Repurposing Segmentation as a Practical LVI-NULL Mitigation in SGX

Lukas Giner, Andreas Kogler, Claudio Canella, Michael Schwarz, Daniel Gruss

May 06, 2022

Who am I?

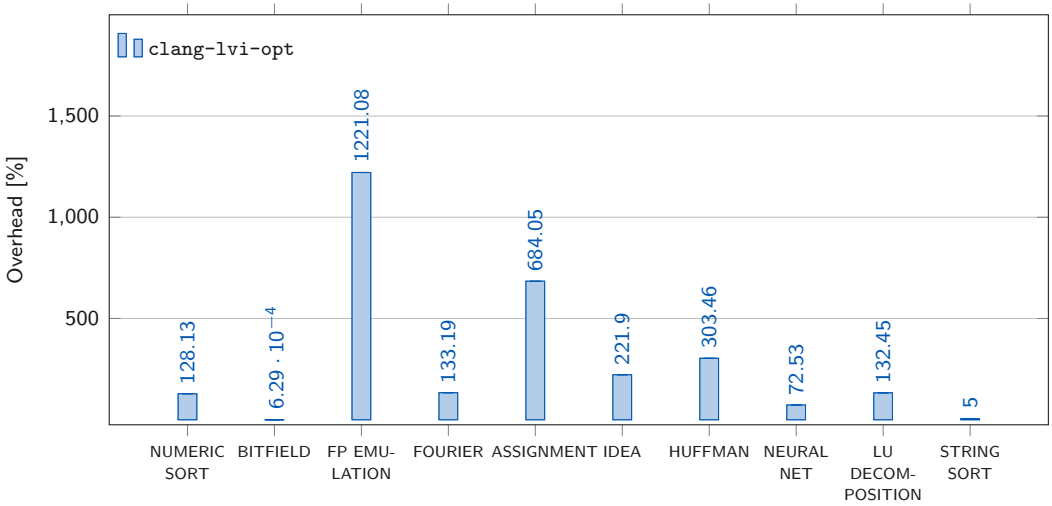
Lukas Giner

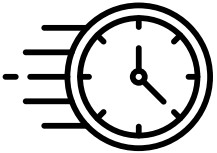
PhD student @ Graz University of Technology

@redrabbyte

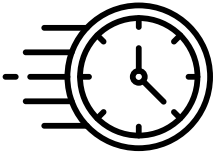
lukas.giner@iaik.tugraz.at

Motivation

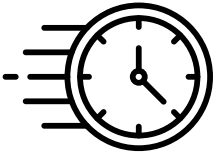

clang-lvi-opt
Overhead [%]
1,500
1,000
500
128.13
$6.29 \cdot 10^{-4}$
1221.08
133.19
684.05
221.9
303.46
72.53
132.45
5
NUMERIC SORT
BITFIELD
FP EMU-LATION
FOURIER
ASSIGNMENT
IDEA
HUFFMAN
NEURAL NET
LU DECOM-POSITION
STRING SORT



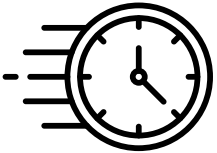
- CPU does many things, all at once



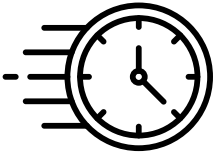
- CPU does many things, all at once
- Parallelizing instructions speeds up execution



- CPU does many things, all at once
- Parallelizing instructions speeds up execution
- Mistakes?



- CPU does many things, all at once
- Parallelizing instructions speeds up execution
- Mistakes?
- Roll back to the mistake, either raise an error or try again



- CPU does many things, all at once
- Parallelizing instructions speeds up execution
- Mistakes?
- Roll back to the mistake, either raise an error or try again
- Undone instructions are called transient

- What can an attacker do with transient instructions?

- What can an attacker do with transient instructions?
- Not all state is gone - traces in μArch state

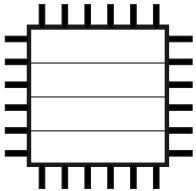
- What can an attacker do with transient instructions?
- Not all state is gone - traces in μArch state
- Caching!

Meltdown: Transiently encoding unauthorized memory

User Memory

	𝔸	𝔹
ℂ	𝔻	𝔼
𝔽	𝔾	ℍ
𝕀	𝕁	𝕂
𝕃	𝕄	ℕ
𝕆	ℙ	ℚ
ℝ	𝕊	𝕋
𝕌	𝕍	𝕎
𝕏	𝕐	ℤ

```
char value = kernel[0]
```



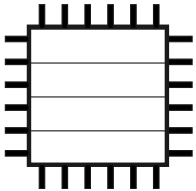
Meltdown: Transiently encoding unauthorized memory

User Memory

	𝔸	𝔹
ℂ	𝔻	𝔼
𝔽	𝔾	ℍ
𝕀	𝕁	𝕂
𝕃	𝕄	ℕ
𝕆	ℙ	ℚ
ℝ	𝕊	𝕋
𝕌	𝕍	𝕎
𝕏	𝕐	ℤ

```
char value = kernel[0]
```

Page fault (Exception)



Meltdown: Transiently encoding unauthorized memory

User Memory

	𝔸	𝔹
ℂ	𝔻	𝔼
𝔽	𝔾	ℍ
𝕀	𝕁	𝕂
𝕃	𝕄	ℕ
𝕆	ℙ	ℚ
ℝ	𝕊	𝕋
𝕌	𝕍	𝕎
𝕏	𝕐	ℤ

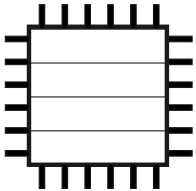
```
char value = kernel[0]
```

Page fault (Exception)

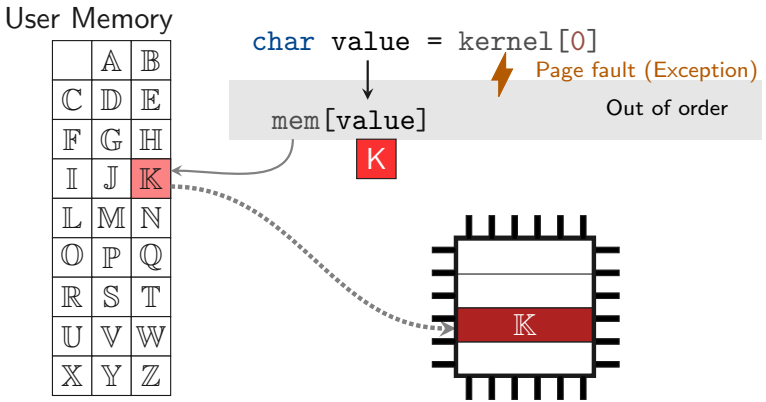
Out of order

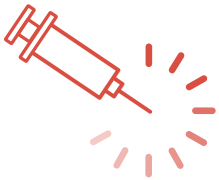
```
mem[value]
```

K

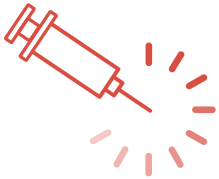


Meltdown: Transiently encoding unauthorized memory

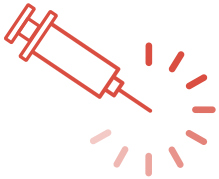




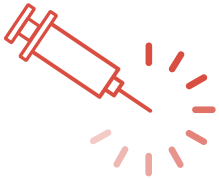
- What if we turn Meltdown around?



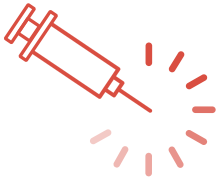
- What if we turn Meltdown around?
- uʍopʇlǝW



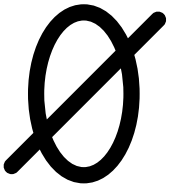
- What if we turn Meltdown around?
- Instead of leaking, we insert a value



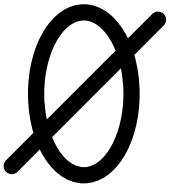
- What if we turn Meltdown around?
- Instead of leaking, we insert a value
- Transiently steer the victim to give up data!



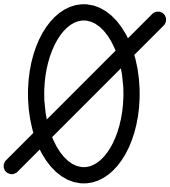
- What if we turn Meltdown around?
- Instead of leaking, we insert a value
- Transiently steer the victim to give up data!
- Exfiltrate with cache, as usual



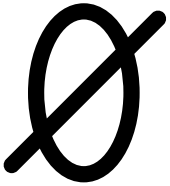
- How do these values get there?



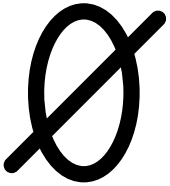
- How do these values get there?
- No need to care, Intel fixed it!



- How do these values get there?
- No need to care, Intel fixed it!
  (it's buffers and loose address matching)

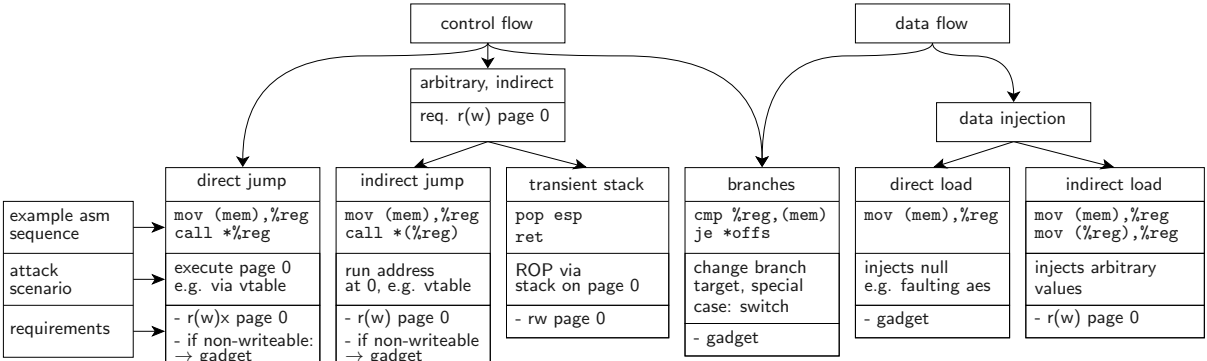


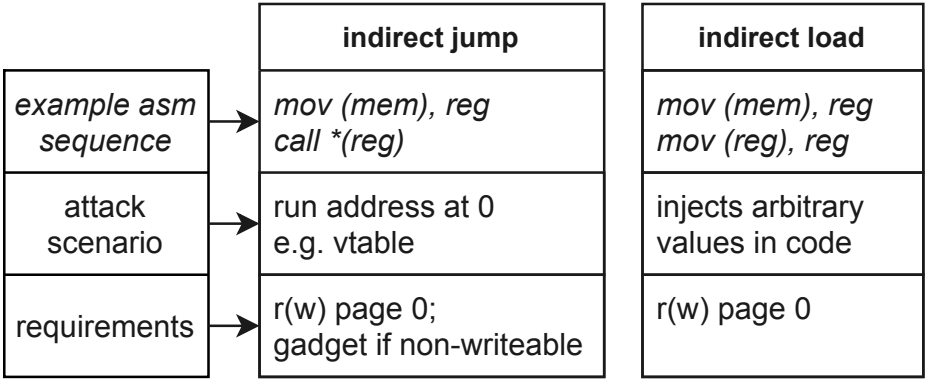
- How do these values get there?
- No need to care, Intel fixed it!
  (it's buffers and loose address matching)
- New CPUs just "inject" 0 instead

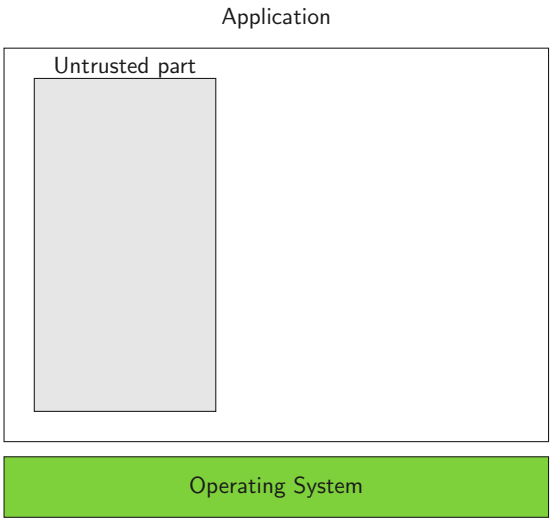


- How do these values get there?
- No need to care, Intel fixed it!
  (it's buffers and loose address matching)
- New CPUs just "inject" 0 instead
- Problem solved?

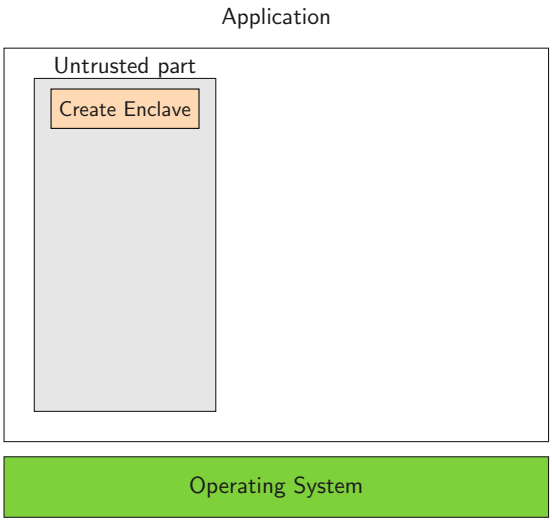
LVI-NULL Vectors



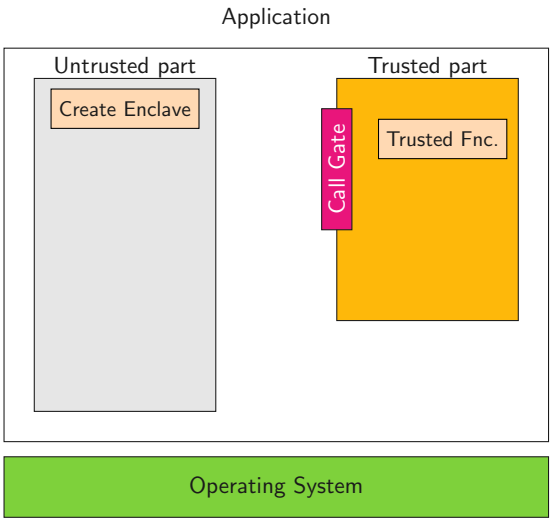

indirect jump
indirect load
example asm sequence
mov (mem), reg
call *(reg)
mov (mem), reg
mov (reg), reg
attack scenario
run address at 0
e.g. vtable
injects arbitrary values in code
requirements
r(w) page 0;
gadget if non-writeable
r(w) page 0



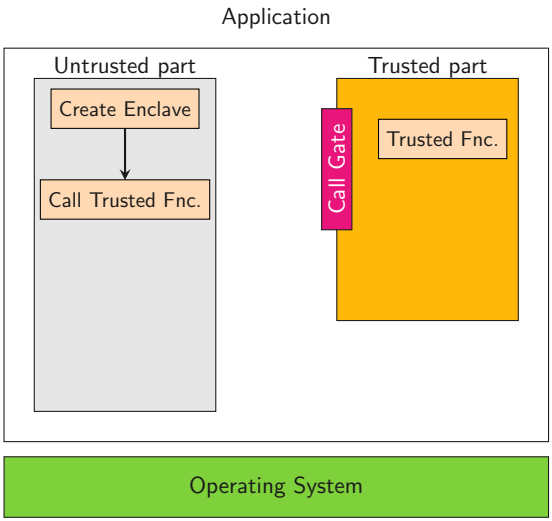
- SGX protects enclave from OS



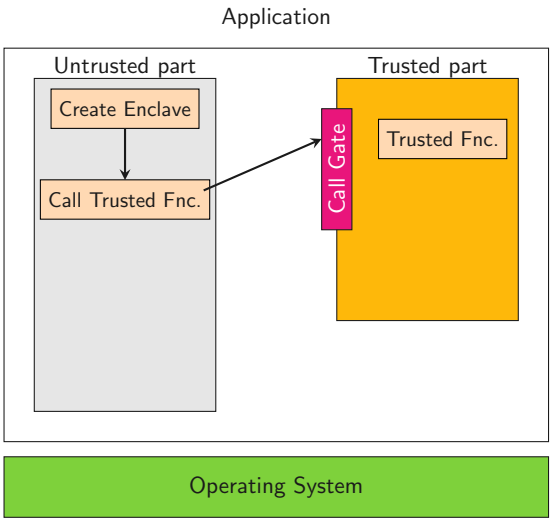
- SGX protects enclave from OS



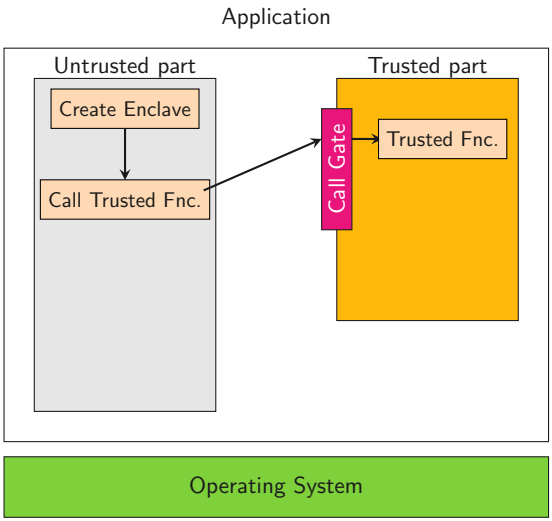
- SGX protects enclave from OS



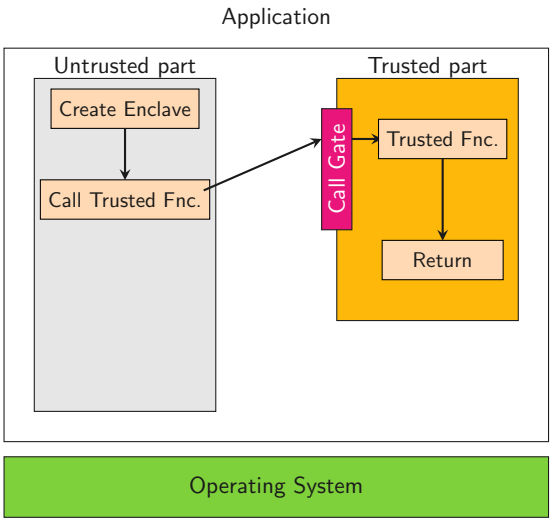
- SGX protects enclave from OS



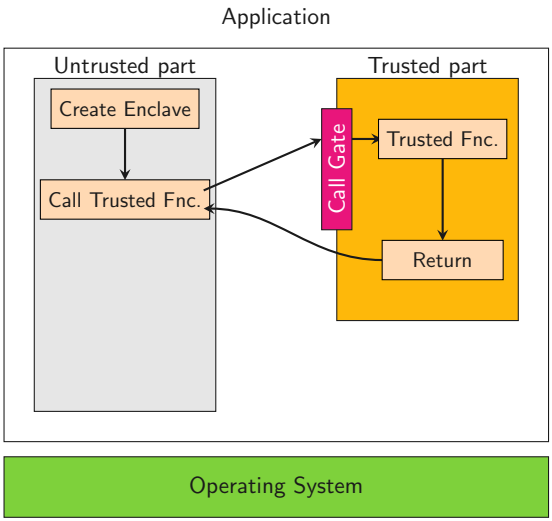
- SGX protects enclave from OS



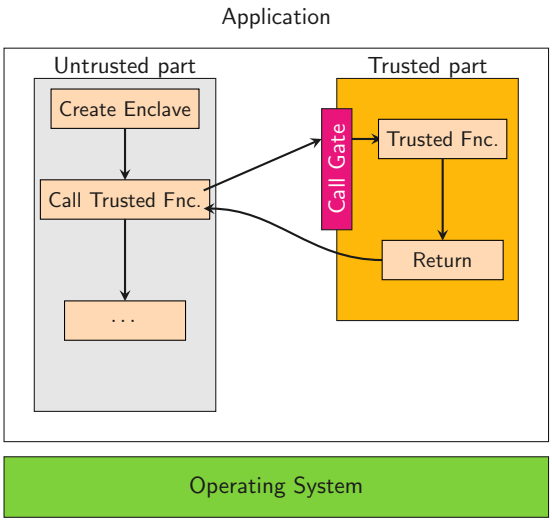
- SGX protects enclave from OS



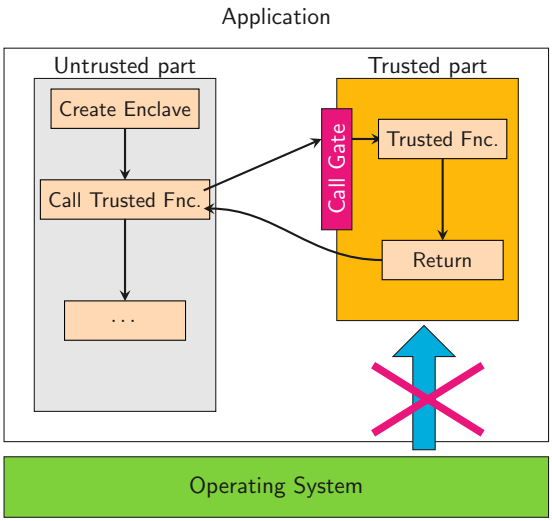
- SGX protects enclave from OS



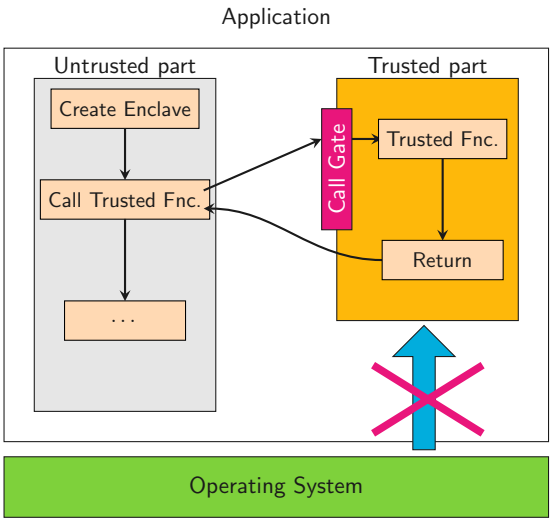
- SGX protects enclave from OS



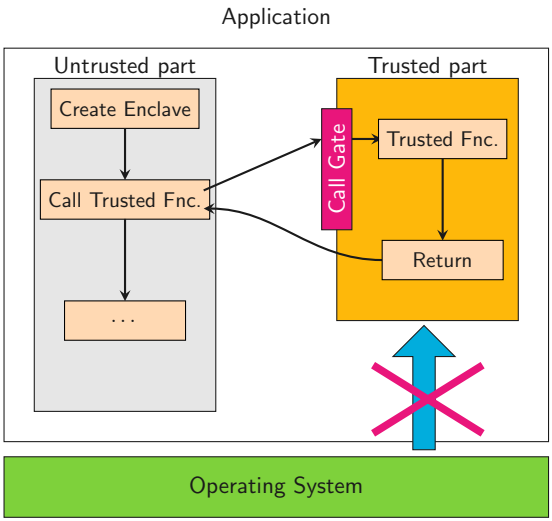
- SGX protects enclave from OS



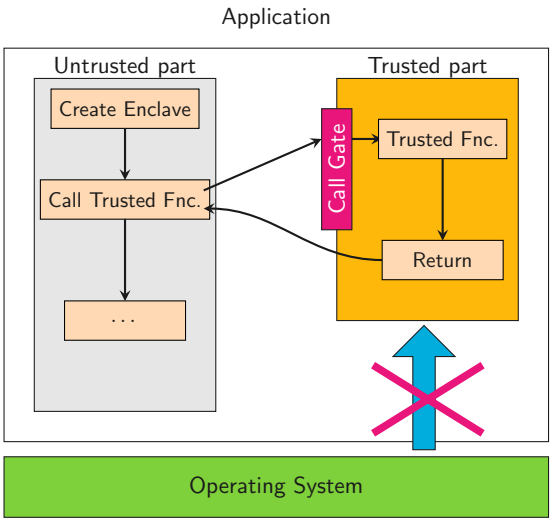
- SGX protects enclave from OS



- SGX protects enclave from OS
- But OS still controls parts of the PTE



- SGX protects enclave from OS
- But OS still controls parts of the PTE
- Removing accessed bit injects faults



- SGX protects enclave from OS
- But OS still controls parts of the PTE
- Removing accessed bit injects faults
- Instruction-targeted injection with SGX-Step

- Problem: Injecting 0 allows attacker controlled loads

- Problem: Injecting 0 allows attacker controlled loads
- Can we prevent redirection to zero page?

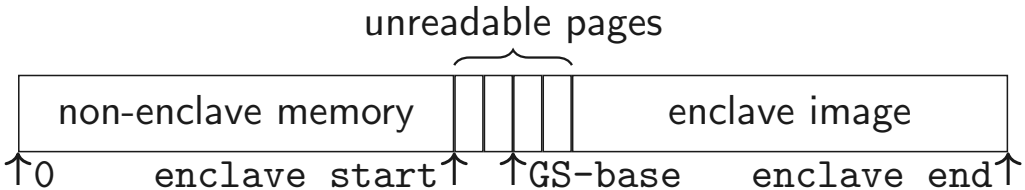
Mitigating in Software

- Problem: Injecting 0 allows attacker controlled loads
- Can we prevent redirection to zero page?
- Make all loads relative to enclave memory!

Mitigating in Software

- Problem: Injecting 0 allows attacker controlled loads
- Can we prevent redirection to zero page?
- Make all loads relative to enclave memory!
- Revive an ancient mechanism, Segmentation!

- Problem: Injecting 0 allows attacker controlled loads
- Can we prevent redirection to zero page?
- Make all loads relative to enclave memory!
- Revive an ancient mechanism, Segmentation!



Mitigated Code

```
push %rbp
```

```
sub  $0x8,%rsp
mov  %rbp,%gs:(%rsp)
```

- All loads over gs

Mitigated Code

```
push %rbp
```

```
sub  $0x8,%rsp
mov  %rbp,%gs:(%rsp)
```

- All loads over gs
- Instruction with implicit load

Mitigated Code

```
push %rbp
```

```
sub  $0x8,%rsp
mov  %rbp,%gs:(%rsp)
```

- All loads over gs
- Instruction with implicit load→
  replace with other instructions

Mitigated Code

```
push %rbp
```

```
callq 400480 <func>
```

```
sub  $0x8,%rsp
mov  %rbp,%gs:(%rsp)
lea  $return_address(%rip),%r11
sub  $0x8,%rsp
mov  %r11,%gs:(%rsp)
jmpq 400480 <func>
```

- All loads over gs
- Instruction with implicit load→
  replace with other instructions

Mitigated Code

```
push %rbp
```

```
callq 400480 <func>
```

```
pop  %rbp
```

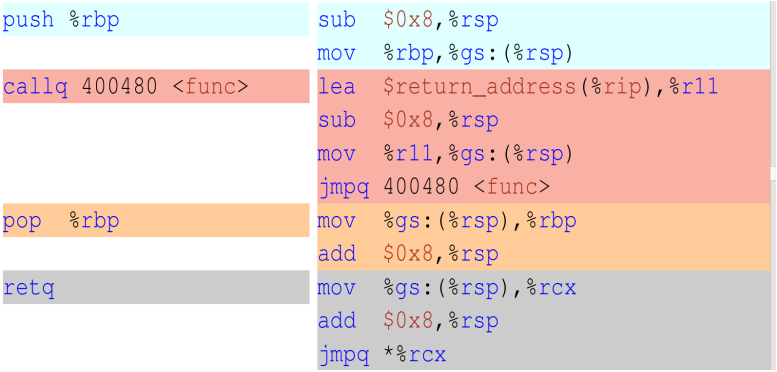
```
sub  $0x8,%rsp
mov  %rbp,%gs:(%rsp)
```

```
lea  $return_address(%rip),%r11
sub  $0x8,%rsp
mov  %r11,%gs:(%rsp)
jmpq 400480 <func>
```

```
mov  %gs:(%rsp),%rbp
add  $0x8,%rsp
```

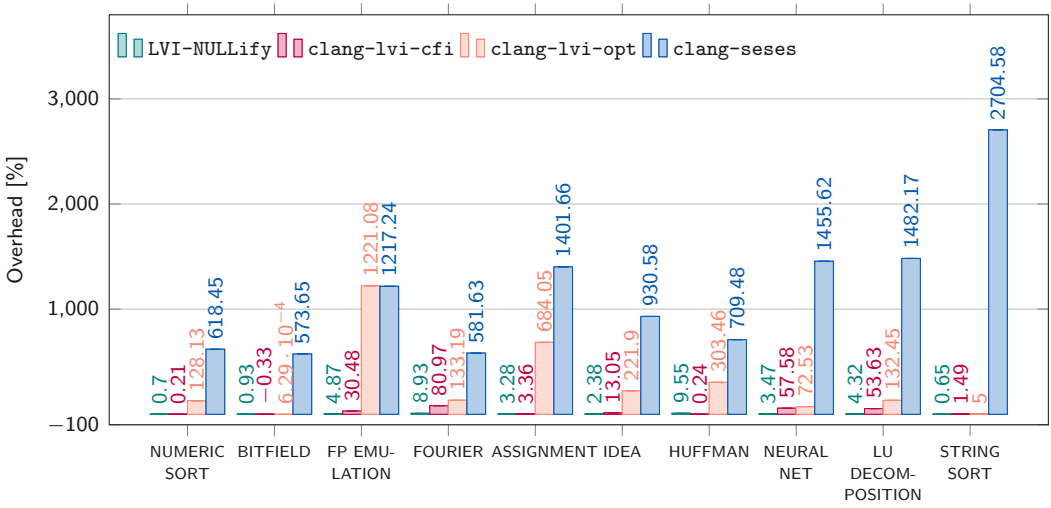
- All loads over gs
- Instruction with implicit load→
  replace with other instructions

Mitigated Code

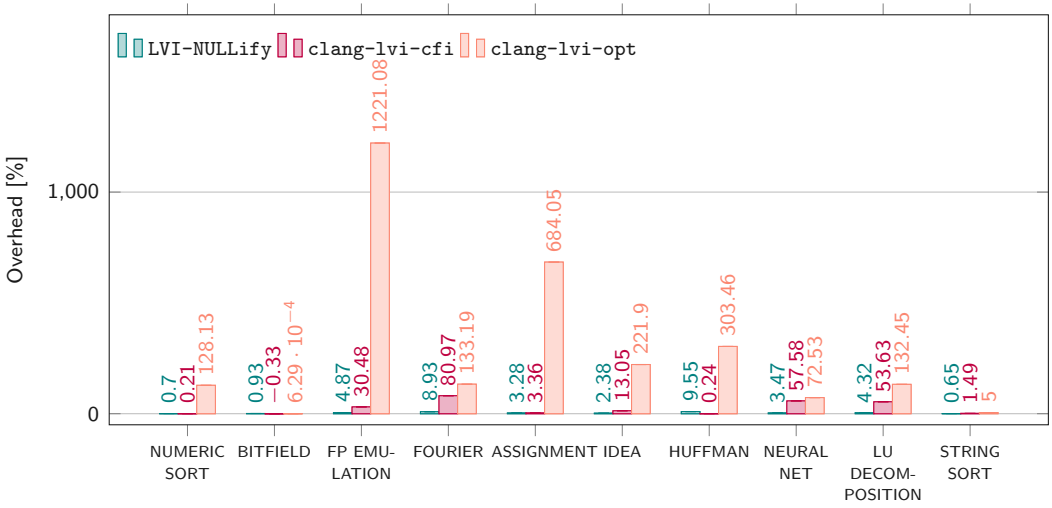


- All loads over gs
- Instruction with implicit load→ replace with other instructions

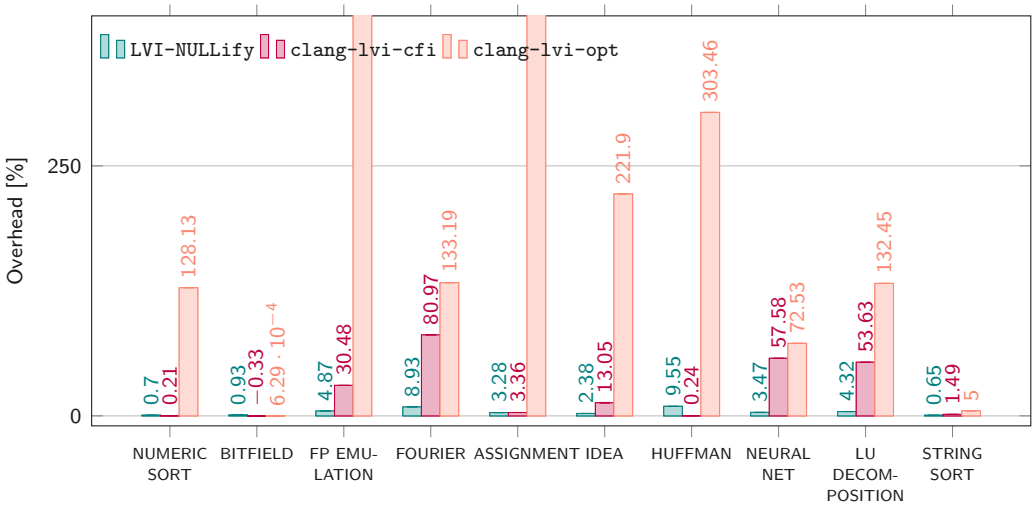
Performance



Performance



Performance



Conclusion

LVI-NULLify on GitHub

- LVI-NULL is still here (in Rocket Lake)..

Conclusion

LVI-NULLify on GitHub

- LVI-NULL is still here (in Rocket Lake)..
  ..but SGX isn't, in affected models

Conclusion

LVI-NULLify on GitHub

- LVI-NULL is still here (in Rocket Lake)..
  ..but SGX isn't, in affected models
- For now, only Comet Lake is affected

Conclusion

LVI-NULLify on GitHub

- LVI-NULL is still here (in Rocket Lake)..
  ..but SGX isn't, in affected models
- For now, only Comet Lake is affected
- Maybe relative addressing will be useful somewhere else?

Repurposing Segmentation as a Practical LVI-NULL Mitigation in SGX

Lukas Giner (@redrabbyte), Andreas Kogler (@0xhilbert), Claudio Canella (@cc0x1f)
Michael Schwarz (@misc0110), Daniel Gruss (@lavados)