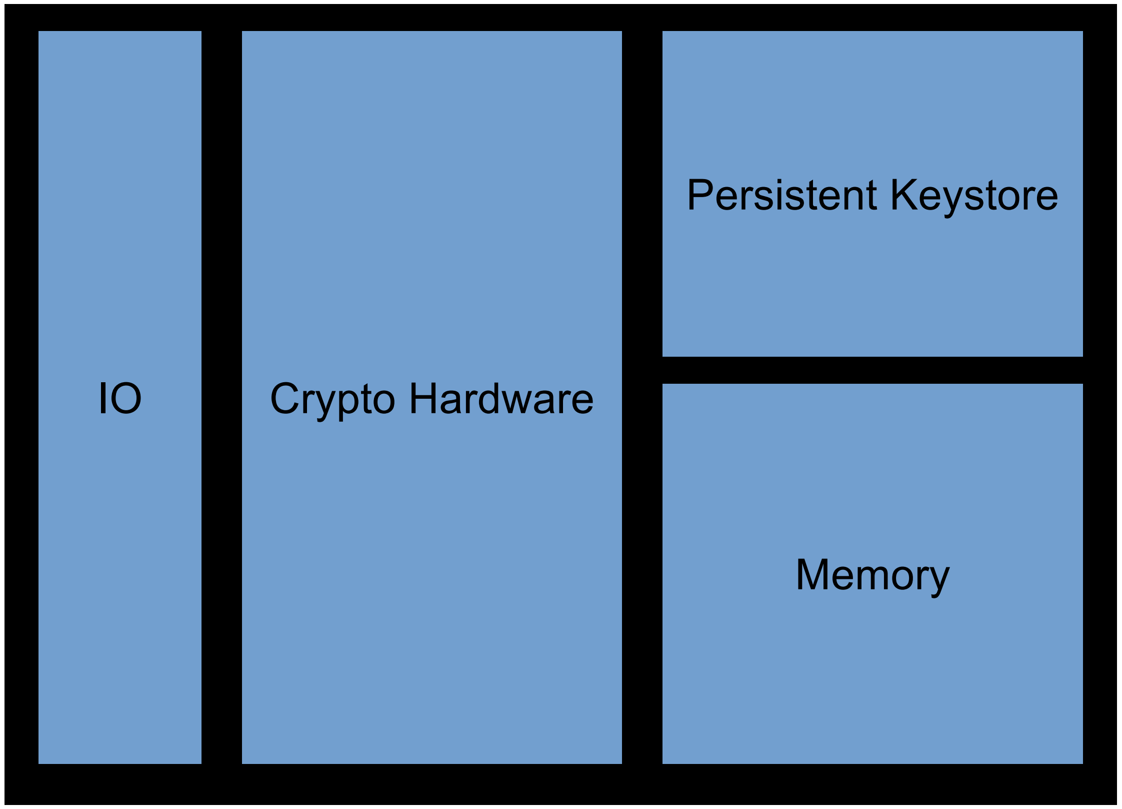
- Security isn't just limited to the “programmer's model”
- Don’t “roll your own crypto”
- Bugs can be around for a long time before they are discovered
- Make sure you’re aware of what the hardware is doing underneath your code!

What can hardware provide for YOU?

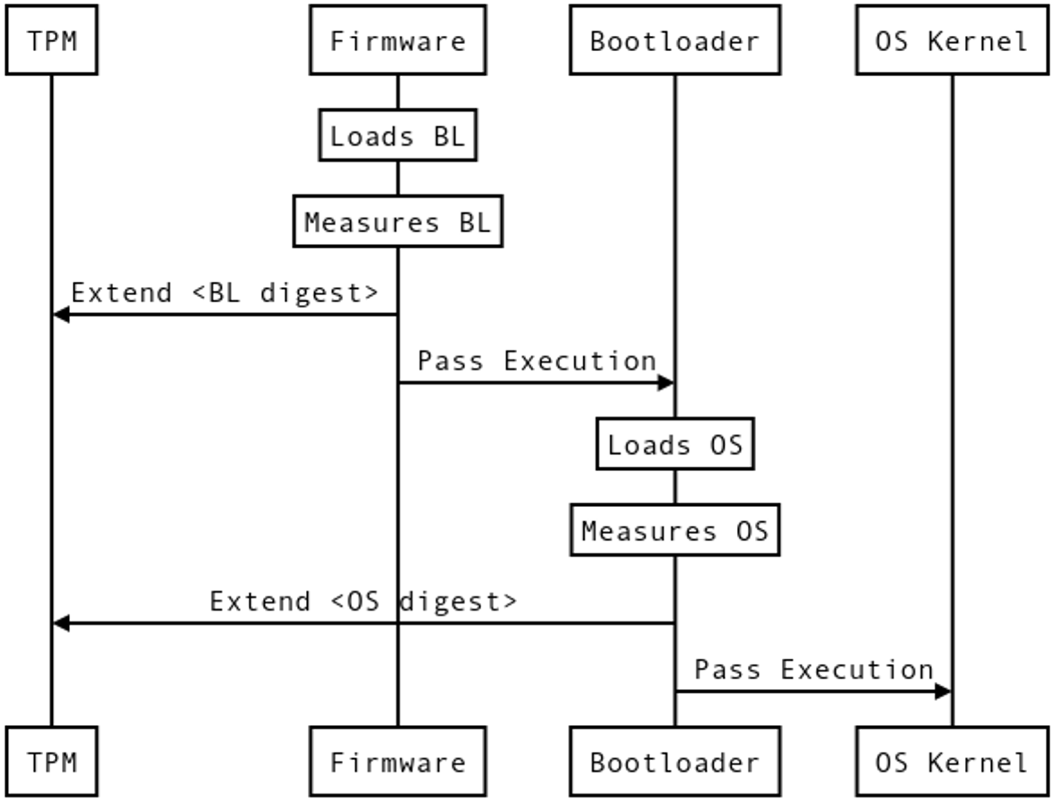
What can hardware provide for YOU?

- Hardware AES
- Trusted Platform Modules
- Enclaves
- Physically Unclonable Functions
- Codesign for Software Security: CHERI and MTE

Trusted Platform Module

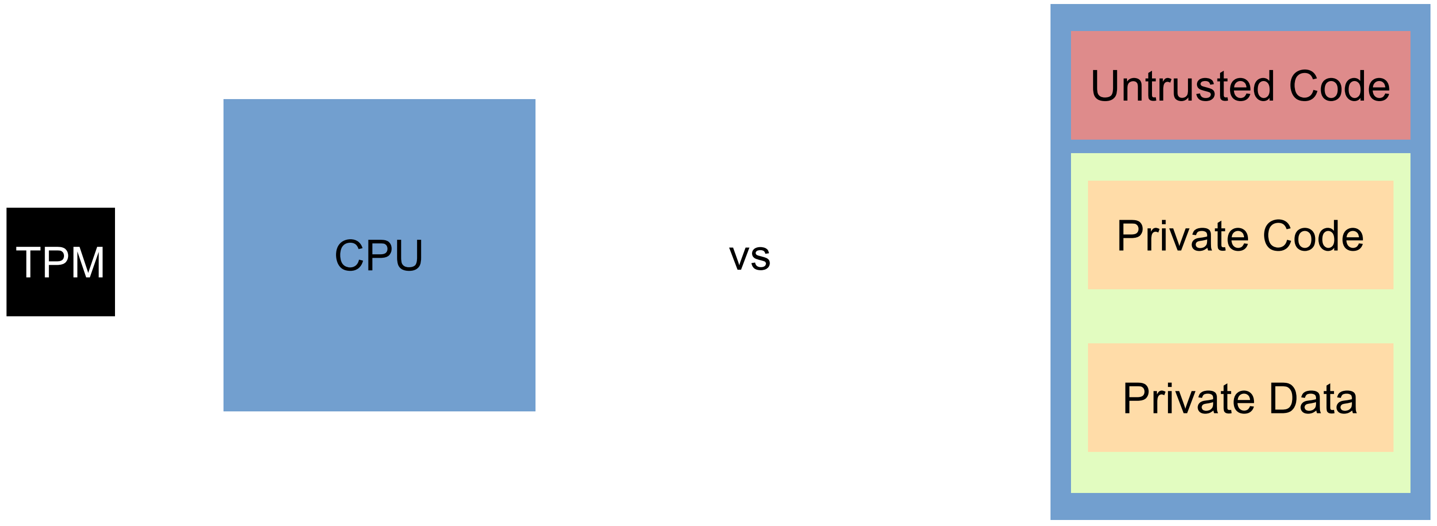


Chain of Verification

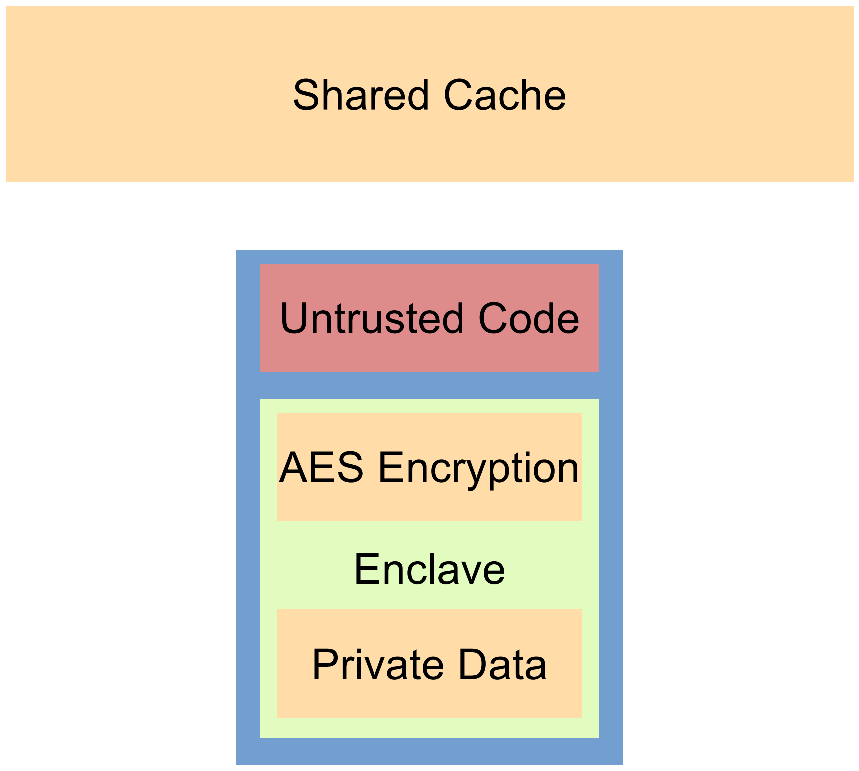


From google.github.io/tpm-js/

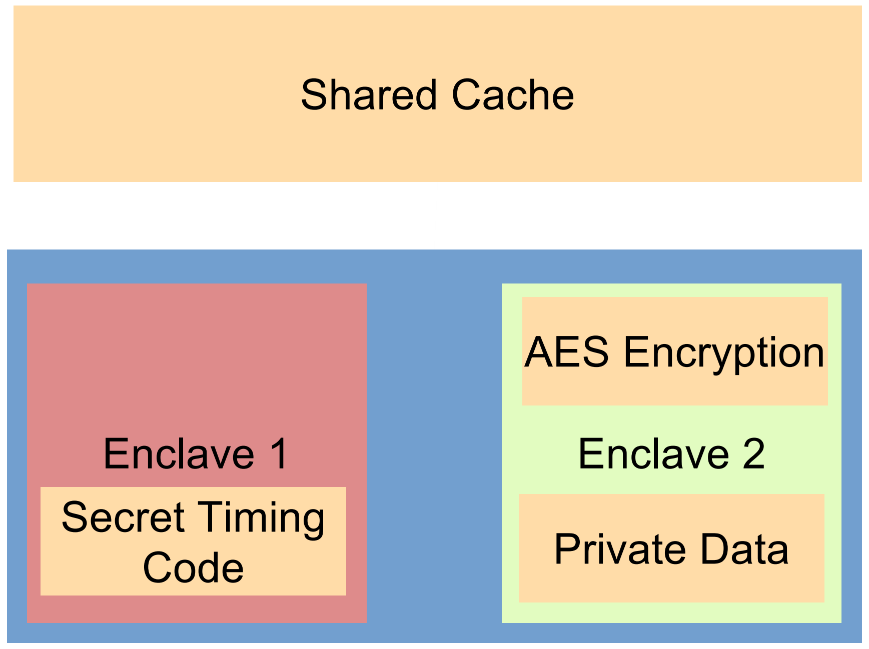
Enclaves (Trustzone, SGX, SEV)



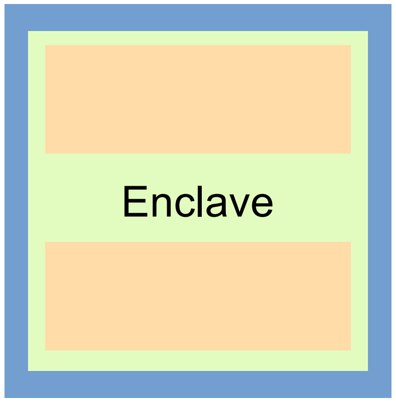
Issues: Side Channels



Issues: Side Channels

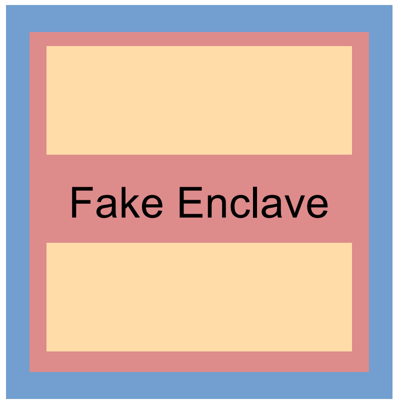


Issues: Am I really in an Enclave (SEV)?

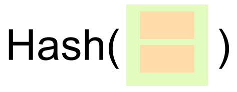


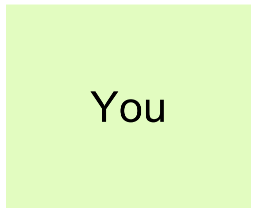
1. Downgrade Firmware
2. Leak Root Key through Signature Check Vulnerability
3. Profit

Issues: Am I really in an Enclave (SEV)?

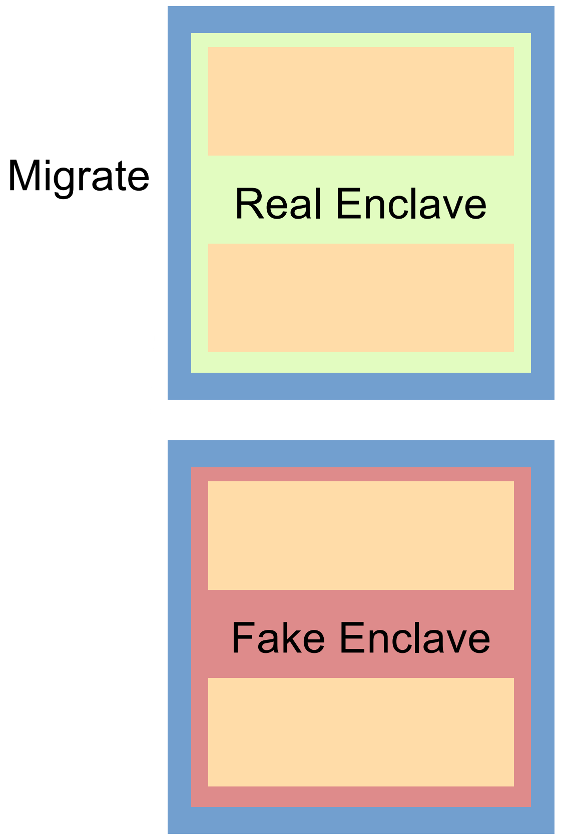


1. Downgrade Firmware
2. Leak Root Key through Signature Check Vulnerability
3. Profit



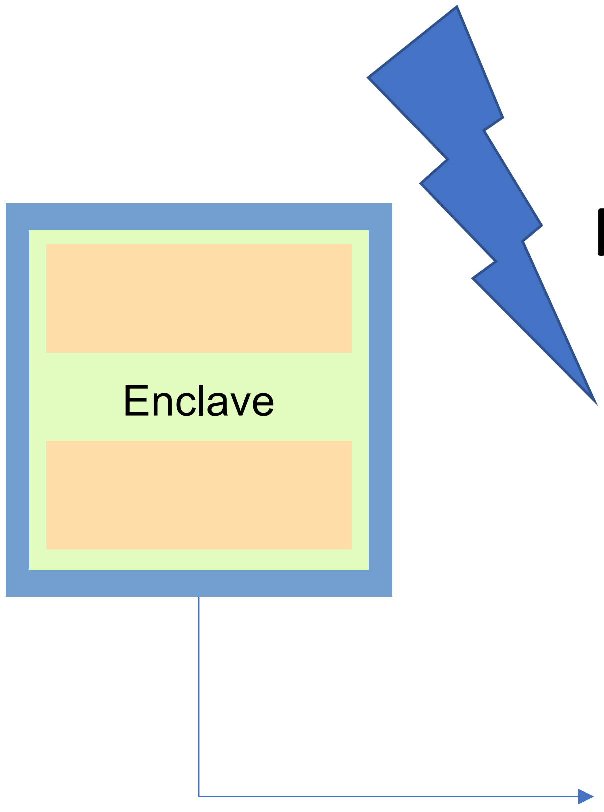


Issues: Am I really in an Enclave (SEV)?



1. Downgrade Firmware
2. Leak Root Key through Signature Check Vulnerability
3. Profit

Plundervolt

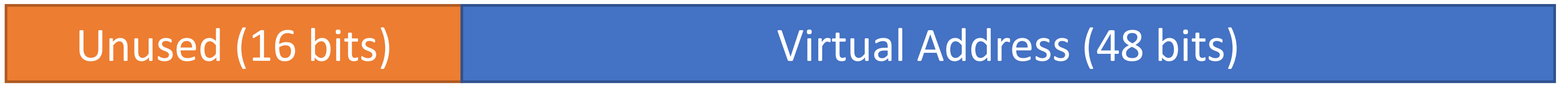


Root-user Undervolt

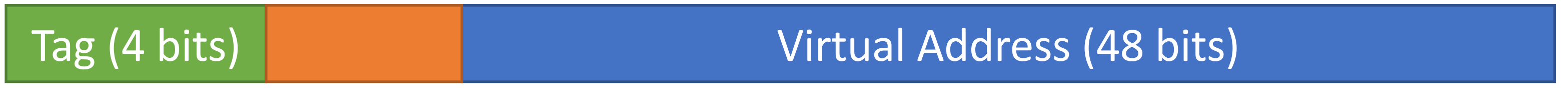
Extract keys of AES,
Or redirect pointers to
Outside enclave

Codesign for Security: CHERI vs MTE

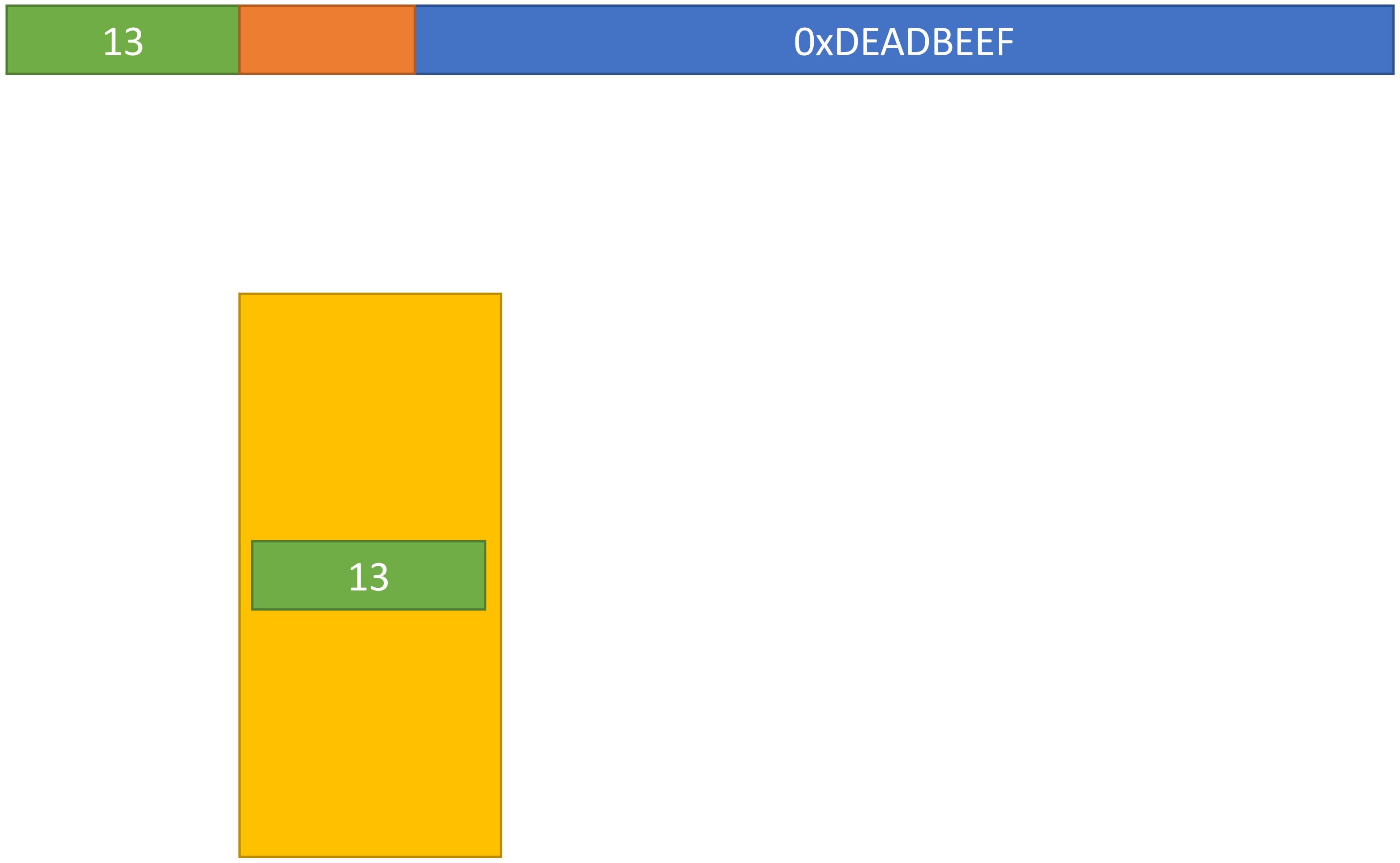
MTE (Memory Tagging Extension)



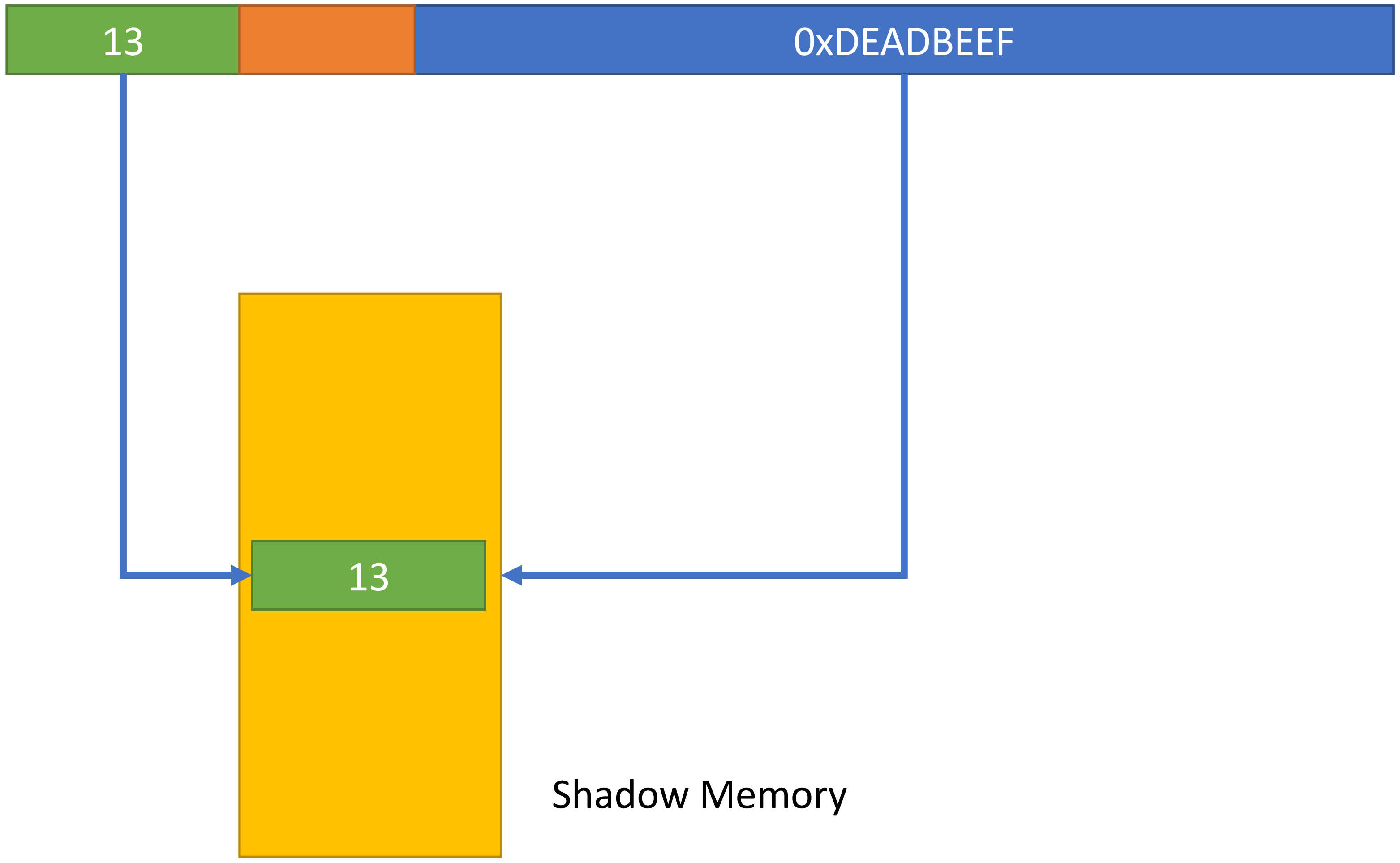
MTE (Memory Tagging Extension)



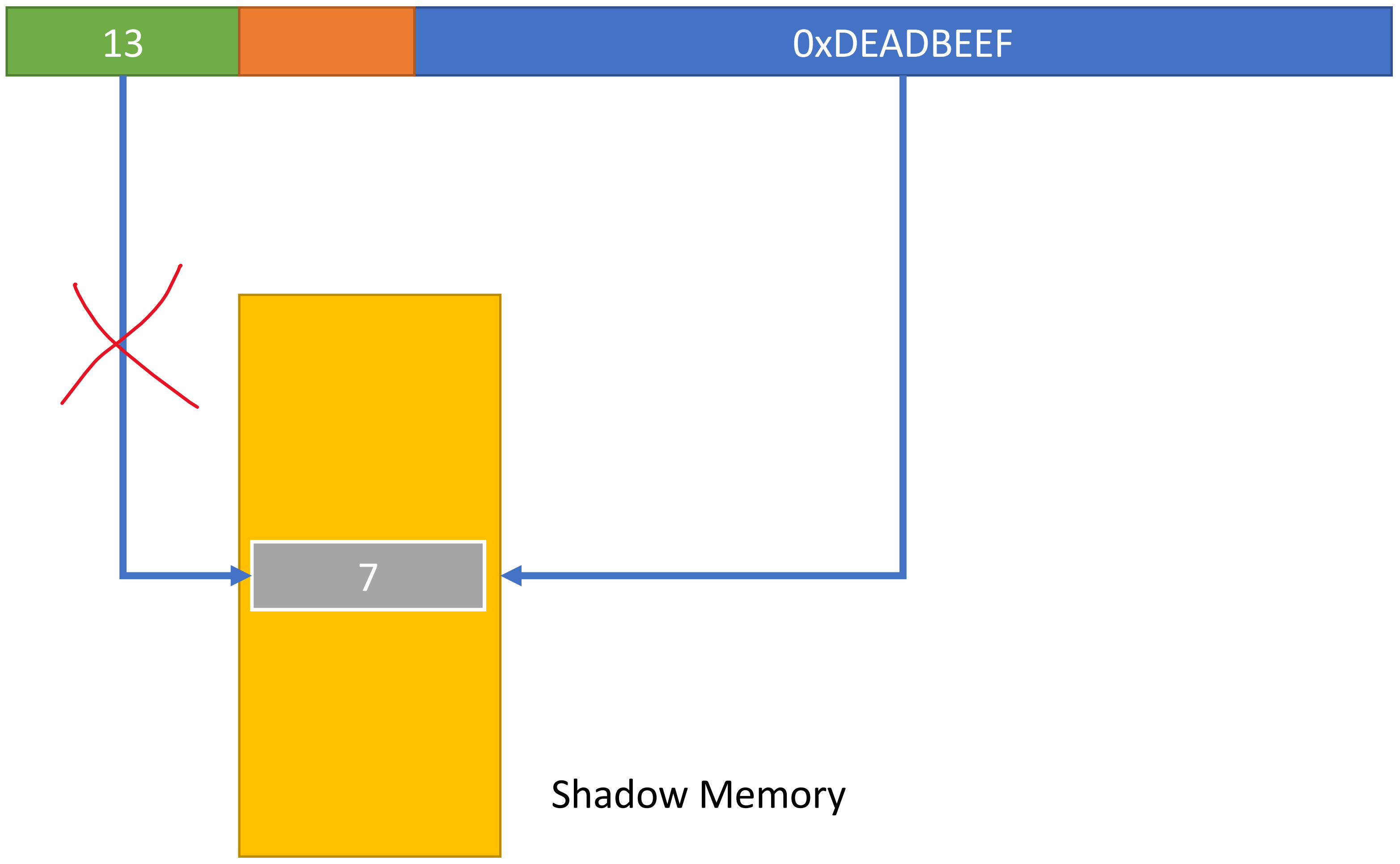
MTE (Memory Tagging Extension)



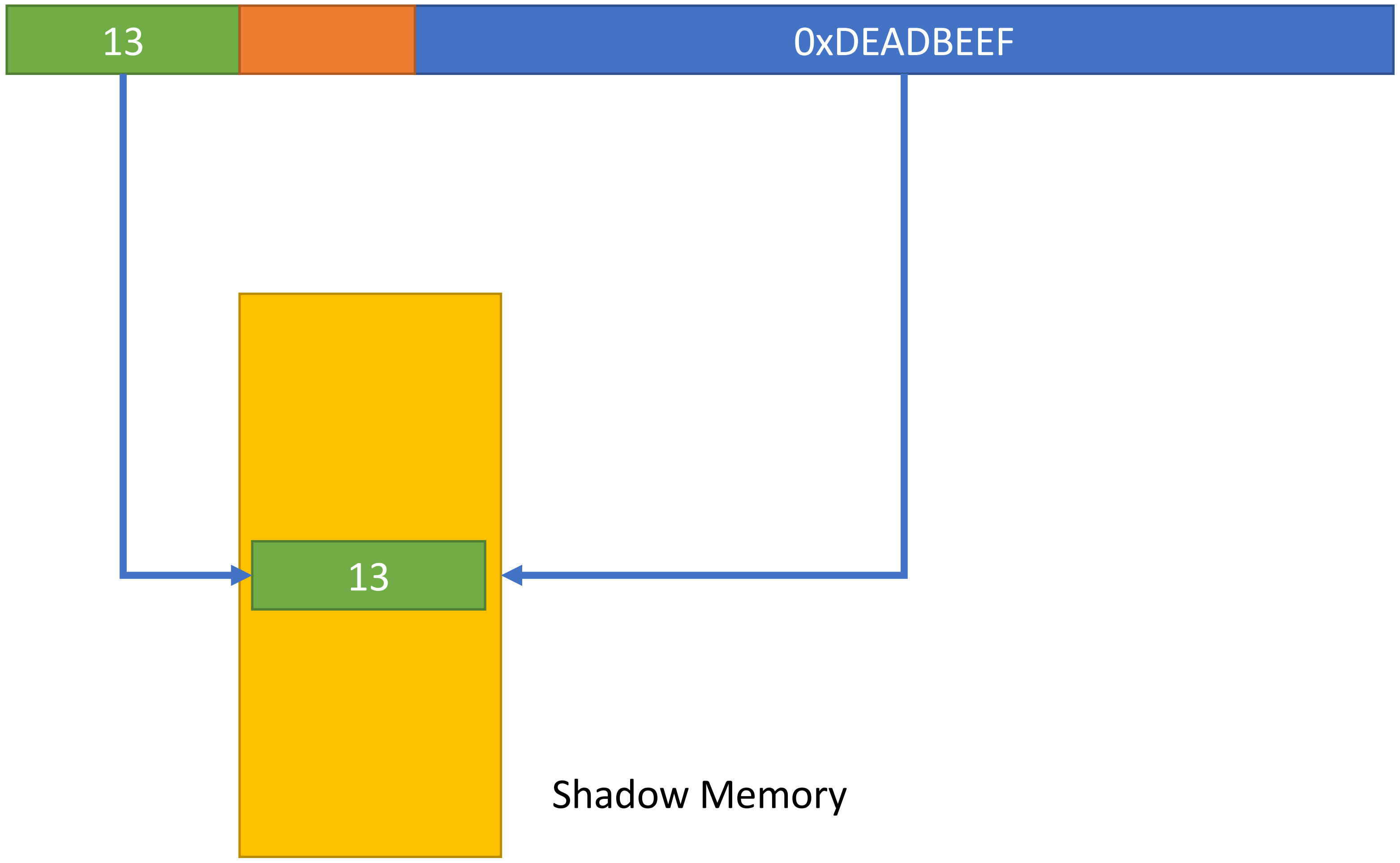
MTE (Memory Tagging Extension)



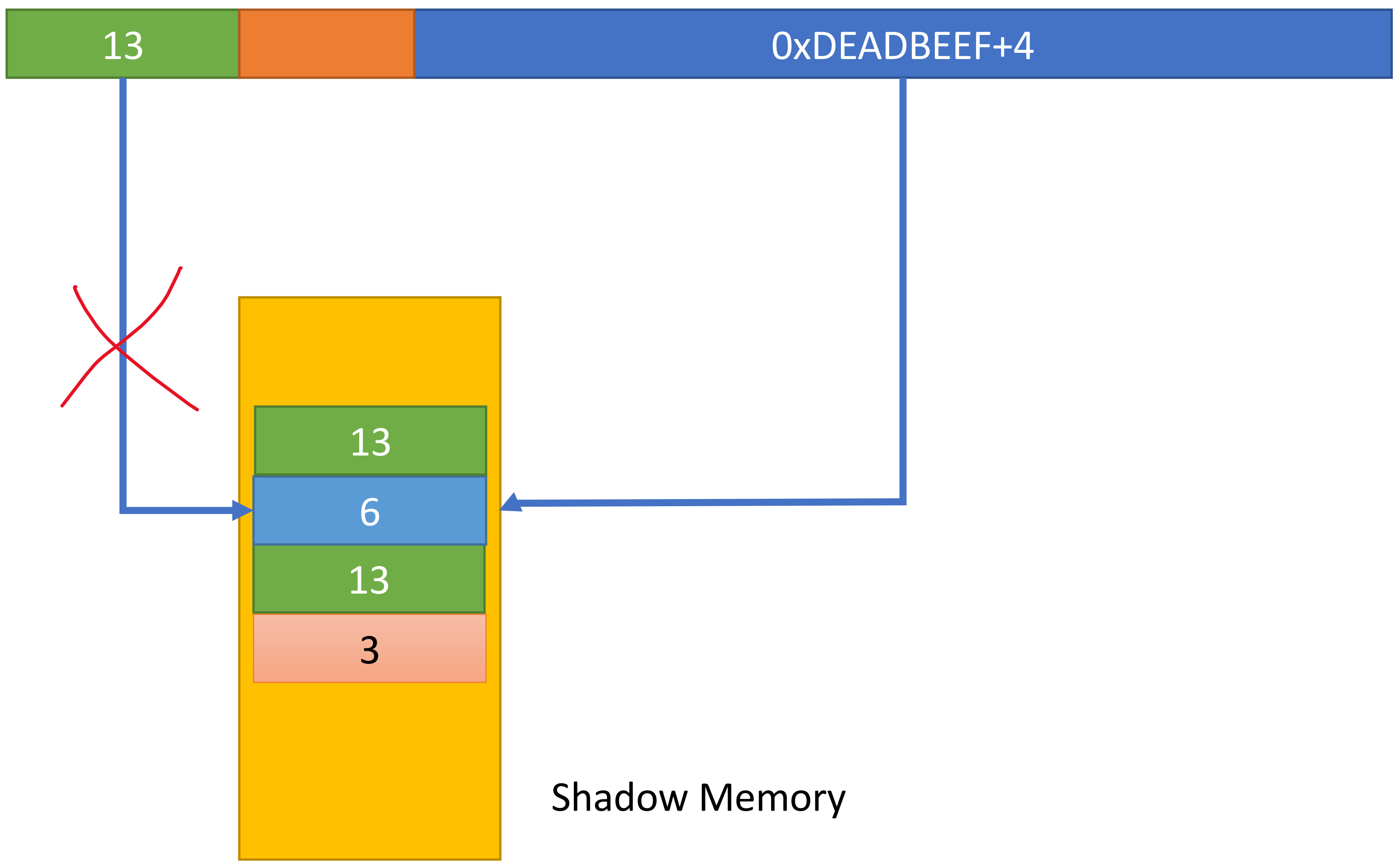
MTE (Memory Tagging Extension)



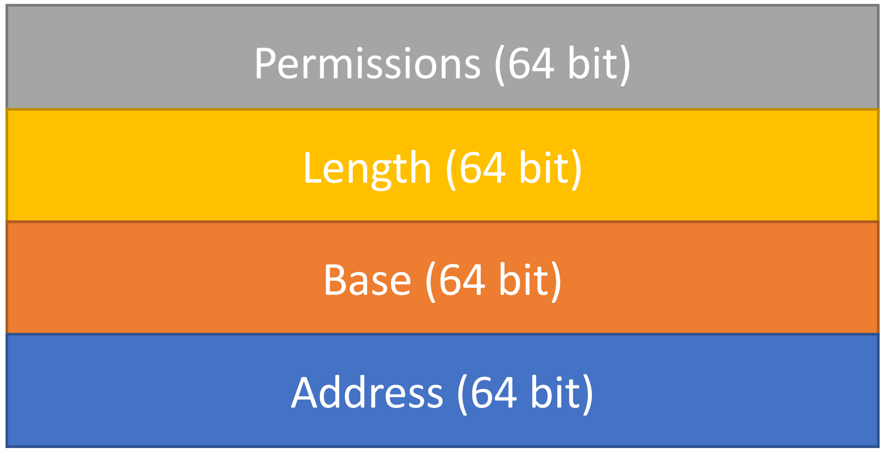
MTE (Memory Tagging Extension)



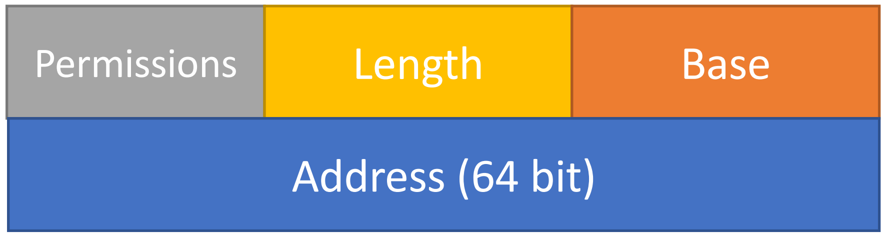
MTE (Memory Tagging Extension)



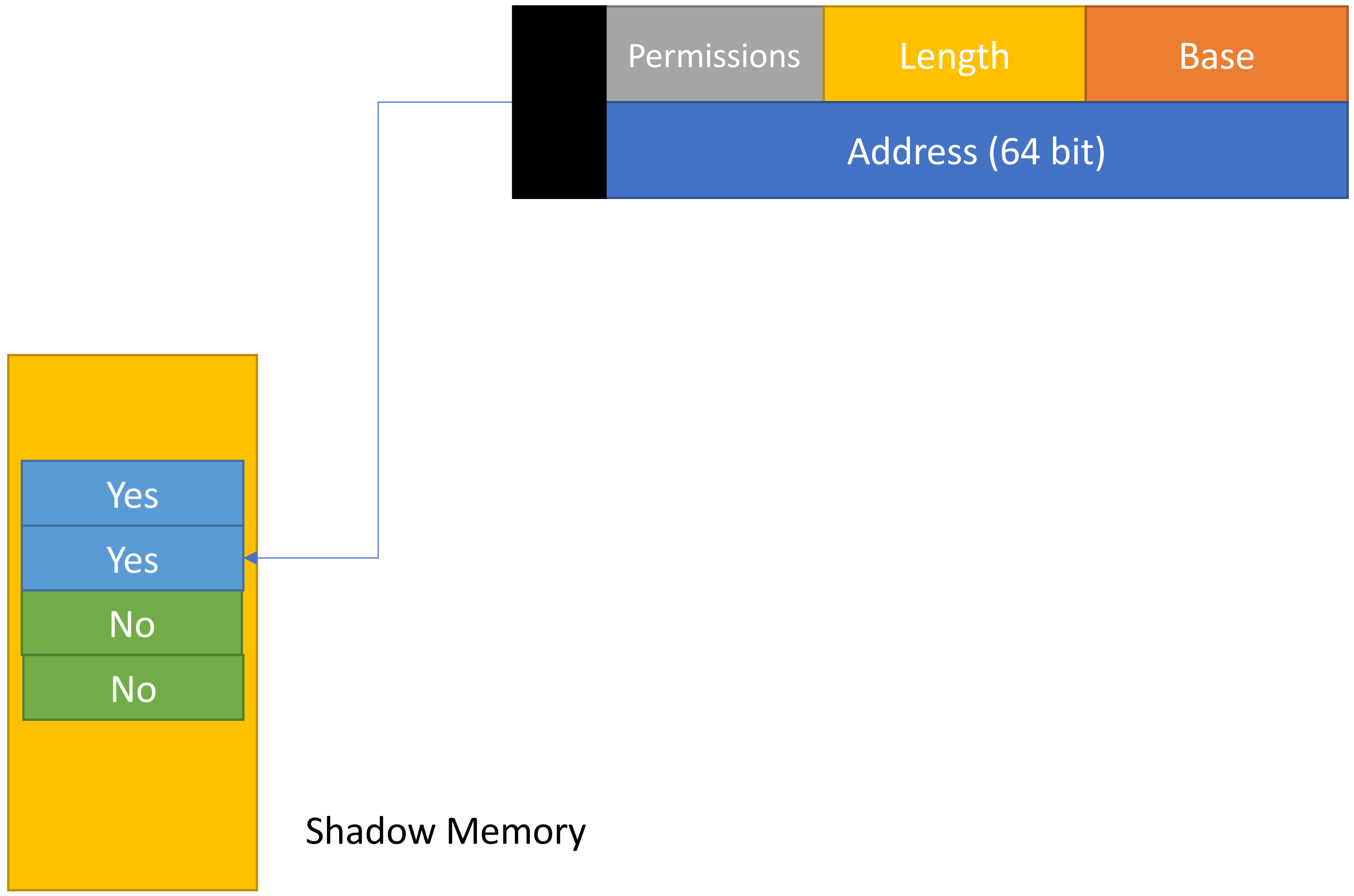
CHERI Capabilities



CHERI Capabilities



CHERI Capabilities



CHERI	MTE
+ Guaranteed mitigation of spatial safety bugs (in pure-cap mode)	- Probabilistic mitigation of temporal/spatial bugs
+ 1-bit/128 shadow space	- 4-bit/128 shadow space
- 128-bit pointers	+ 64-bit pointers
- Standards-incompatible	+ Standards-compatible

Further Reading

- Security Engineering Chapters 18, 19, 5, 27
- Flipping Bits in Memory Without Accessing Them: http://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf
- https://meltdownattack.com/
- https://plundervolt.com/
- CHERI Concentrate https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2019tc-cheri-concentrate.pdf
- https://google.github.io/tpm-js/
- What you get is what you C: https://www.cl.cam.ac.uk/~rja14/Papers/whatyouc.pdf