

CPU Vulnerabilities

Finding Bugs Where Software Ends

Michael Schwarz | FOSAD Summer School 2026



It's Me Again!

You might know me from
lectures like ...

CPU Vulnerabilities Finding Bugs Where Software Ends

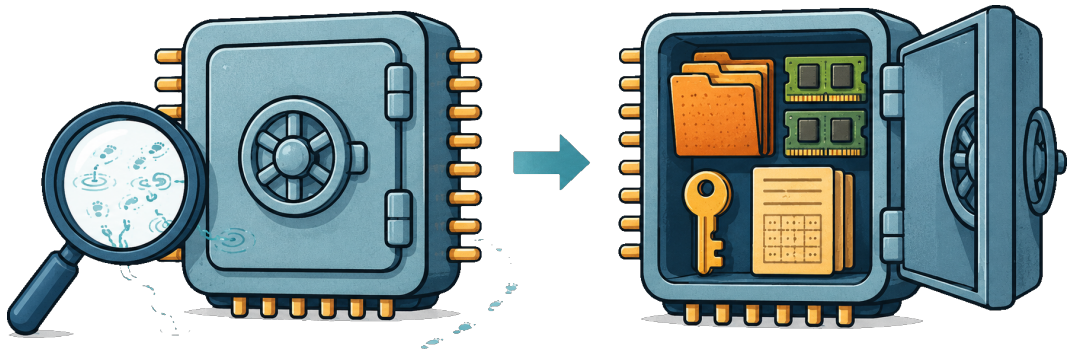
Yesterday's talk



Dr. Michael Schwarz
Still Faculty @ CISPA



Welcome back



Yesterday we searched for meta-data leakage. Today we want to get actual data.

**“Data leakage? Someone
would have seen that”**





Start with Strong Attackers

REMOVE PRIVILEGES



Malicious hypervisor controls VM execution and memory

STRENGTHEN THE ATTACK





Start with Strong Attackers

REMOVE PRIVILEGES



Malicious hypervisor controls VM execution and memory

Operating-system attacker kernel privileges on the host

STRENGTHEN THE ATTACK





Start with Strong Attackers

REMOVE PRIVILEGES



Malicious hypervisor controls VM execution and memory

Operating-system attacker kernel privileges on the host

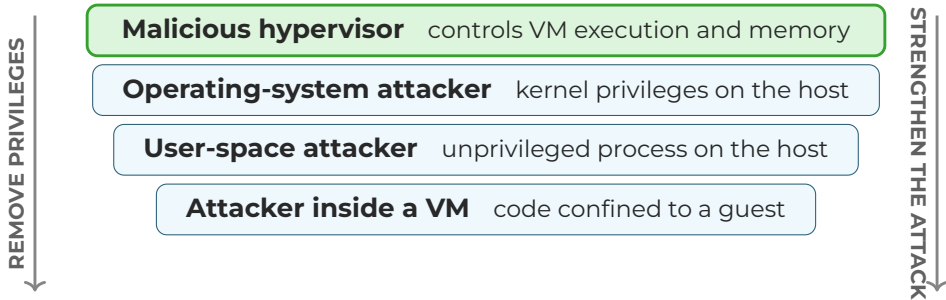
User-space attacker unprivileged process on the host

STRENGTHEN THE ATTACK



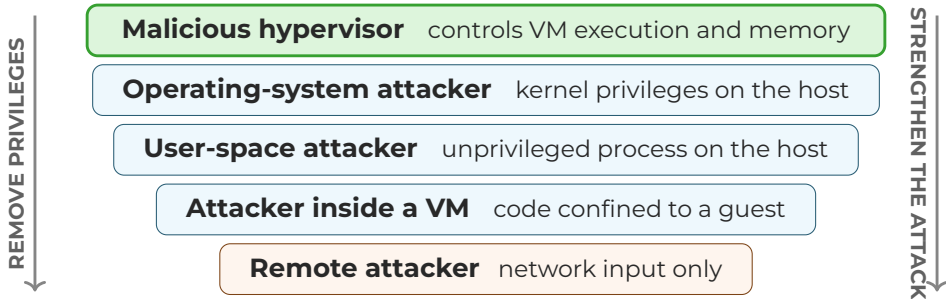


Start with Strong Attackers





Start with Strong Attackers

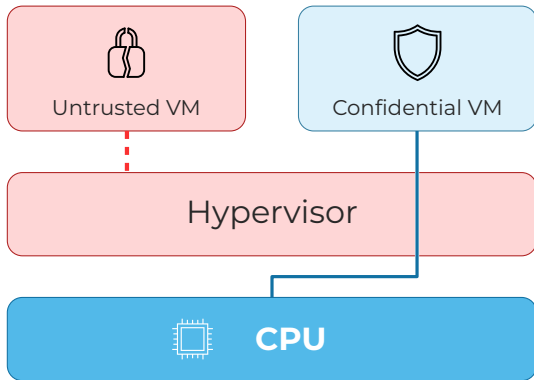


A strong threat model is a starting point, not the final result

maximize control + observability → understand the primitive → generalize the attack



Confidential Virtual Machines







INVD Instruction



Flushes internal CPU caches



Does not write modifications to
memory



INVD Instruction



Flushes internal CPU caches



Does not write modifications to memory

Intel

Use this instruction with care. Data cached internally and not written back to main memory will be lost. [...] software should instead use the WBINVD instruction.



CacheWarp: Basic Primitive

Hypervisor

Victim VM

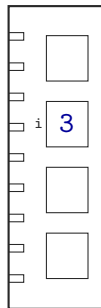
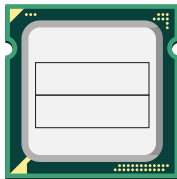
CPU Cache

DRAM

invd

```
▶ printf("%d", i);  
  i = 4;  
  printf("%d", i);
```

Output





CacheWarp: Basic Primitive

Hypervisor

Victim VM

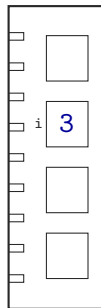
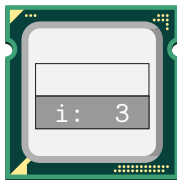
CPU Cache

DRAM

invd

```
▶ printf("%d", i);  
  i = 4;  
  printf("%d", i);
```

Output





CacheWarp: Basic Primitive

Hypervisor

Victim VM

CPU Cache

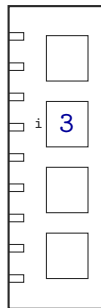
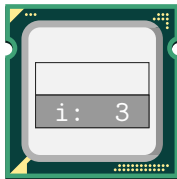
DRAM

invd

```
▶ printf("%d", i);  
  i = 4;  
  printf("%d", i);
```

Output

3





CacheWarp: Basic Primitive

Hypervisor

Victim VM

CPU Cache

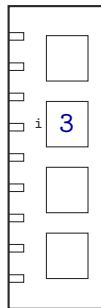
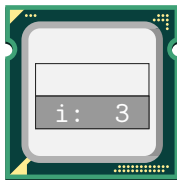
DRAM

invd

```
printf("%d", i);  
▶ i = 4;  
printf("%d", i);
```

Output

3





CacheWarp: Basic Primitive

Hypervisor

Victim VM

CPU Cache

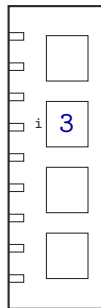
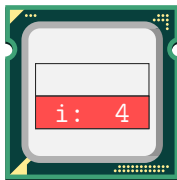
DRAM

invd

```
printf("%d", i);  
▶ i = 4;  
printf("%d", i);
```

Output

3





CacheWarp: Basic Primitive

Hypervisor

Victim VM

CPU Cache

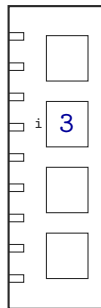
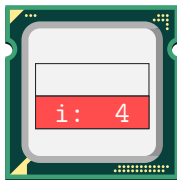
DRAM

▶ invd

```
printf("%d", i);  
i = 4;  
printf("%d", i);
```

Output

3





CacheWarp: Basic Primitive

Hypervisor

Victim VM

CPU Cache

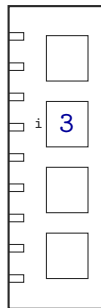
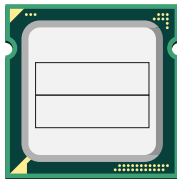
DRAM

▶ invd

```
printf("%d", i);  
i = 4;  
printf("%d", i);
```

Output

3





CacheWarp: Basic Primitive

Hypervisor

Victim VM

CPU Cache

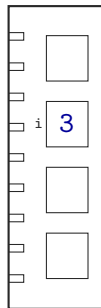
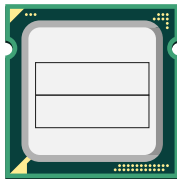
DRAM

invd

```
printf("%d", i);  
i = 4;  
▶ printf("%d", i);
```

Output

3





CacheWarp: Basic Primitive

Hypervisor

Victim VM

CPU Cache

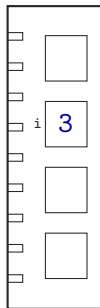
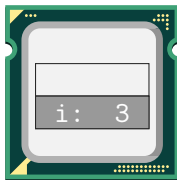
DRAM

invd

```
printf("%d", i);  
i = 4;  
▶ printf("%d", i);
```

Output

3





CacheWarp: Basic Primitive

Hypervisor

Victim VM

CPU Cache

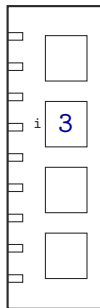
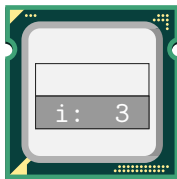
DRAM

invd

```
printf("%d", i);  
i = 4;  
▶ printf("%d", i);
```

Output

```
3  
3
```





sudo Exploit

```
int uid = 0;

uid = getuid();
if( uid == 0 ) {
    // continue as root
} else {
    // ask for password
}
```



sudo Exploit

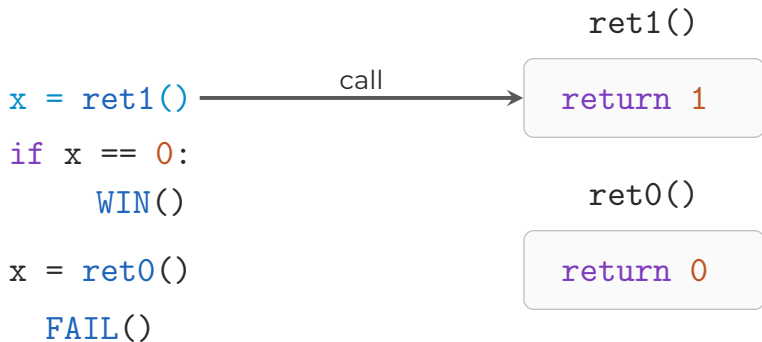
```
int uid = 0;

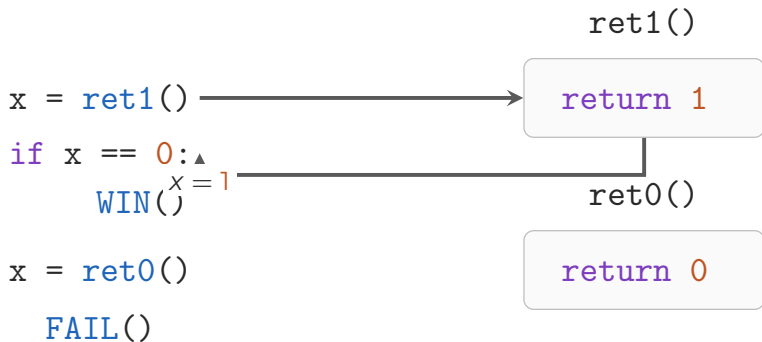
uid = getuid(); ← INVD
if( uid == 0 ) {
    // continue as root
} else {
    // ask for password
}
```



```
int uid = 0;

if( uid == 0 ) {
    // continue as root
} else {
    // ask for password
}
```





```
    x = ret1()  
    if x == 0:  
        WIN()  
false → x = ret0()  
        FAIL()
```

ret1()

return 1

ret0()

return 0

```
x = ret1()  
if x == 0:  
    WIN()  
x = ret0()  
    FAIL()
```

call

ret1()

return 1

ret0()

return 0

```
x = ret1()
```

```
if x == 0:
```

```
    WIN()
```

```
x = ret0()
```

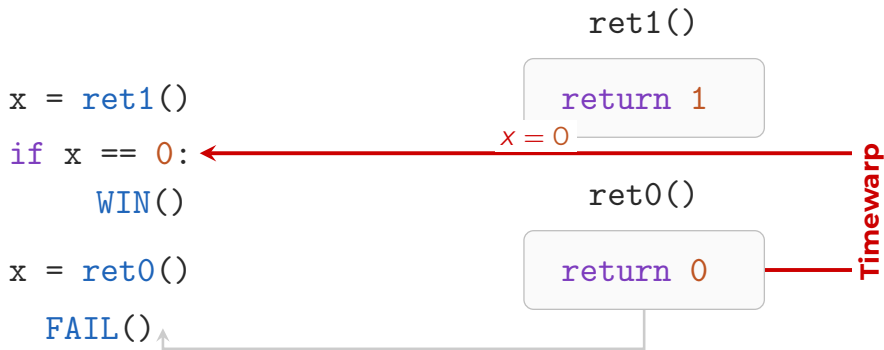
```
FAIL() ▲ normal execution  
      x = 0
```

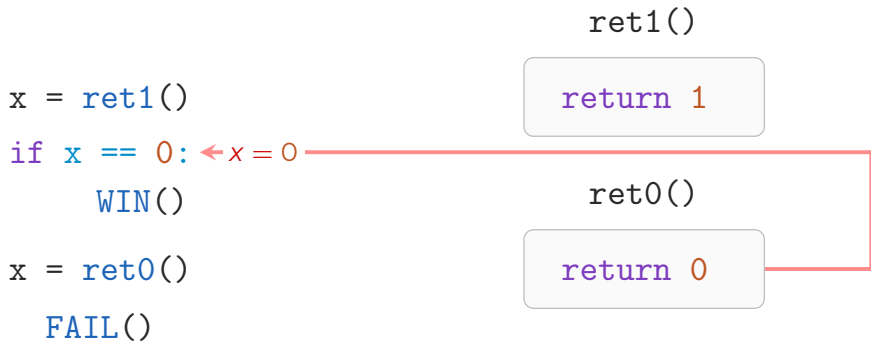
```
ret1()
```

```
return 1
```

```
ret0()
```

```
return 0
```





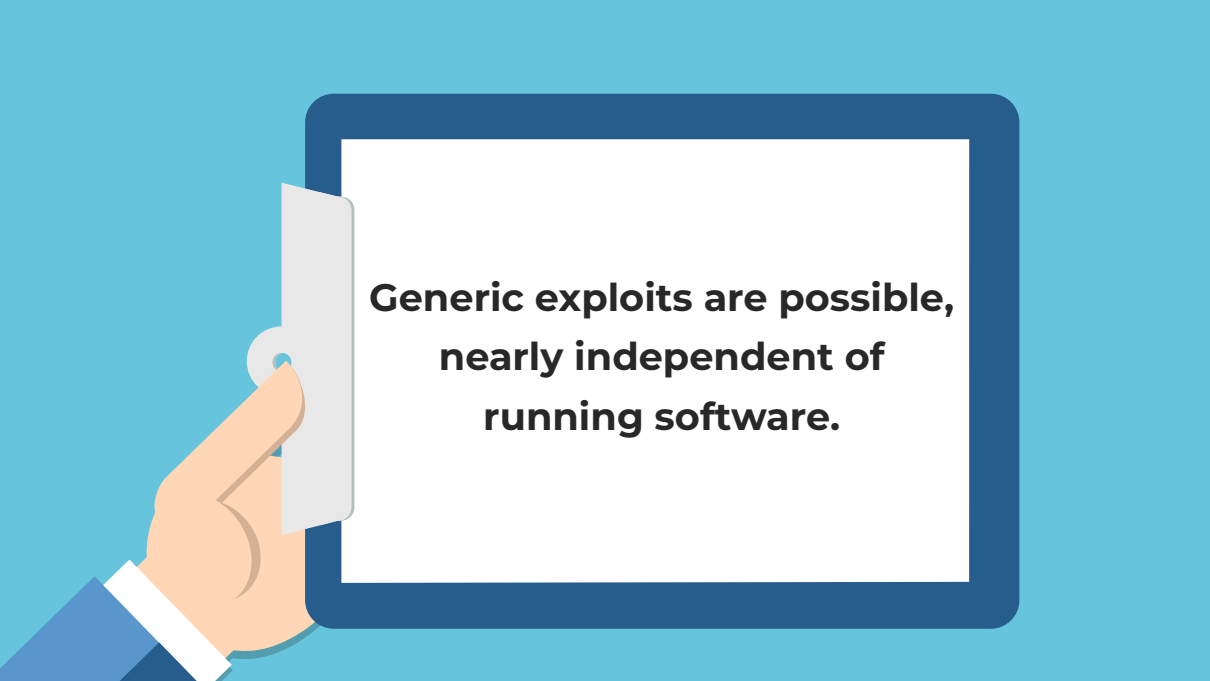
```
x = ret1()  
┌ if x == 0:  
true → WIN()  
  
x = ret0()  
    FAIL()
```

ret1()

return 1

ret0()

return 0

An illustration of a hand holding a clipboard against a light blue background. The hand is positioned on the left, holding a grey clipboard. The clipboard contains a white rectangular area with a dark blue border. Inside this area, the text "Generic exploits are possible, nearly independent of running software." is written in a bold, black, sans-serif font. The hand is wearing a blue sleeve with a white cuff.

**Generic exploits are possible,
nearly independent of
running software.**

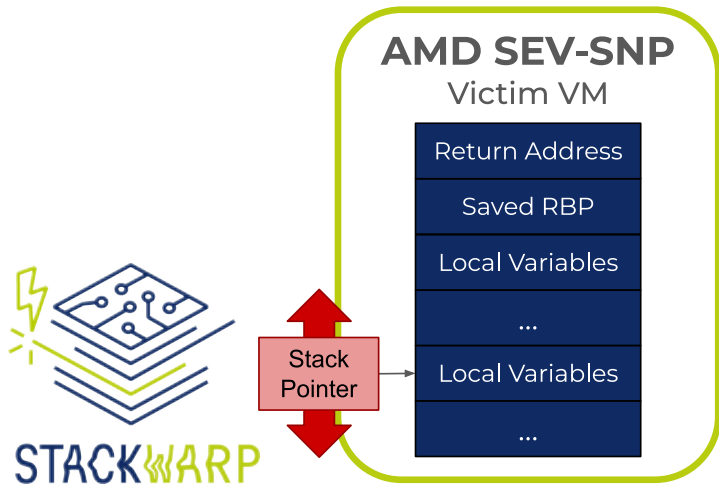


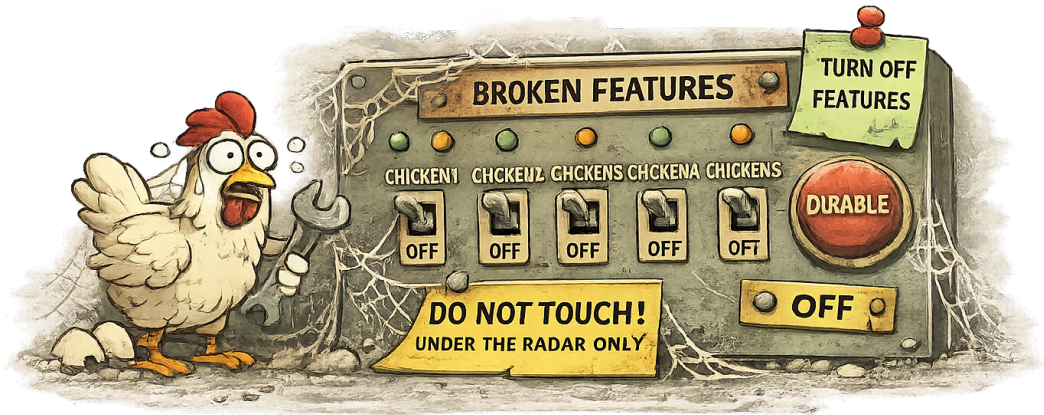
AMD: Not Great, Not Terrible

AMD Security Bulletin

AMD has provided a hot-loadable microcode patch and updated the firmware image [...] This issue has not been found to impact AMD 4th generation “Genoa” EPYC™ processors (“Zen 4” microarchitecture).

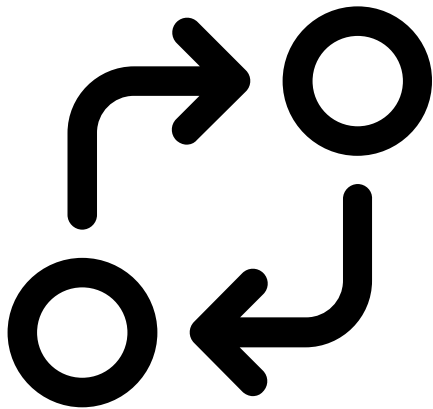
AMD INVD Instruction Security Vulnerability (AMD-SB-3005)







Feature Selection



Features can be **on** or **off**

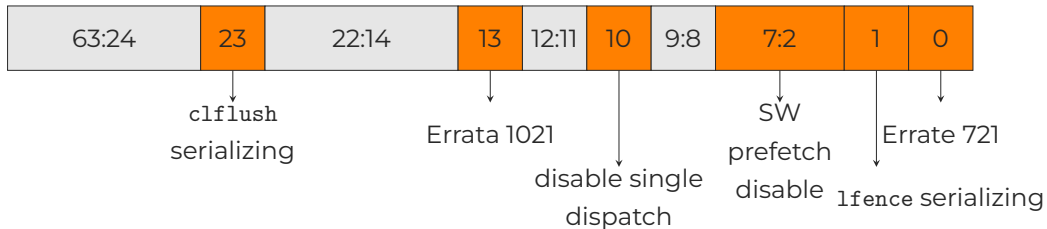


Cycle on/off → state known when used



First Observable Effect

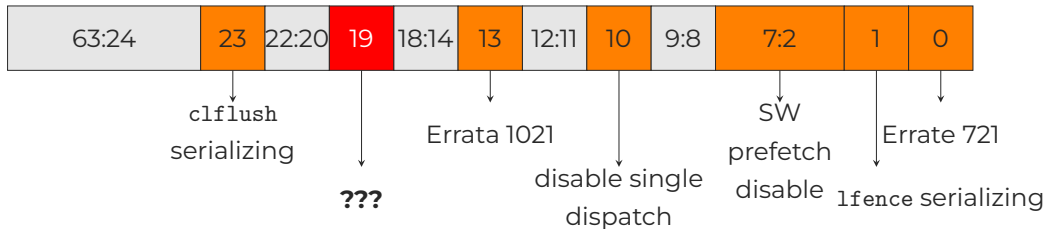
MSR 0xC0011029





First Observable Effect

MSR 0xC0011029





Observable Effect

```
Program received signal SIGILL, Illegal instruction.
```

```
0x00007ffff7d1ba88 in __GI___libc_read (fd=0, buf=0x7ffff7f9c000, nbytes=131072) at ../  
sysdeps/unix/sysv/linux/read.c:26
```

```
26      in ../sysdeps/unix/sysv/linux/read.c  
(gdb) bt
```

```
#0 0x00007ffff7d1ba88 in __GI___libc_read (fd=0, buf=0x7ffff7f9c000, nbytes=131072) at ../  
sysdeps/unix/sysv/linux/read.c:26
```

```
#1 0x00005555555557f66 in ?? ()
```

```
#2 0x00005555555557285 in ?? ()
```

```
#3 0x00007ffff7c2a1ca in __libc_start_call_main (main=main@entry=0x555555556730, argc=  
argc@entry=1, argv=argv@entry=0x7fffffff3d8) at ../sysdeps/nptl/libc_start_call_main.h  
:58
```

```
#4 0x00007ffff7c2a28b in __libc_start_main_impl (main=0x555555556730, argc=1, argv=0  
x7fffffff3d8, init=<optimized out>, fini=<optimized out>, rtd_fini=<optimized out>,  
stack_end=0x7fffffff3c8) at ../csu/libc-start.c:360
```

```
#5 0x00005555555557ae5 in ?? ()
```



Understanding the Crash

That strongly indicates control flow got redirected (misaligned jump/return)

Look on the stack for values near 0x7ffff7d1ba88. If you see that address sitting where a return address should be, you likely have a smashed stack / overwritten return address.





Performance Counter Evidence

Instruction class	Bit set
Implicit stack ops (push/pop, call/ret)	More μ ops
Explicit <code>rsp</code> updates	Fewer μ ops
Non-stack arithmetic	Unchanged



The Culprit: The Stack Engine

Program

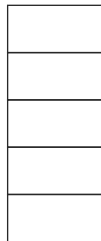
```
▶ push 1  
  push 2  
  push 3  
  mov %rax, %rsp
```

Stack Engine

$\Delta\text{RSP}: 0$

Victim Stack

RSP →



← μRSP



The Culprit: The Stack Engine

Program

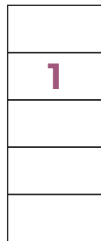
```
▶ push 1  
  push 2  
  push 3  
  mov %rax, %rsp
```

Stack Engine

$\Delta\text{RSP} : -8$

Victim Stack

RSP →



$\leftarrow \mu\text{RSP}$



The Culprit: The Stack Engine

Program

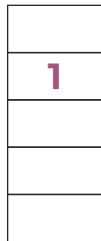
```
push 1  
▶ push 2  
push 3  
mov %rax, %rsp
```

Stack Engine

$\Delta\text{RSP} : -8$

Victim Stack

RSP →



← μRSP



The Culprit: The Stack Engine

Program

```
push 1  
▶ push 2  
push 3  
mov %rax, %rsp
```

Stack Engine

$\Delta\text{RSP}:-16$

Victim Stack

RSP →



← μRSP



The Culprit: The Stack Engine

Program

```
push 1  
push 2  
▶ push 3  
mov %rax, %rsp
```

Stack Engine

$\Delta\text{RSP}:-16$

Victim Stack

RSP →



← μRSP



The Culprit: The Stack Engine

Program

```
push 1  
push 2  
▶ push 3  
mov %rax, %rsp
```

Stack Engine

$\Delta RSP: -24$

Victim Stack

RSP →



← μRSP



The Culprit: The Stack Engine

Program

```
push 1  
push 2  
push 3  
▶ mov %rax, %rsp
```

Stack Engine

$\Delta\text{RSP}:-24$

Victim Stack

RSP →



← μRSP



The Culprit: The Stack Engine

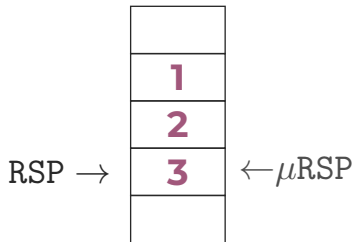
Program

```
push 1  
push 2  
push 3  
▶ mov %rax, %rsp
```

Stack Engine

$\Delta\text{RSP}: 0$

Victim Stack





StackWarp: Basic Primitive

Attacker VM

▶ `pop %rax`
[*SE off*]

[*SE on*]

Victim VM

`push 1`

`push 2`

Stack Engine

$\Delta RSP: 0$

Victim Stack

RSP →





StackWarp: Basic Primitive

Attacker VM

▶ `pop %rax`
[*SE off*]

[*SE on*]

Victim VM

`push 1`

`push 2`

Stack Engine

$\Delta RSP: 8$

Victim Stack

RSP →





StackWarp: Basic Primitive

Attacker VM

pop %rax

▶ [*SE off*]

[*SE on*]

Victim VM

push 1

push 2

Stack Engine

$\Delta RSP: 8$

Victim Stack

RSP →





StackWarp: Basic Primitive

Attacker VM

pop %rax

▶ [*SE off*]

[*SE on*]

Victim VM

push 1

push 2

Stack Engine

$\Delta RSP: 8$

Victim Stack

RSP →





StackWarp: Basic Primitive

Attacker VM

pop %rax

[*SE off*]

[*SE on*]

Victim VM

▶ push 1

push 2

Stack Engine

$\Delta RSP: 8$

Victim Stack

RSP →





StackWarp: Basic Primitive

Attacker VM

pop %rax

[*SE off*]

[*SE on*]

Victim VM

▶ push 1

push 2

Stack Engine

$\Delta RSP: 8$

Victim Stack

RSP →





StackWarp: Basic Primitive

Attacker VM

pop %rax

[*SE off*]

▶ [*SE on*]

Victim VM

push 1

push 2

Stack Engine

$\Delta RSP: 8$

Victim Stack

RSP →





StackWarp: Basic Primitive

Attacker VM

pop %rax

[*SE off*]

▶ [*SE on*]

Victim VM

push 1

push 2

Stack Engine

$\Delta RSP: 8$

Victim Stack

RSP →





StackWarp: Basic Primitive

Attacker VM

pop %rax

[*SE off*]

[*SE on*]

Victim VM

push 1

▶ push 2

Stack Engine

$\Delta RSP: 0$

Victim Stack

RSP →





StackWarp: Basic Primitive

Attacker VM

pop %rax

[*SE off*]

[*SE on*]

Victim VM

push 1

▶ push 2

Stack Engine

$\Delta RSP: 0$

Victim Stack

RSP →





Attack Targets



**Fault-attacks
on Crypto**



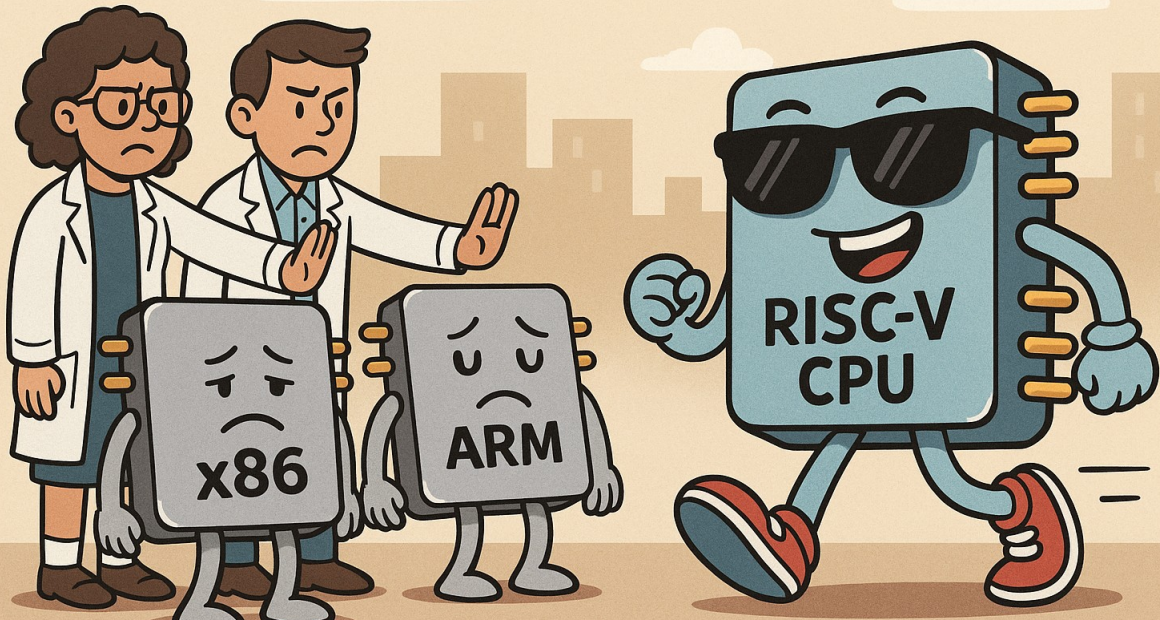
**Privilege
Escalation**



**Kernel Code
Execution**

**“Malicious hypervisors sound
more theoretical than
practical”**







Ghost Write

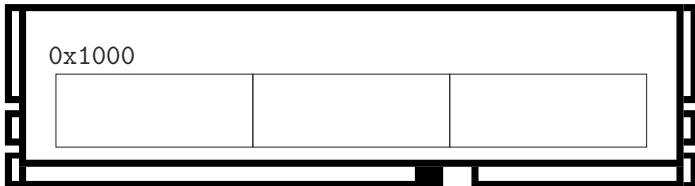


RISC-V Vector Instructions

vse128.v v0, 0(t0)



t0	v0	v1	v2
0x1000	RISC-V_is_a_super	r_cool_CPU_archi	ture!



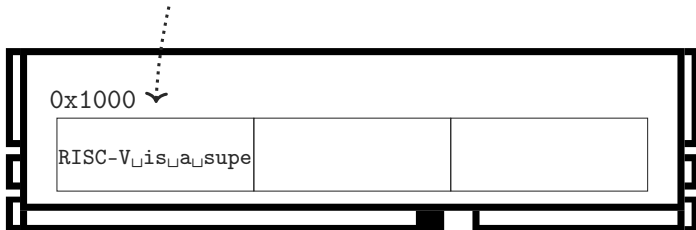


RISC-V Vector Instructions

```
vse128.v v0, 0(t0)
```



t0	v0	v1	v2
0x1000	RISC-V_is_a_super	cool_CPU_archi	ture!



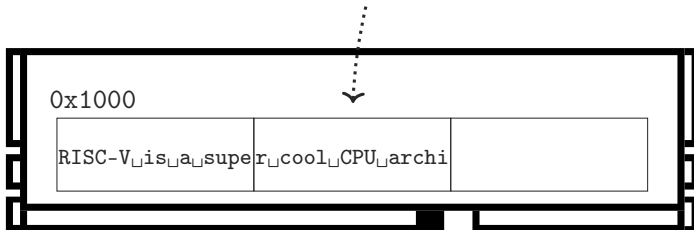


RISC-V Vector Instructions

```
vse128.v v0, 0(t0)
```



t0	v0	v1	v2
0x1000	RISC-V_is_a_super	r_cool_CPU_archi	ture!





RISC-V Vector Instructions

```
vse128.v v0, 0(t0)
```



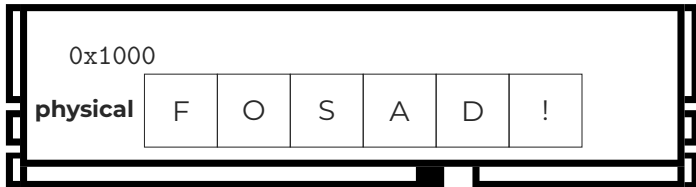
t0	v0	v1	v2
0x1000	RISC-V_is_a_super	r_cool_CPU_archi	ture!





RISC-V Flawed Vector Instruction (C910)

t0	v0	v1	v2	v3	v4	v5
0x1000	R	I	S	C	-	V



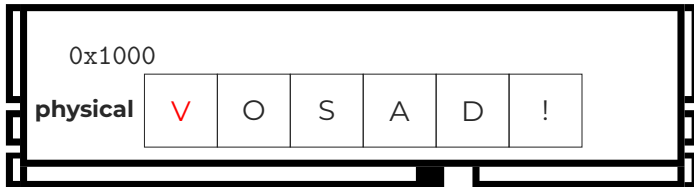


RISC-V Flawed Vector Instruction (C910)

vse128.v v0, 0(t0)



t0	v0	v1	v2	v3	v4	v5
0x1000	R	I	S	C	-	V





RISC-V Flawed Vector Instruction (C910)

vse128.v v0, 0(t0)



t0	v0	v1	v2	v3	v4	v5
0x1000	R	I	S	C	-	V



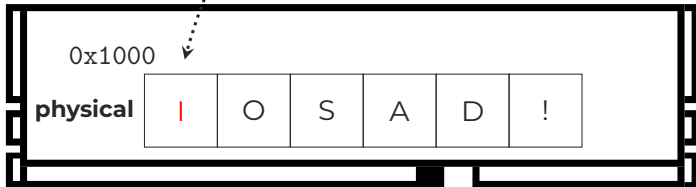


RISC-V Flawed Vector Instruction (C910)

vse128.v v0, 0(t0)



t0	v0	v1	v2	v3	v4	v5
0x1000	R	I	S	C	-	V



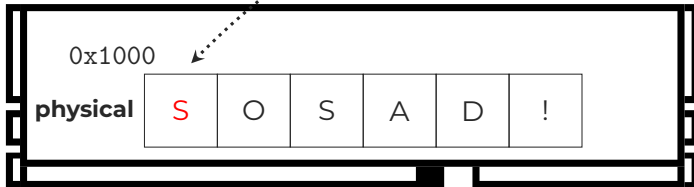


RISC-V Flawed Vector Instruction (C910)

vse128.v v0, 0(t0)



t0	v0	v1	v2	v3	v4	v5
0x1000	R	I	S	C	-	V



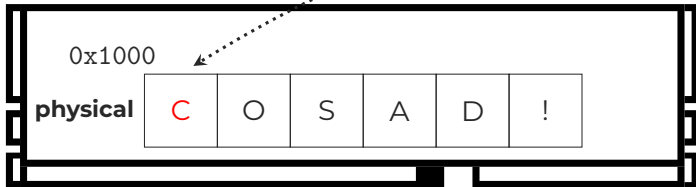


RISC-V Flawed Vector Instruction (C910)

vse128.v v0, 0(t0)



t0	v0	v1	v2	v3	v4	v5
0x1000	R	I	S	C	-	V



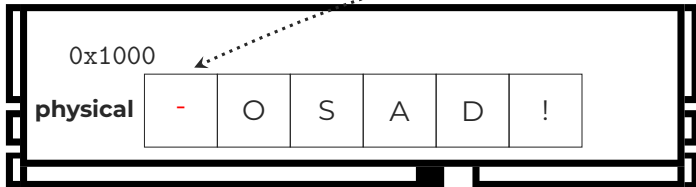


RISC-V Flawed Vector Instruction (C910)

vse128.v v0, 0(t0)



t0	v0	v1	v2	v3	v4	v5
0x1000	R	I	S	C	-	V



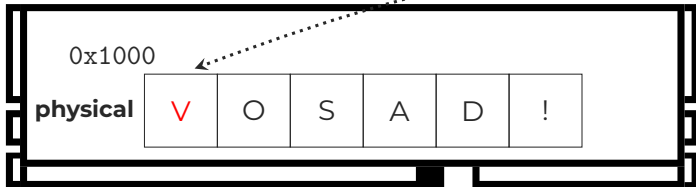


RISC-V Flawed Vector Instruction (C910)

vse128.v v0, 0(t0)



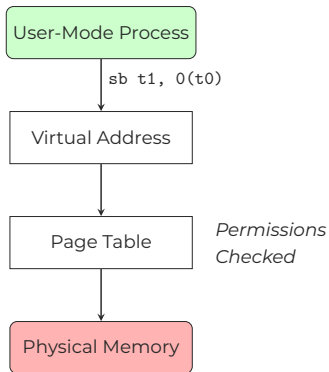
t0	v0	v1	v2	v3	v4	v5
0x1000	R	I	S	C	-	V





GhostWrite: Circumventing Virtual Memory

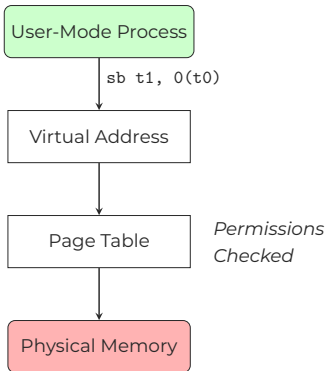
Normal Memory Write



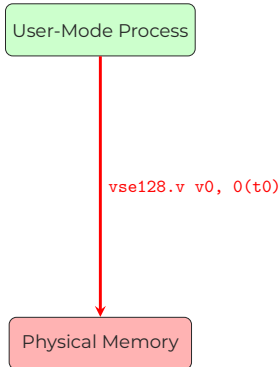


GhostWrite: Circumventing Virtual Memory

Normal Memory Write



GhostWrite





GhostWrite

An architectural bug in a RISC-V vector implementation allowed writes to **physical** memory.

Result: Write-anything-anywhere primitive

Architectural bugs are not an x86-only problem.

**“But nobody uses RISC-V in
real deployments”**



Elastic Metal

RV1

The world's first RISC-V servers available in the cloud.

Taste the new open processor architecture now.

Will you take the risk?

[Order Now →](#)

[Read the press release →](#)





```
Welcome to Ubuntu 23.10 (GNU/Linux 5.10.113+ riscv64)
```

```
0 updates can be applied immediately.
```

```
ubuntu@risc:~$ sudo ./a.out
Virtual address:      3feab9f000
Physical address:     3efc9b000
```

```
ubuntu@risc:~$
```



intel

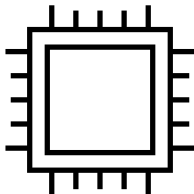
AMD



LOONGSON



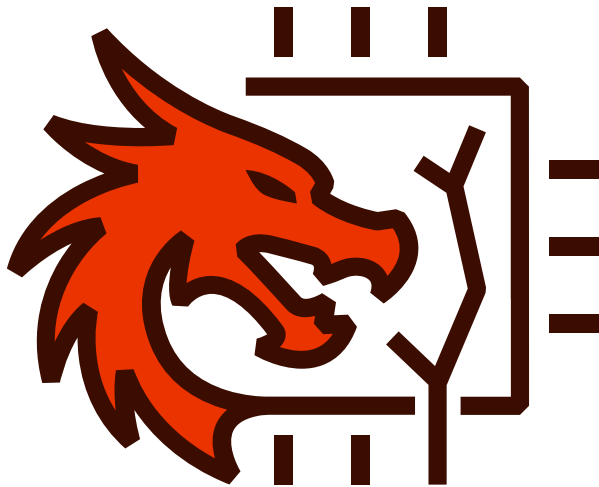
Chinese domestic
CPUs produced by
Loongson



RISC architecture
supported by Linux,
GCC, QEMU



Mainly in
government and
enterprise





Vectors are Difficult



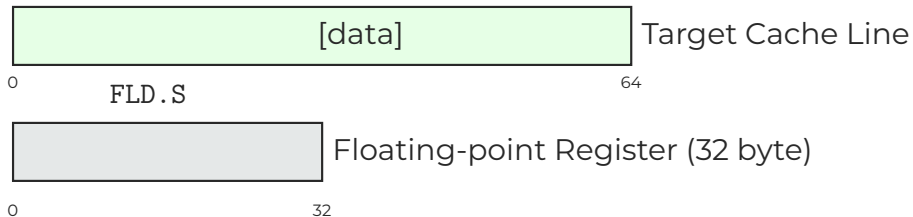
Floating-point Register (32 byte)

0

32

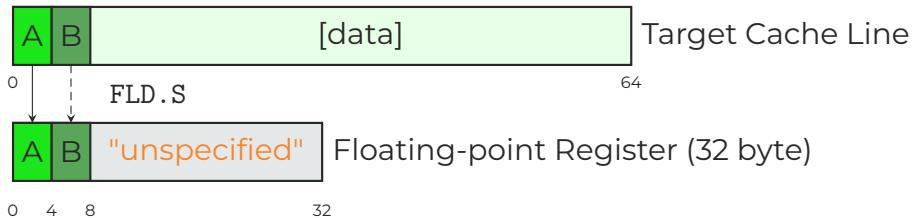


Vectors are Difficult



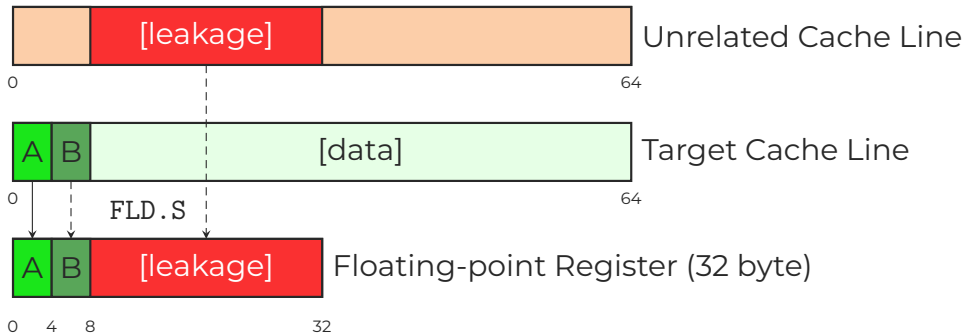


Vectors are Difficult





Vectors are Difficult





Vectors are Difficult



Floating-point Register (32 byte)

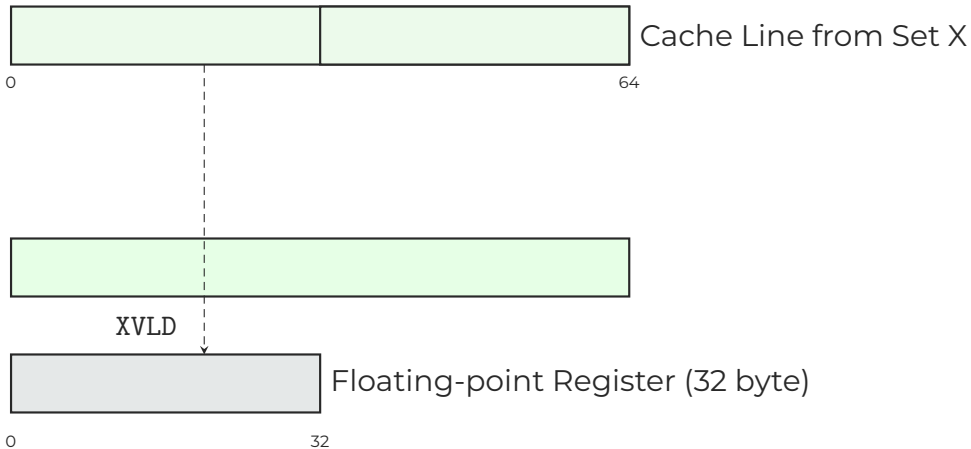
0

32

30

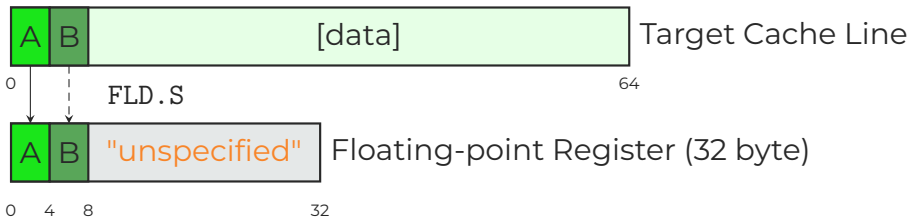
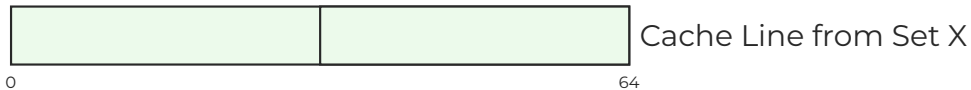


Vectors are Difficult



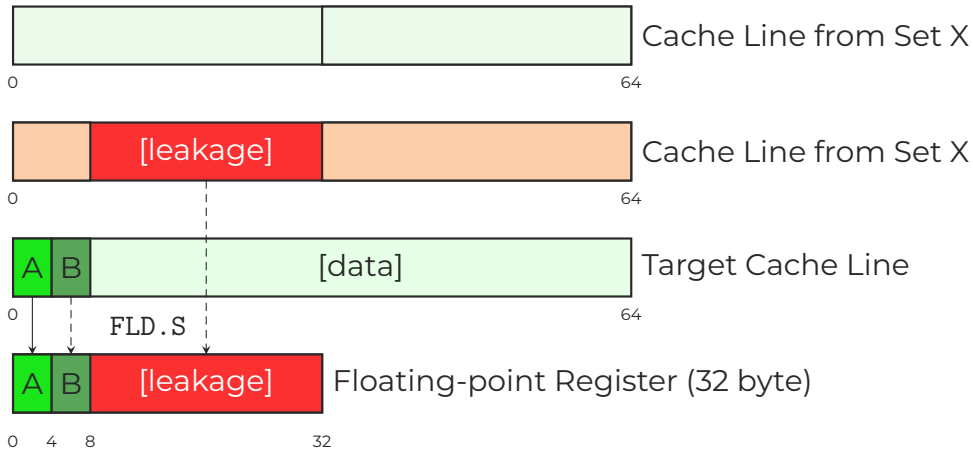


Vectors are Difficult





Vectors are Difficult





LoongLeak Consequence

Cache location

set s , way w

L1 Cache Line



Readout

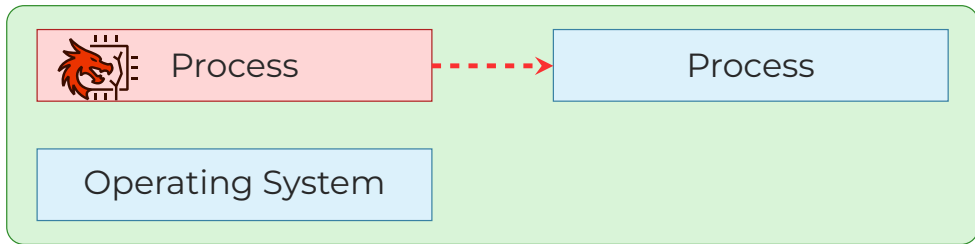
→ **300 MB/s**
non-destructive

Any data in L1 becomes readable

Same logical core or sibling logical core

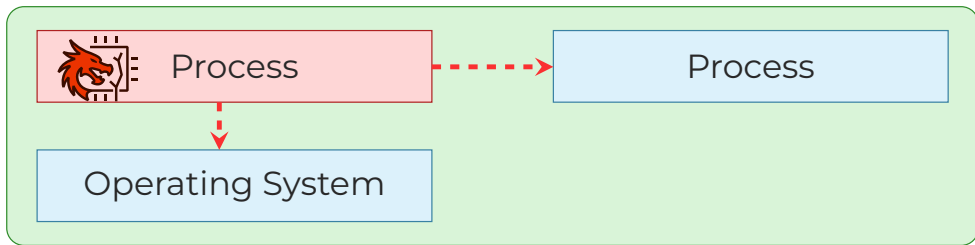


Threat Model



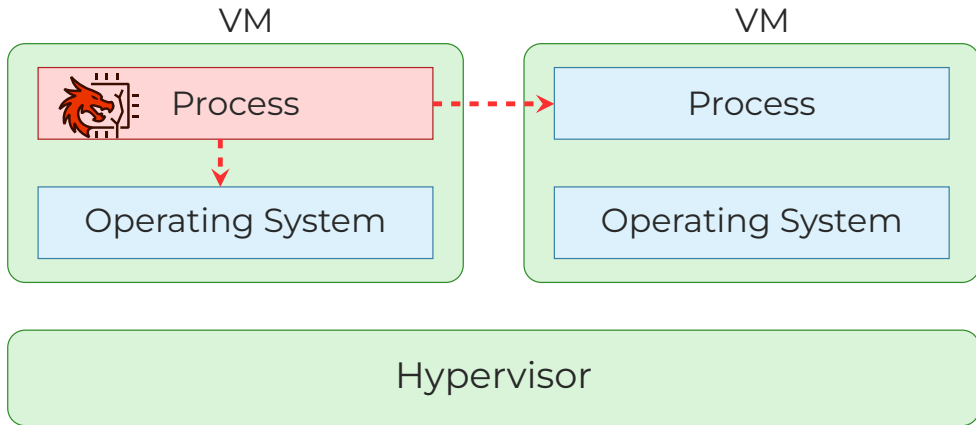


Threat Model



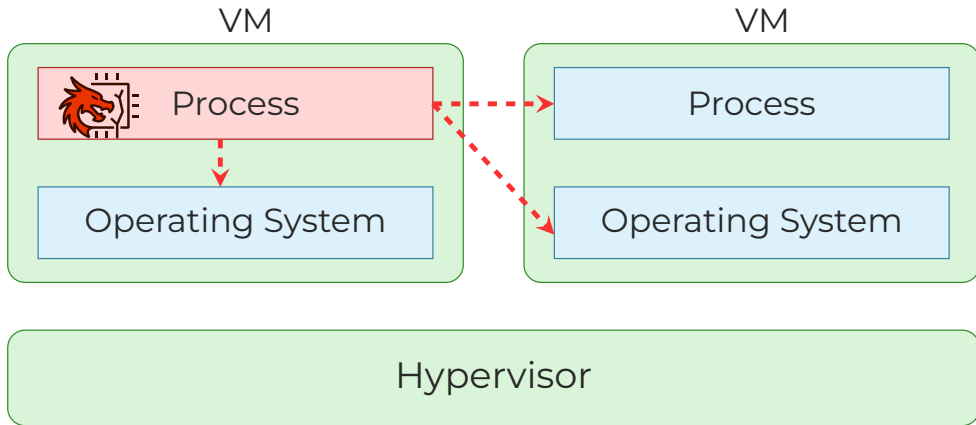


Threat Model



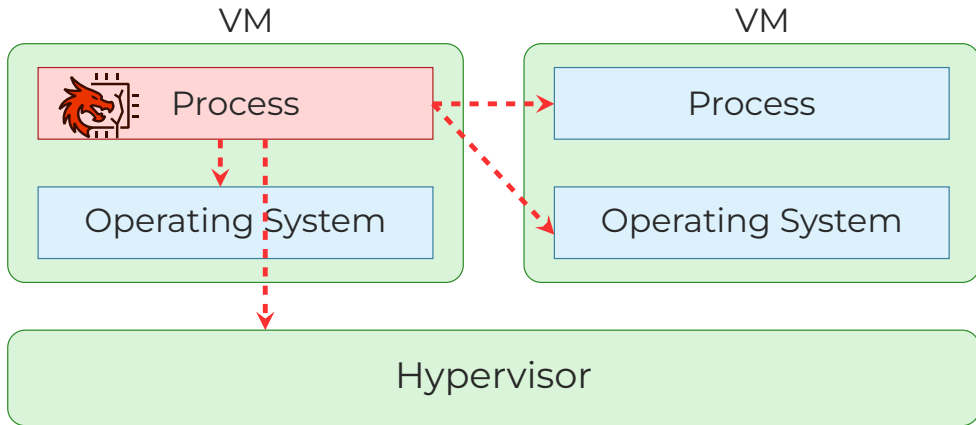


Threat Model





Threat Model

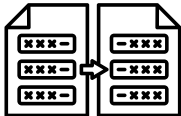




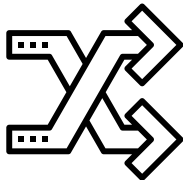
Attack Targets



**LUKS Disk
Encryption**



**Root Password
Hash**



**(K)ASLR
Break**

“Is it really that bad?”





State of CPU Vulnerabilities

	Vulnerability	Conf.	Integ.	Minimum attacker	ISA
	StackWarp		✓	Hypervisor	x86
	CacheWarp		✓	Hypervisor	x86
	Fabricked	✓	✓	Hypervisor + UEFI	x86
	Staleus	✓	✓	Hypervisor	x86
	ÆPIC Leak	✓		Hypervisor / OS	x86
	Zenbleed	✓		user space	x86
	GhostWrite	✓	✓	user space	RISC-V
	LoongLeak	✓		user space	LoongArch



Pandora's Box



Security vulnerabilities in CPUs are **real**



Pandora's Box



Side channels often used to see them



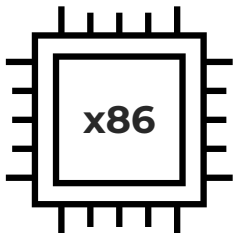
Pandora's Box



Architectural bugs keep showing up

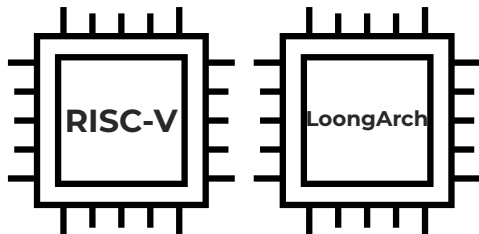


Mitigations



Firmware

Microcode/firmware updates



None

Requires new hardware,
some OS workarounds



Expectations for CPU Vulnerabilities

EXPLOIT-O-METER



Exploits now?



Expectations for CPU Vulnerabilities

EXPLOIT-O-METER



Exploits in a few years?



Expectations for CPU Vulnerabilities

EXPLOIT-O-METER



Exploits if someone builds automated tooling?



Takeaways



CPUs must be treated like **software**

just faster, harder to debug, and deployed at massive scale



CPU bugs are exploitable

across VMs, enclaves, and privilege boundaries



Testing still in its infancy

needs observability, metrics, strategies