SERBERUS: Protecting Cryptographic Code from Spectres at Compile-Time

Nicholas Mosier,[1] Hamed Nemati,[2] John Mitchell,[1] Caroline Trippel[1]

May 22, 2024
45th IEEE Symposium on Security and Privacy

[1] Stanford University

[2] KTH VETENSKAP OCH KONST

Security-critical code, like crypto code, is written to avoid leaking **secrets** sequentially

```
void
crypto_core_salsa(u8 *out,u8 *in, u8 *key,
                  u8 *c, int rounds) {
    // ...
    j1 = x1 = LOAD32_LE(key + 0);
    j2 = x2 = LOAD32_LE(key + 4);
    // ...
    for (i = 0; i < rounds; i += 2) {
      x4 ^= ROTL32(x0  + x12, 7);
      x8 ^= ROTL32(x4  + x0, 9);
     // ...
    }
    // ...
}
```

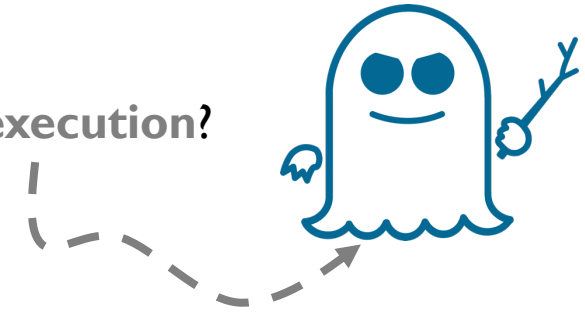
libsodium

Constant-time (CT) programming defense: Avoid passing **secrets** to the **unsafe operands** of **transmitters**

control-flow	memory access	variable-time ops
`if (unsafe)`	`y = A[unsafe]`	`z = unsafe/unsafe`

… in every **sequential execution** of the program.

What about **transient execution**?



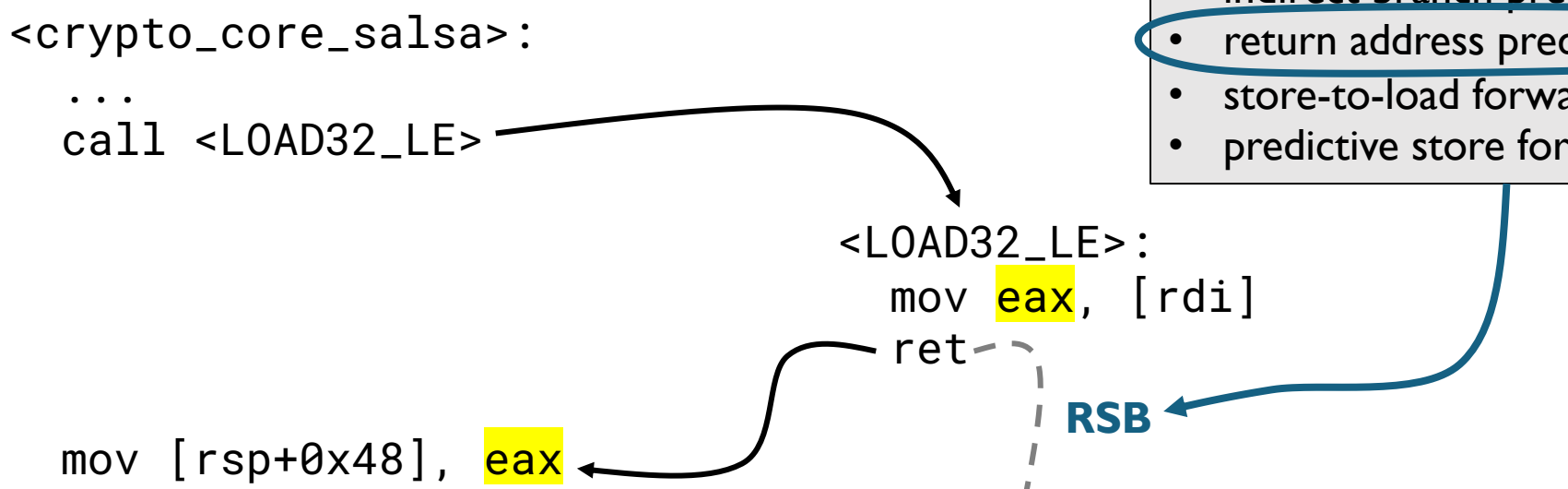
```
void
crypto_core_salsa(u8 *out,u8 *in, u8 *key,
                  u8 *c, int rounds) {
    // ...
    j1 = x1 = LOAD32_LE(key + 0);
    j2 = x2 = LOAD32_LE(key + 4);
    // ...
    for (i = 0; i < rounds; i += 2) {
      x4 ^= ROTL32(x0  + x12, 7);
      x8 ^= ROTL32(x4  + x0, 9);
     // ...
    }
    // ...
}
```

libsodium

```
<LOAD32_LE>:
  mov eax, [rdi]
  ret
```

Spectre Attacks on Constant-Time Crypto Code

Speculation primitives introduce mispredictions:
- conditional branch prediction (**PHT**)
- indirect branch prediction (**BTB**)
- return address prediction (**RSB**)
- store-to-load forwarding (**STL**)
- predictive store forwarding (**PSF**)



transient transmitter
(does not architecturally commit)

Spectre Attack!
secret key transiently leaked!

SERBERUS: first defense against Spectre attacks, involving **any combination of these speculation primitives**, that is readily deployable on **existing hardware**.

Comprehensive Spectre defenses face multiple challenges

Hardware speculation:

1. **Unconstrained control-flow**
4. Unconstrained data-flow (see paper)

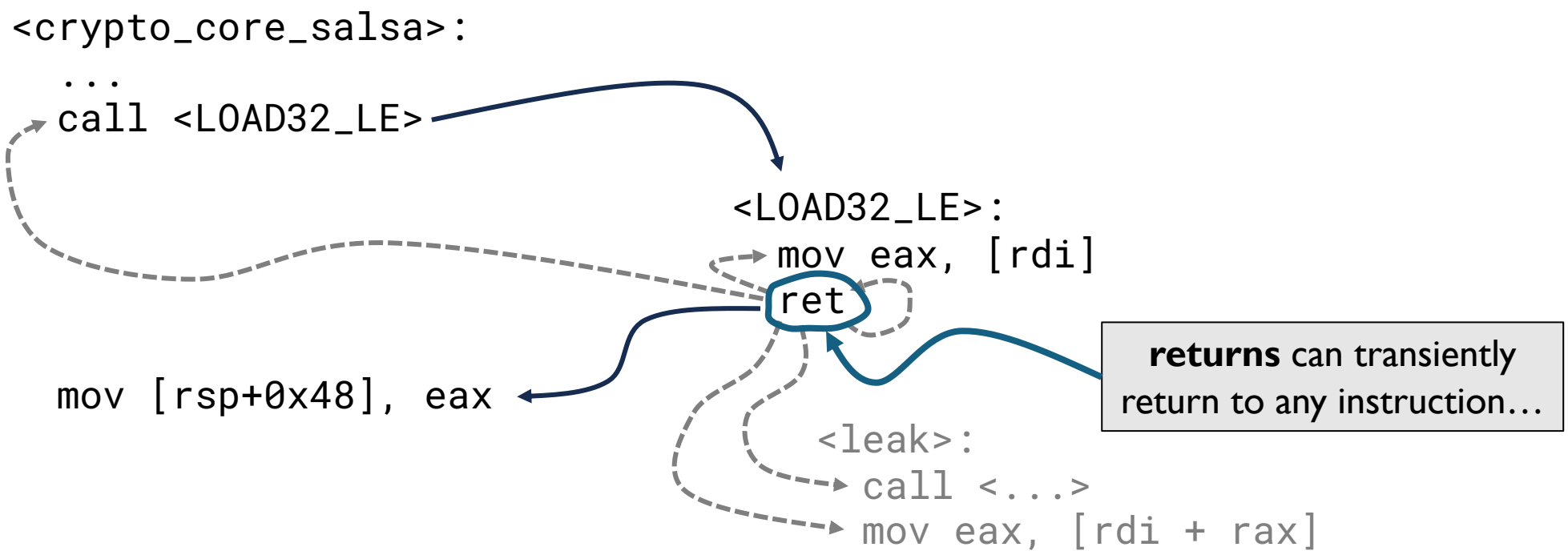
Legal constant-time code patterns:

5. No explicit requirement of static security types (see paper)
2. **Passing secret arguments by value**

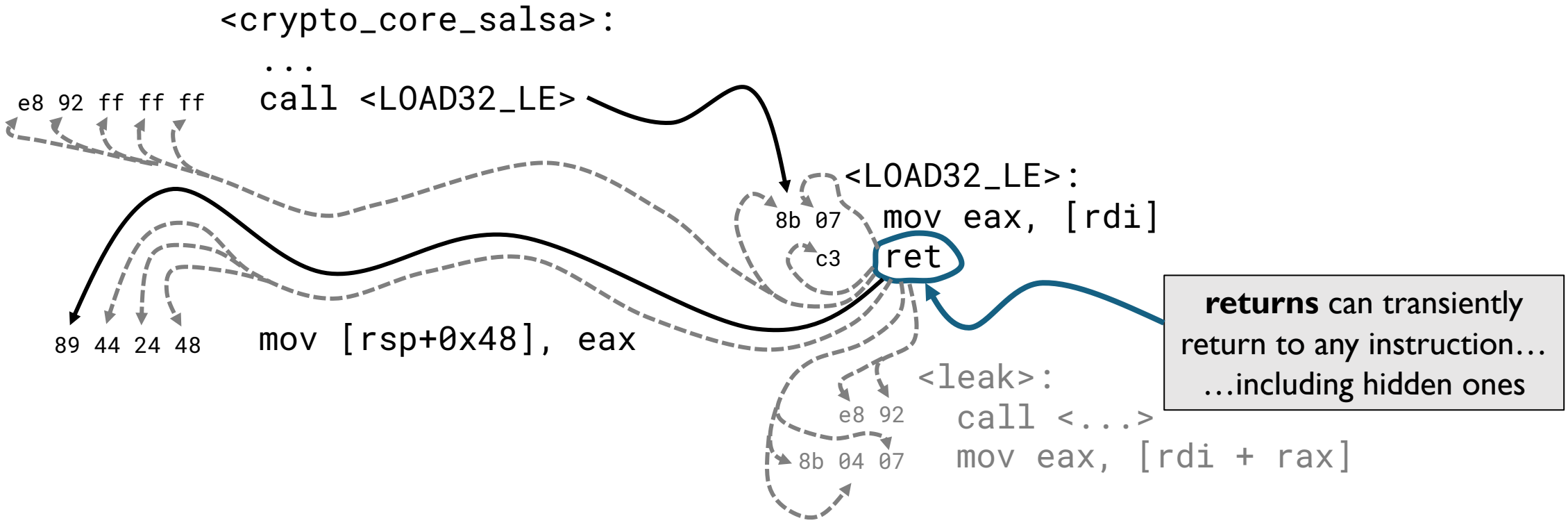
Root-causing Spectre leakage:

3. **Existing techniques inapplicable to constant-time**

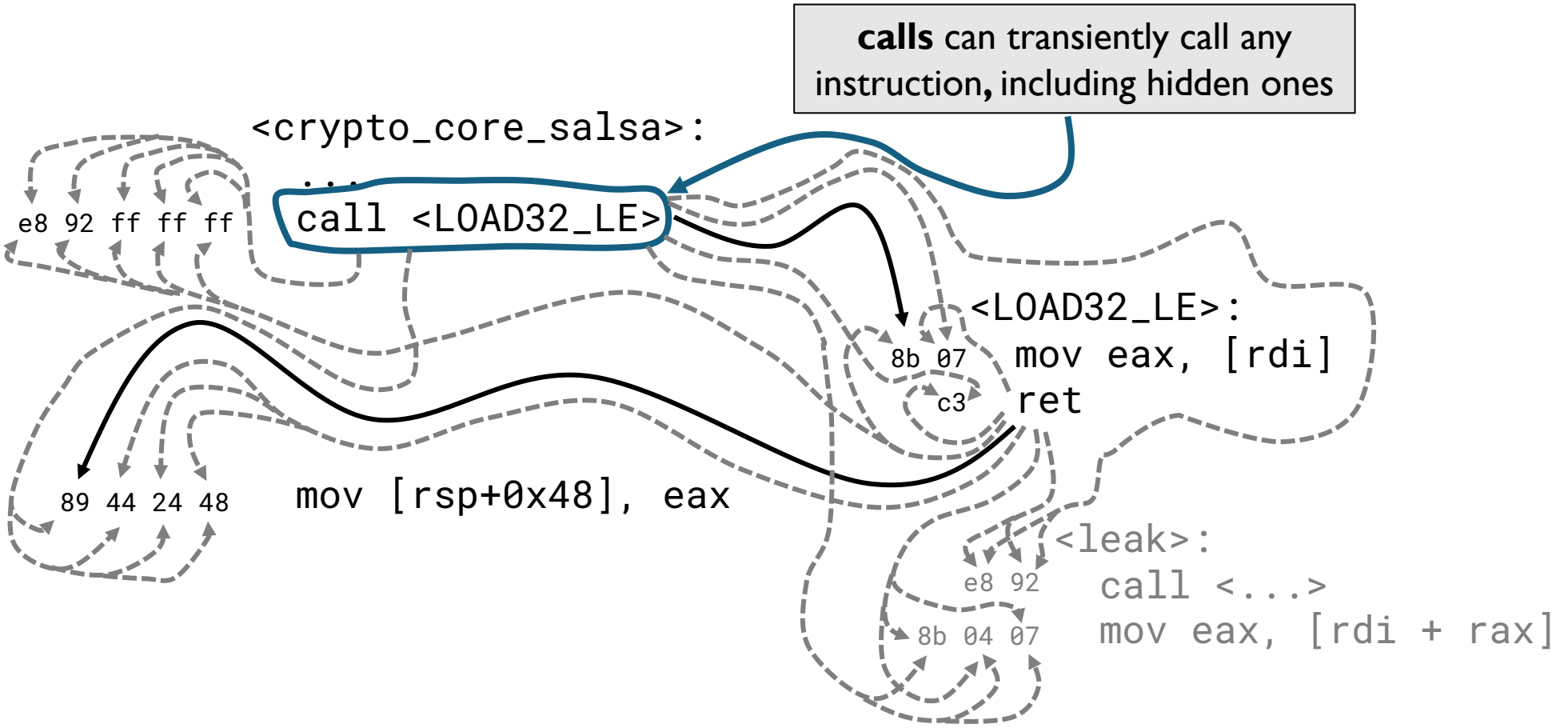
Challenge #1: Unconstrained speculative control-flow renders comprehensive software defenses against Spectre impractical



Challenge #1: Unconstrained speculative control-flow renders comprehensive software defenses against Spectre impractical

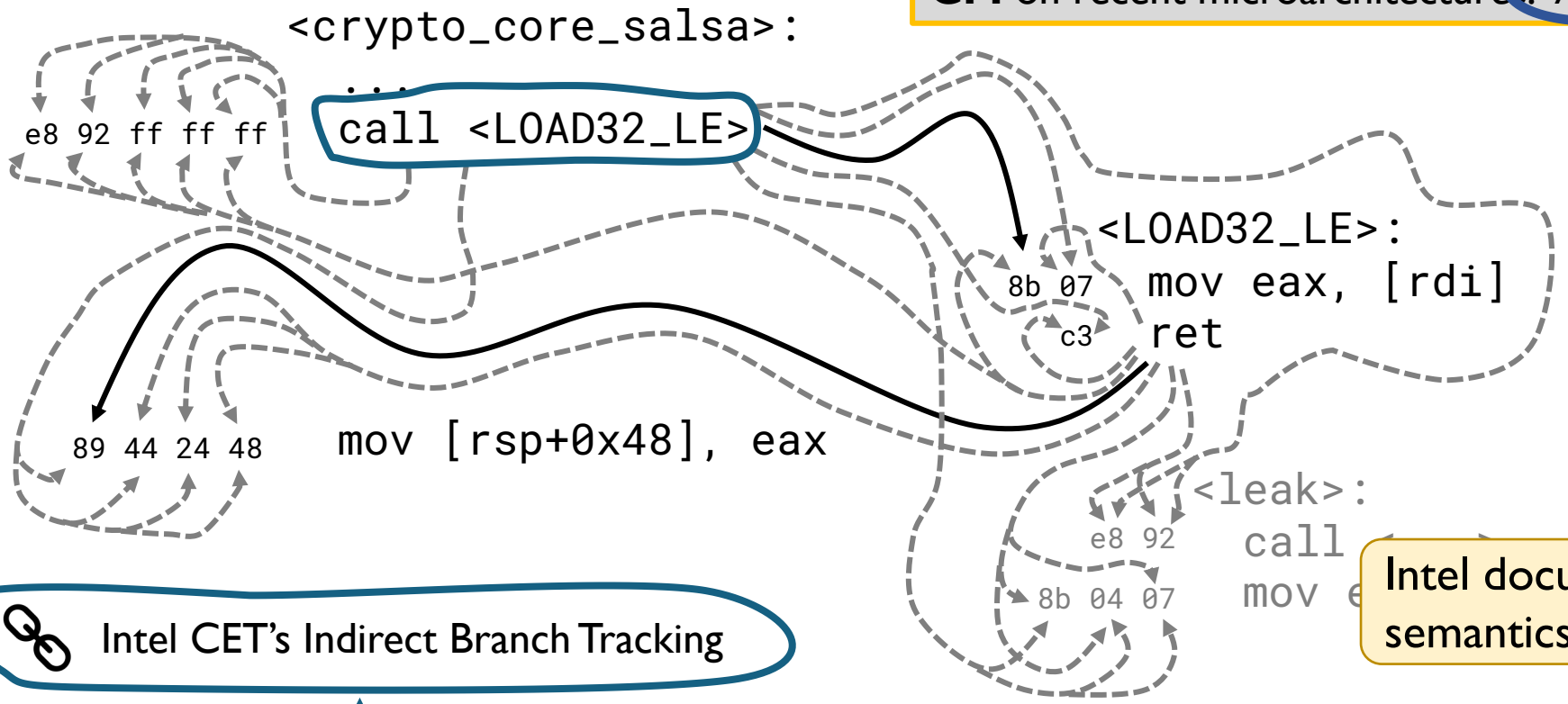


Challenge #1: Unconstrained speculative control-flow renders comprehensive software defenses against Spectre impractical



Solution #1: Use Intel ISA extensions to constrain speculative control-flow

Constrain speculative control-flow with **Intel Control-Flow Enforcement Technology (CET)**, which provides **speculative CFI** on recent microarchitectures: *Alder Lake-N, Arizona Beach*



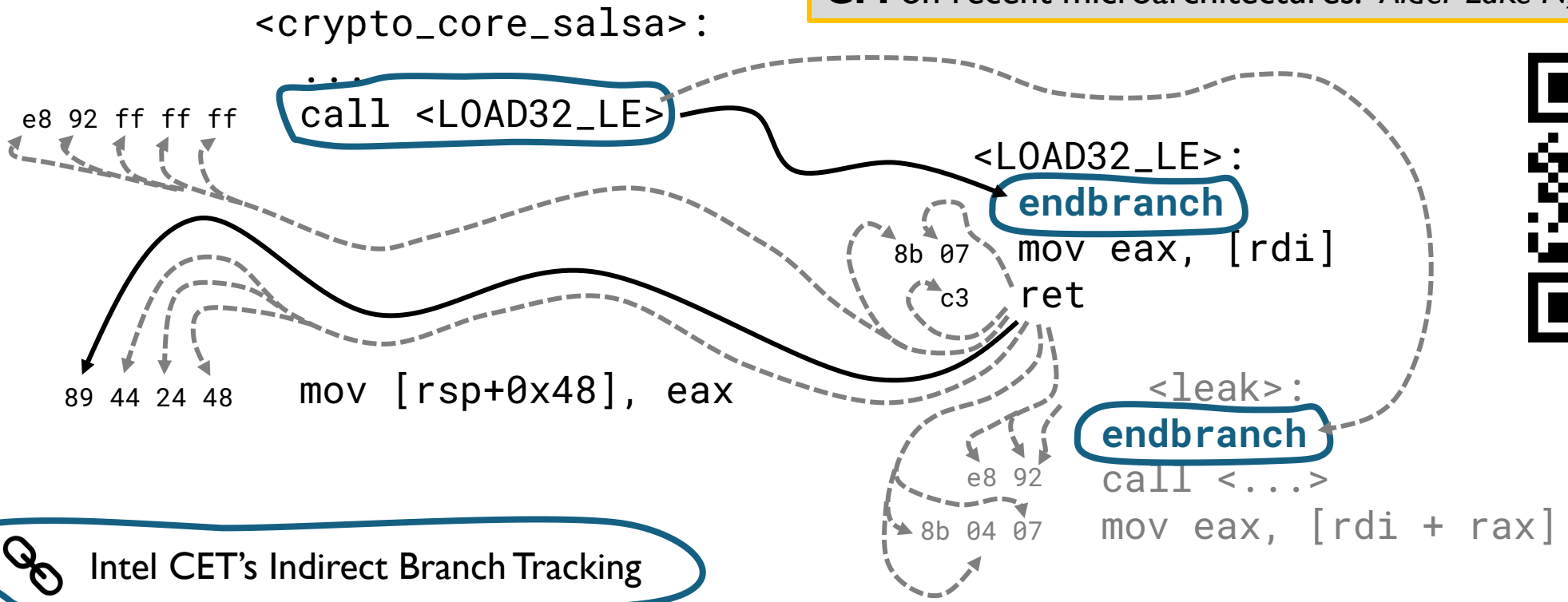
Intel documented CET's speculative semantics in response to SERBERUS

Intel CET's Indirect Branch Tracking

calls/jumps can only transiently jump to **endbranch** instructions

Solution #1: Use Intel ISA extensions to constrain speculative control-flow

Constrain speculative control-flow with **Intel Control-Flow Enforcement Technology (CET)**, which provides **speculative CFI** on recent microarchitectures: *Alder Lake-N, Arizona Beach*

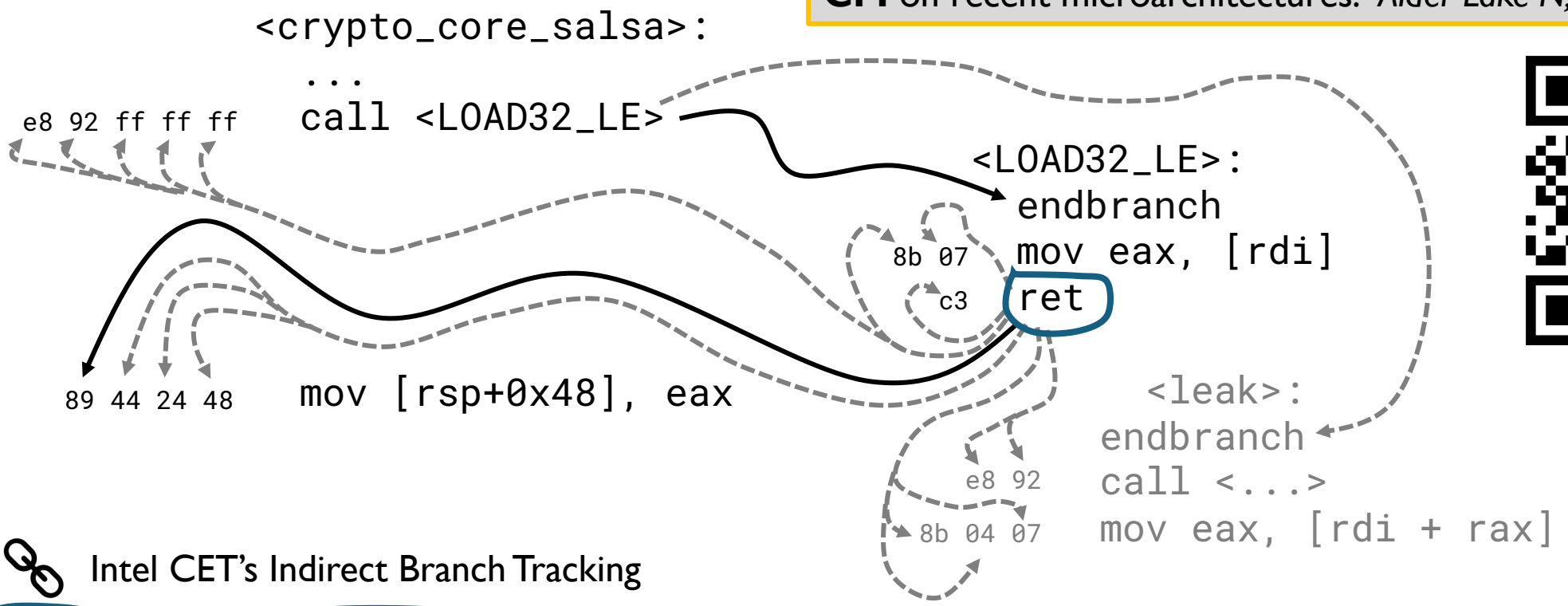


Intel CET's Indirect Branch Tracking

calls/jumps can only transiently jump to **endbranch** instructions

Solution #1: Use Intel ISA extensions to constrain speculative control-flow

Constrain speculative control-flow with **Intel Control-Flow Enforcement Technology (CET)**, which provides **speculative CFI** on recent microarchitectures: *Alder Lake-N, Arizona Beach*

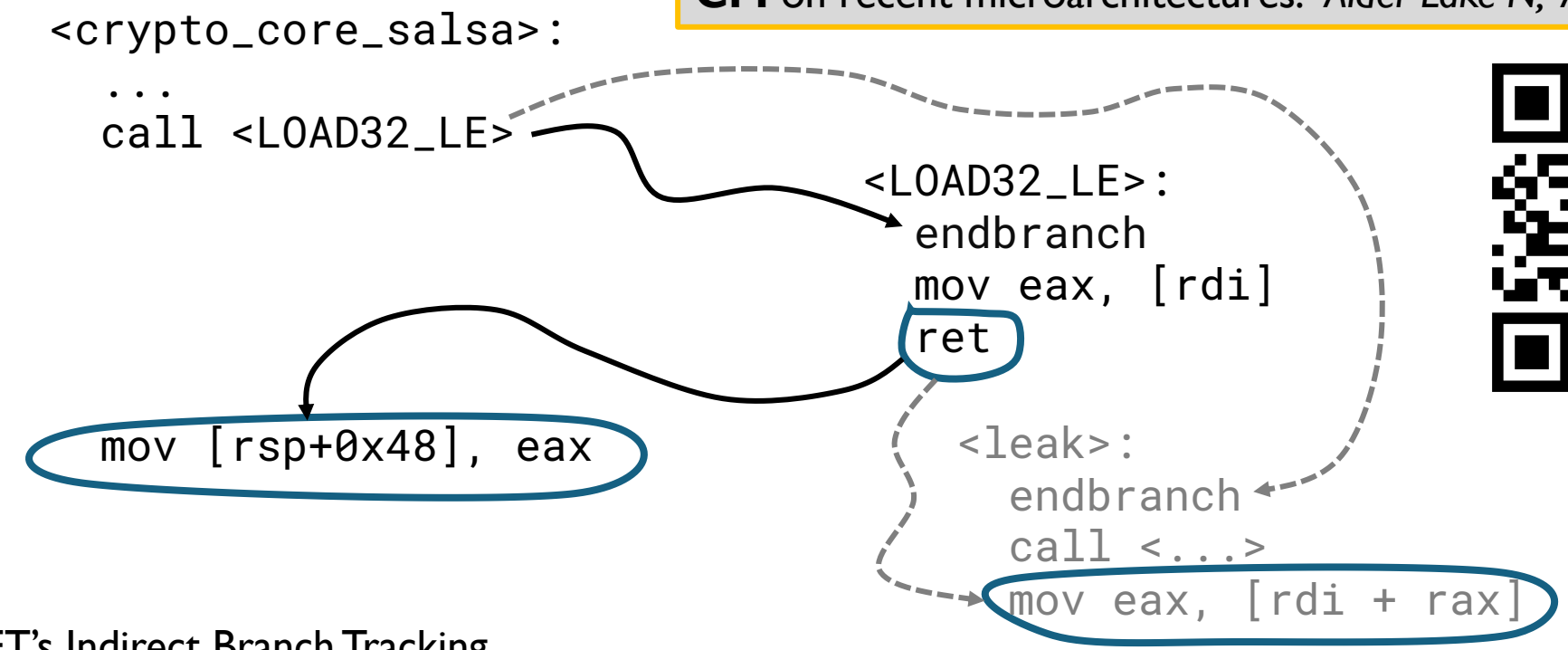


- Intel CET's Indirect Branch Tracking
- Intel CET's Shadow Stack
- RRSBA Disable speculation control

returns can only return to callsites

Solution #1: Use Intel ISA extensions to constrain speculative control-flow

Constrain speculative control-flow with **Intel Control-Flow Enforcement Technology (CET)**, which provides **speculative CFI** on recent microarchitectures: *Alder Lake-N, Arizona Beach*



- Intel CET's Indirect Branch Tracking
- Intel CET's Shadow Stack
- RRSBA Disable speculation control

returns can only return to callsites

Solution #1: Use Intel ISA extensions to constrain speculative control-flow

Constrain speculative control-flow with **Intel Control-Flow Enforcement Technology (CET)**, which provides **speculative CFI** on recent microarchitectures: *Alder Lake-N, Arizona Beach*

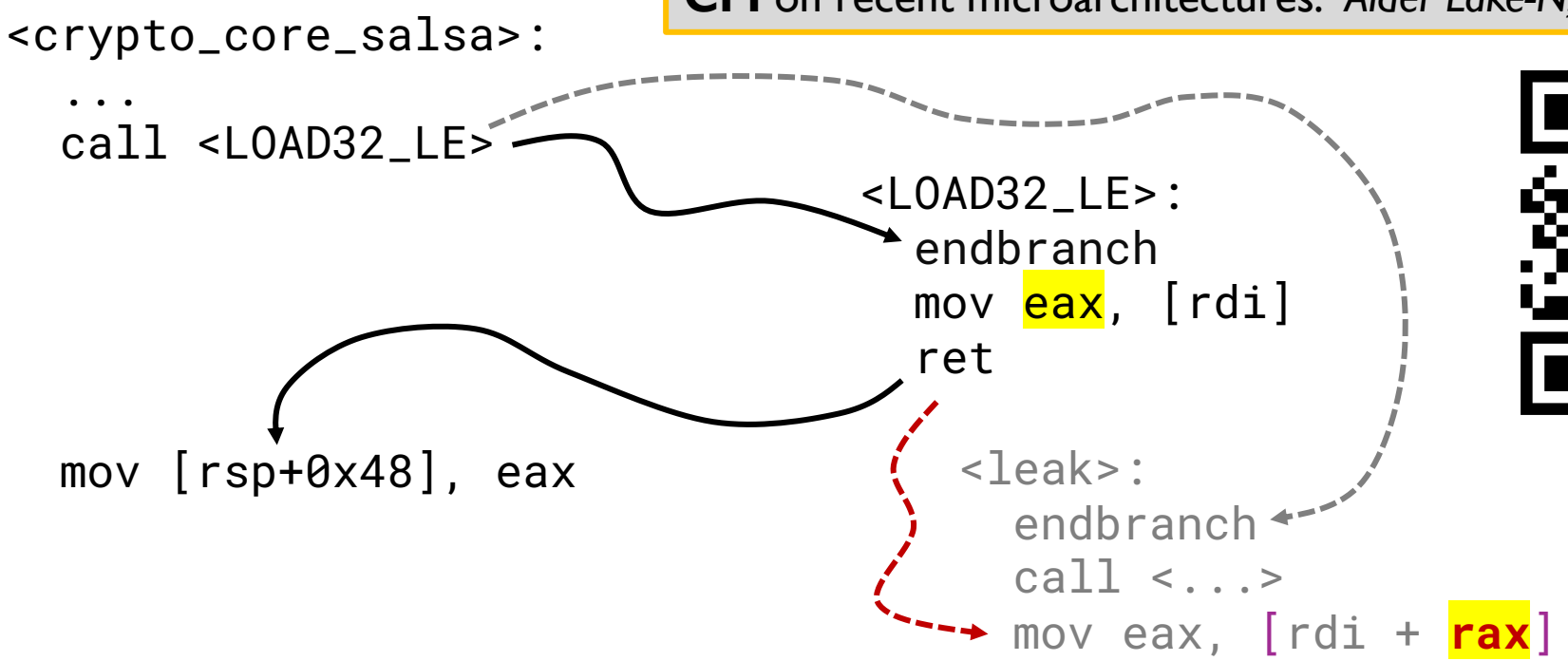
```
<crypto_core_salsa>:
  ...
  call <LOAD32_LE>

<LOAD32_LE>:
  endbranch
  mov eax, [rdi]
  ret

mov [rsp+0x48], eax

<leak>:
  endbranch
  call <...>
  mov eax, [rdi + rax]
```

- Intel CET's Indirect Branch Tracking
- Intel CET's Shadow Stack
- RRSBA Disable speculation control

Not Spectre defenses! Just happens to provide useful restrictions on some μarches

Solution #1: Use Intel ISA extensions to constrain speculative control-flow

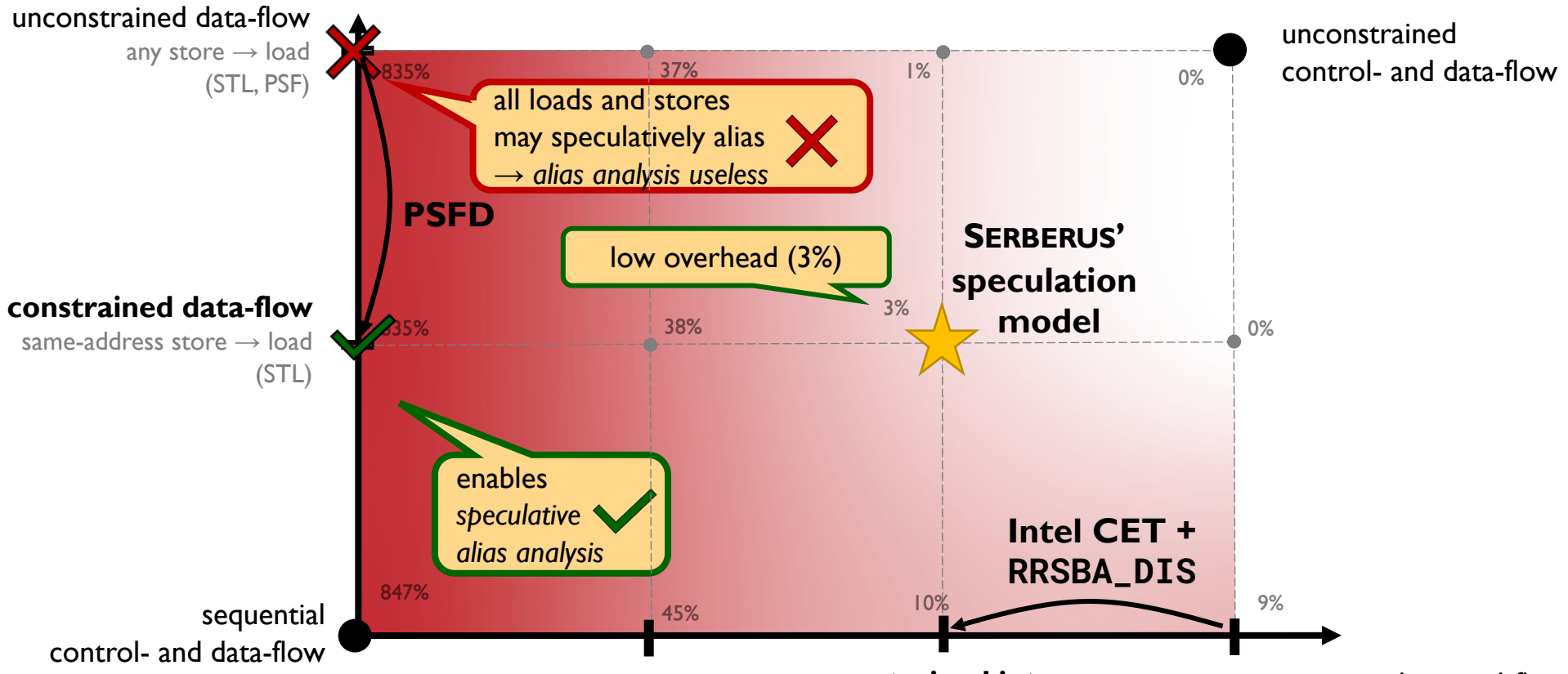
Constrain speculative control-flow with **Intel Control-Flow Enforcement Technology (CET)**, which provides **speculative CFI** on recent microarchitectures: *Alder Lake-N, Arizona Beach*



- Intel CET's Indirect Branch Tracking
- Intel CET's Shadow Stack
- RRSBA Disable speculation control

still have transient leakage, but it's much easier to reason about

Lightweight Speculation Constraints

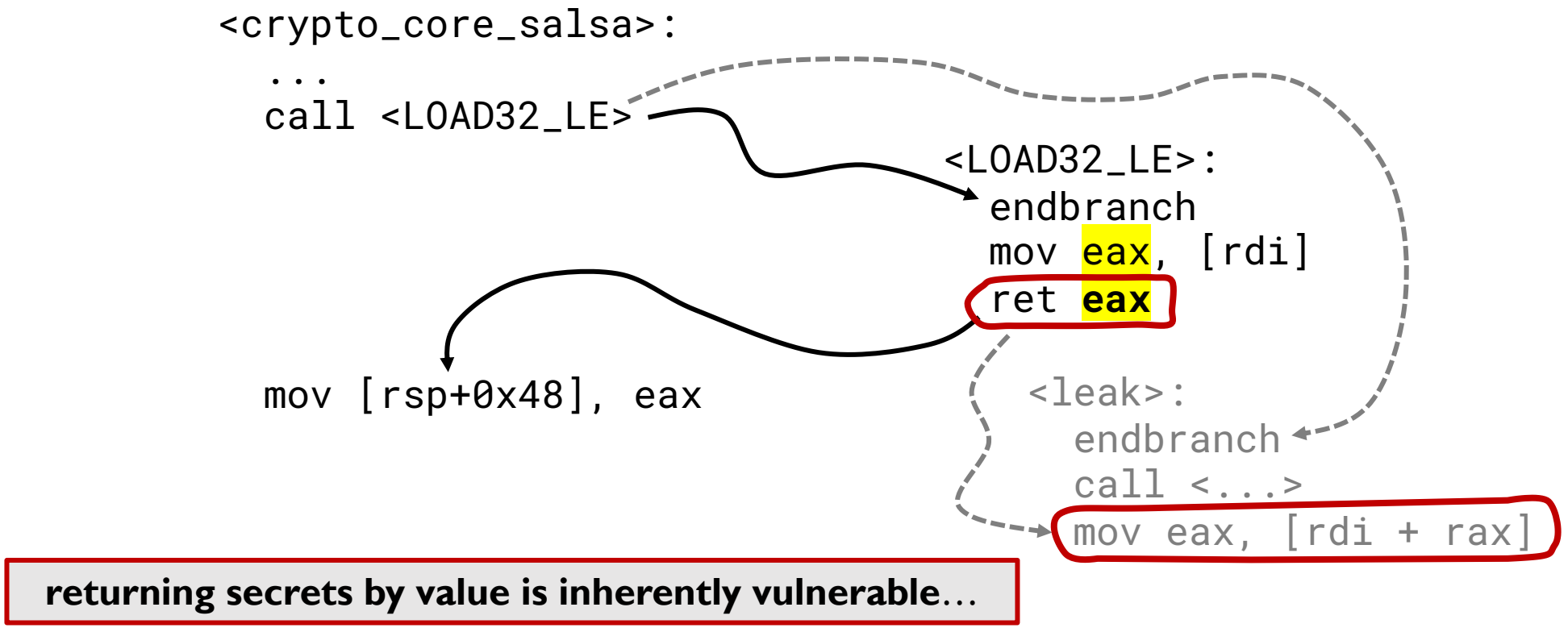


Legend:
color indicates overhead, logarithmically scaled

high — low

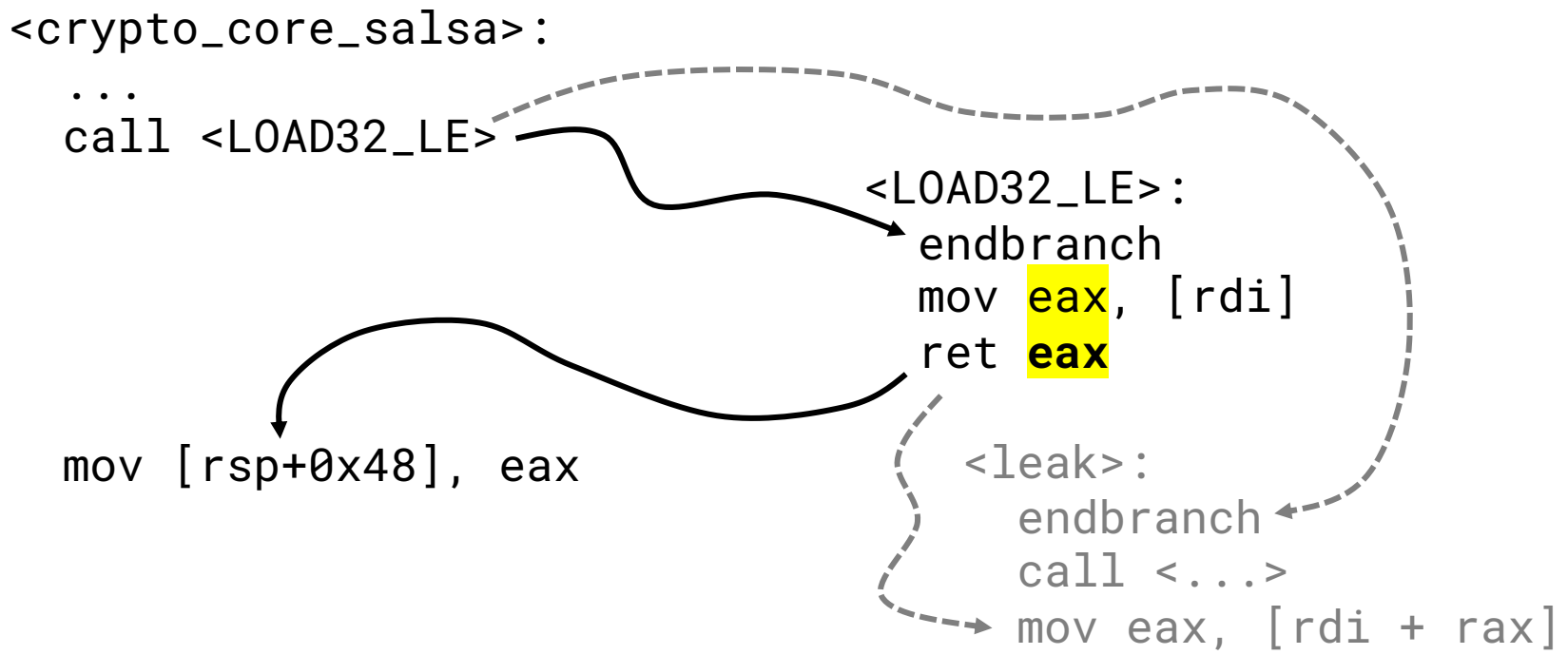
Note: overheads measured on SPEC2017 intspeed.

Challenge #2: Passing secrets by value is unsafe in the Spectre era



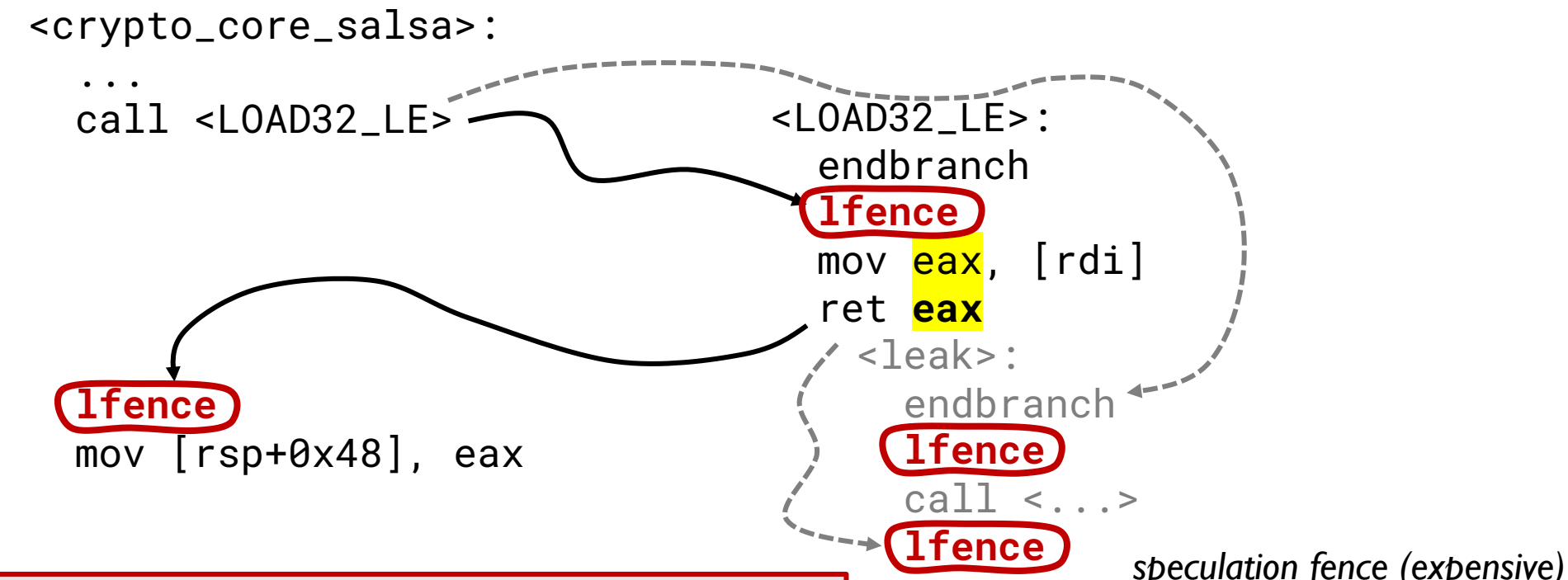
returning secrets by value is inherently vulnerable…

Challenge #2: Passing secrets by value is unsafe in the Spectre era



returning secrets by value is inherently vulnerable…
…and requires expensive protections

Challenge #2: Passing secrets by value is unsafe in the Spectre era



returning secrets by value is inherently vulnerable…
…and requires expensive protections

Solution #2: Static Constant-Time Programming, a strengthening of traditional constant-time

SERBERUS' Solution:
static constant-time (CTS) programming
extends constant-time with:

static security types of variables (see paper)

pass secret arguments by **reference**, not **value**

Solution #2: Static Constant-Time Programming

pass secret arguments by **reference**, not **value**

```
u32 LOAD32_LE(void *p)   ⟶   void LOAD32_LE(void *p, u32 *out)
```

```
<LOAD32_LE>:
  endbranch
  mov eax, [rdi]
  ret eax
```

violates static constant-time!

```
mov [rsp+0x48], eax
```

```
<leak>:
  endbranch
  call <...>
  mov eax, [rdi + rax]
```

Solution #2: Static Constant-Time Programming

pass secret arguments by **reference**, not **value**

```
u32 LOAD32_LE(void *p)
```

```
void LOAD32_LE(void *p, u32 *out)
```

```
<LOAD32_LE>:
  endbranch
  mov ecx, [rdi]
  mov [rsi], ecx
  ret
```

Need a formally-grounded way to identify the **root cause of leakage**

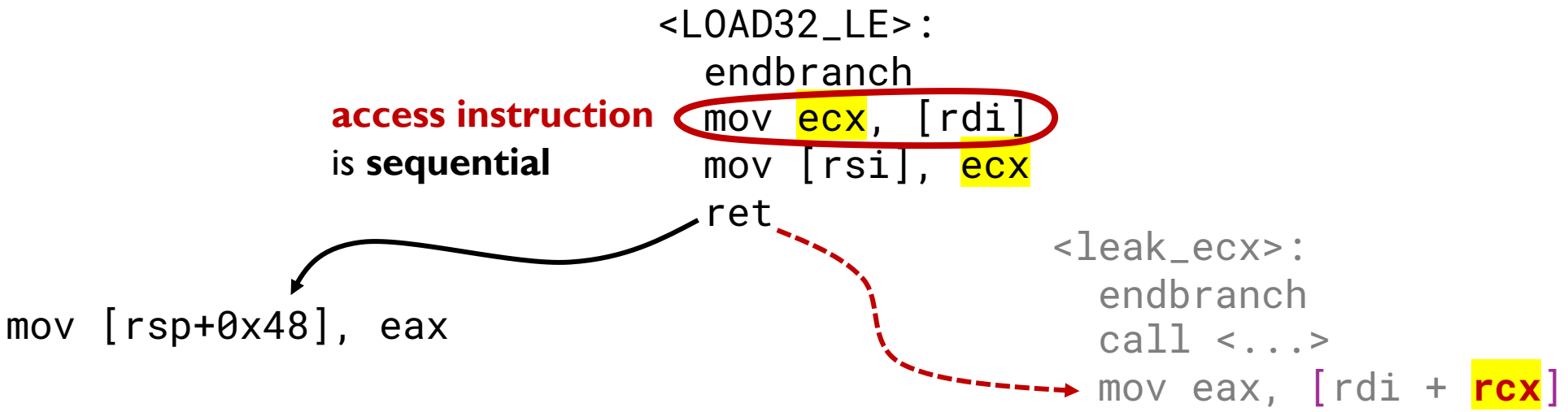
```
mov [rsp+0x48], eax
```

```
<leak>:
  endbranch
  call <...>
  mov eax, [rdi + rax]
```

```
<leak_ecx>:
  endbranch
  call <...>
  mov eax, [rdi + rcx]
```

Challenge #3: Existing defenses do not capture the root cause of Spectre leakage in CT/CTS code

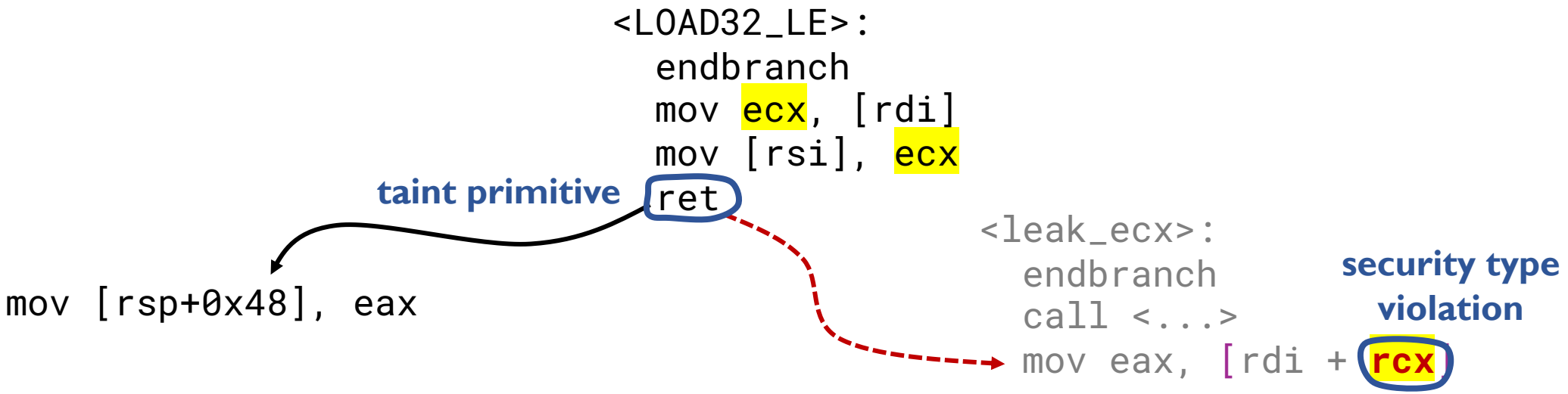
Prior work studies transient **access instructions** [Yu+ MICRO'19], which load secrets into registers.



Not applicable to (static) constant-time programs, which sequentially access secrets

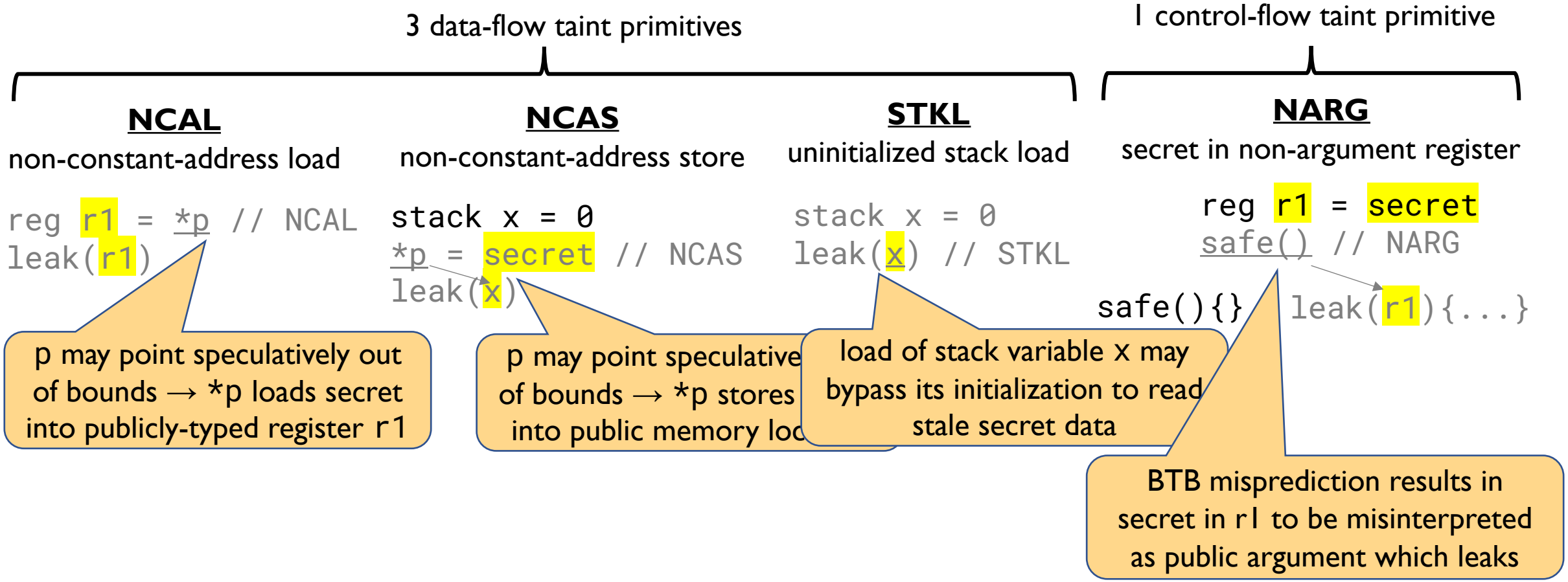
Solution #3: Taint Primitives

We propose the concept of **taint primitives**, instructions that cause a **publicly-typed register** to **transiently hold a secret**, i.e., a security type violation.

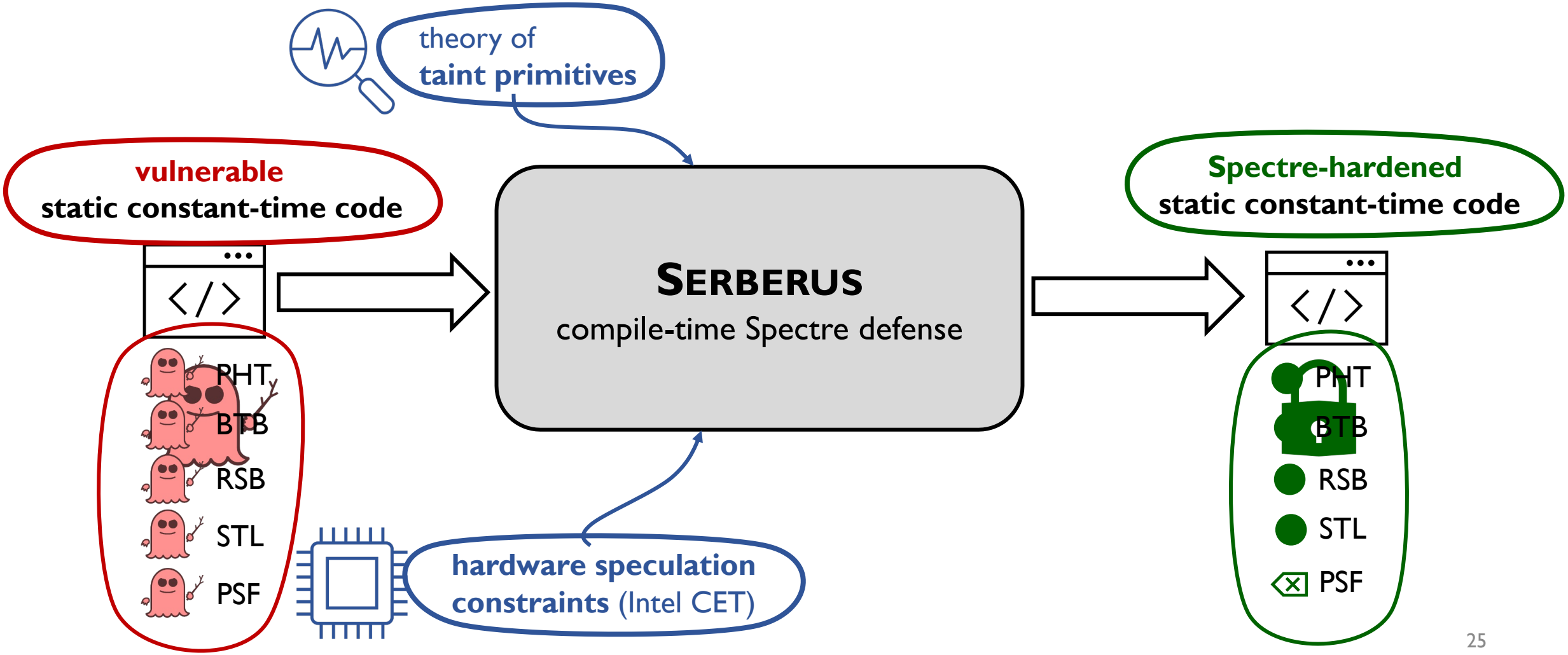


Taint primitives are **necessary** to transiently leak secrets in static constant-time programs.

Taint Primitives in CTS Programs



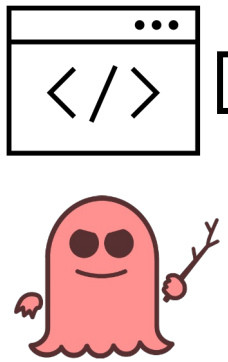
SERBERUS: a comprehensive Spectre defense for static constant-time code



SERBERUS: a comprehensive Spectre defense for static constant-time code

*implemented for **LLVM***

vulnerable
static constant-time code



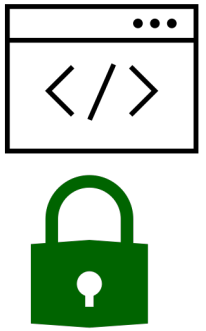
SERBERUS
compile-time Spectre defense

Function-Private Stacks
STKL

Register Cleaning
NARG

Fence Insertion
NCAL
NCAS

Spectre-hardened
static constant-time code

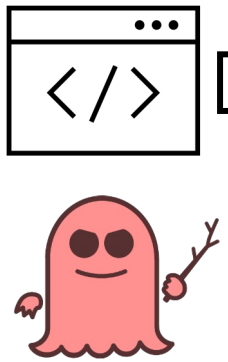


collectively **eliminate *all* taint primitives**
from ***all* transient executions** of the program

SERBERUS: a comprehensive Spectre defense for static constant-time code

*implemented for **LLVM***

vulnerable
static constant-time code



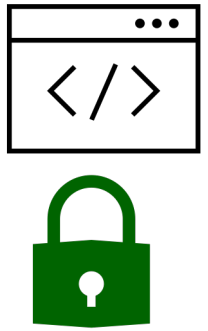
SERBERUS
compile-time Spectre defense

Function-Private Stacks
STKL

Register Cleaning
NARG

Fence Insertion
NCAL
NCAS

Spectre-hardened
static constant-time code



assigns distinct physical stacks to each function to eliminate taint primitives due to stack sharing

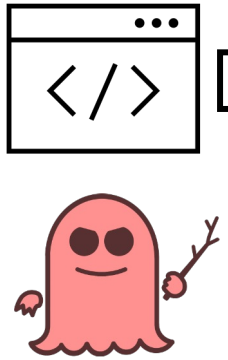
zeroes out registers that m secrets on call/return to elin interprocedural taint primit

inserts a minimal number of speculation fences to eliminate taint primitives not handled by other passes

SERBERUS: a comprehensive Spectre defense for static constant-time code

*implemented for **LLVM***

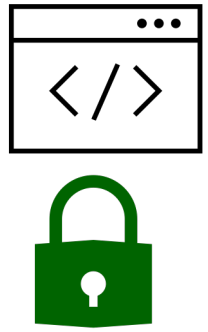
vulnerable
static constant-time code



SERBERUS
compile-time Spectre defense

Function-Private Stacks	***Register Cleaning***	***Fence Insertion***
STKL	NARG	NCAL, NCAS

Spectre-hardened
static constant-time code



programmer transparent: SERBERUS does not need to know secrecy labels

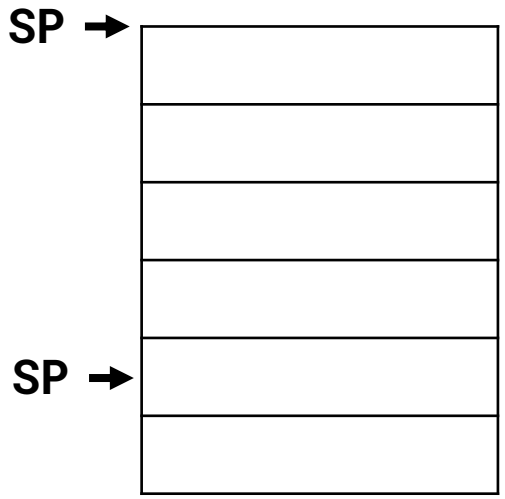
SERBERUS' Function-Private Stacks Pass

Stack sharing is the root cause of STKL: a publicly-typed load may read a stale secret from prior procedure's stack frame.

STKL
uninitialized stack load

```
int x = 0;
y = A[x];
```

```
foo() {
  x = secret;
  ...
}
```

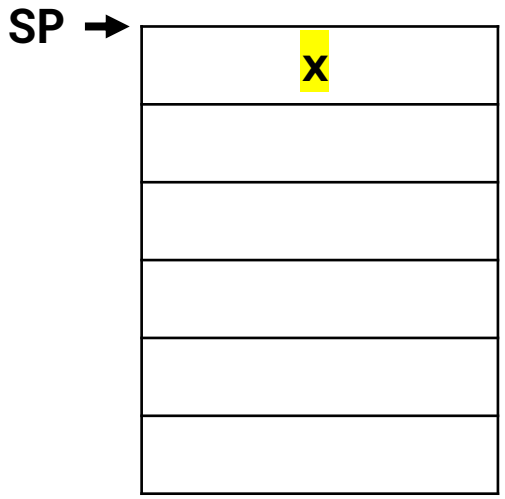


SERBERUS' Function-Private Stacks Pass

STKL
uninitialized stack load

Stack sharing is the root cause of STKL: a publicly-typed load may read a stale secret from prior procedure's stack frame.

```
bar() {
  y = 0;
  z = A[y];
}
```



Solution: allocate a **private stack** to each procedure.

```
foo: ENDCALL
   + LD [ZR+PSP_F],SP // load private SP
     SUB SP,SP,k      // frame allocation
   + LD [SP+0],ZR     // probe for overflow
   + ST [ZR+PSP_F],SP // store private SP
     ...
     CALL r1
   + LD [ZR+PSP_F],SP // load private SP
     ...
   + LD [SP+0],ZR     // probe for underflow
     ADD SP,SP,k      // frame deallocation
   + ST [ZR+PSP_F],SP // store private SP
     RET
```

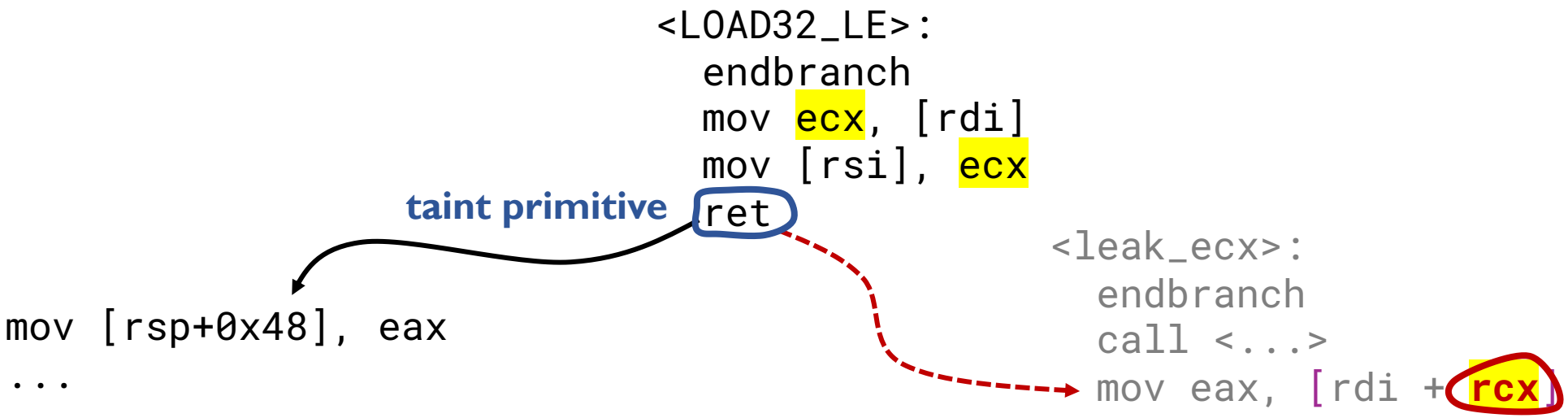
prologue

callsite

epilogue

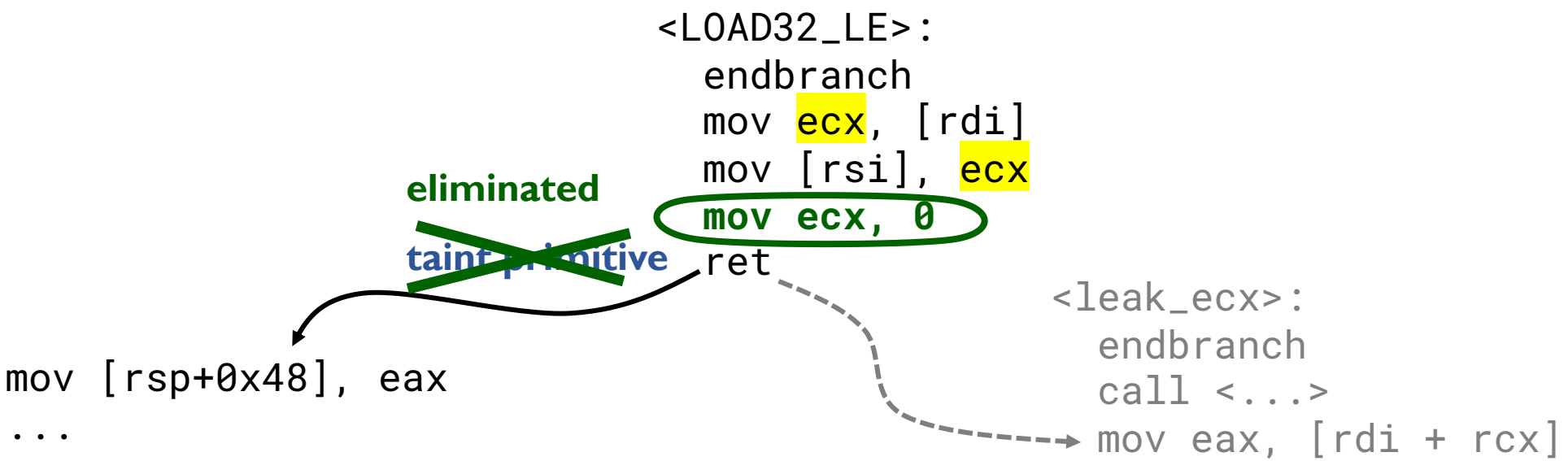
SERBERUS' Register Cleaning Pass

NARG
secret in non-argument register



SERBERUS' Register Cleaning Pass

NARG
secret in non-argument register



SERBERUS' Fence Insertion Pass

NCAL
NCAS

non-constant-address load/store

- Frames speculation fence (LFENCE) insertion as a ***minimum directed multicut*** problem over the **transient control-flow graph**
- **Sources** are loads or stores that may access secrets
- **Sinks** are dependent transmitters

Original Procedure

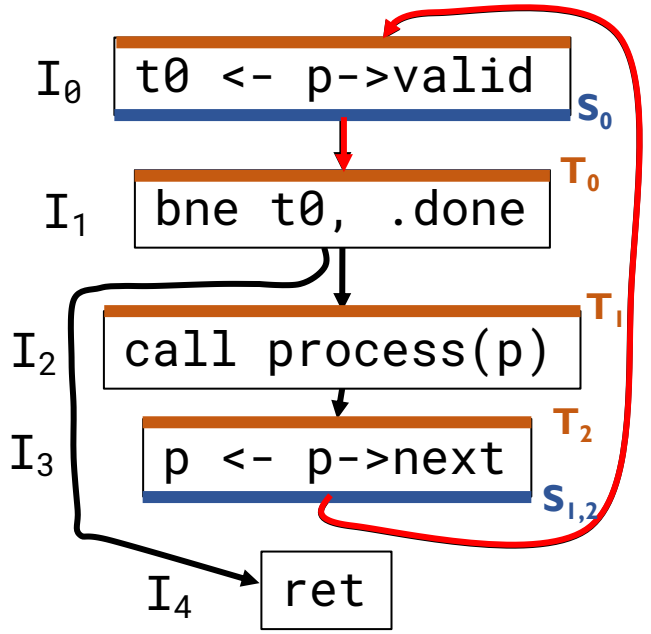
```
while (p->valid) {
  process(p);
  p = p->next;
}
```

Source-Sink Pairs

(I_0, I_1) (I_3, I_0)

(I_3, I_2)

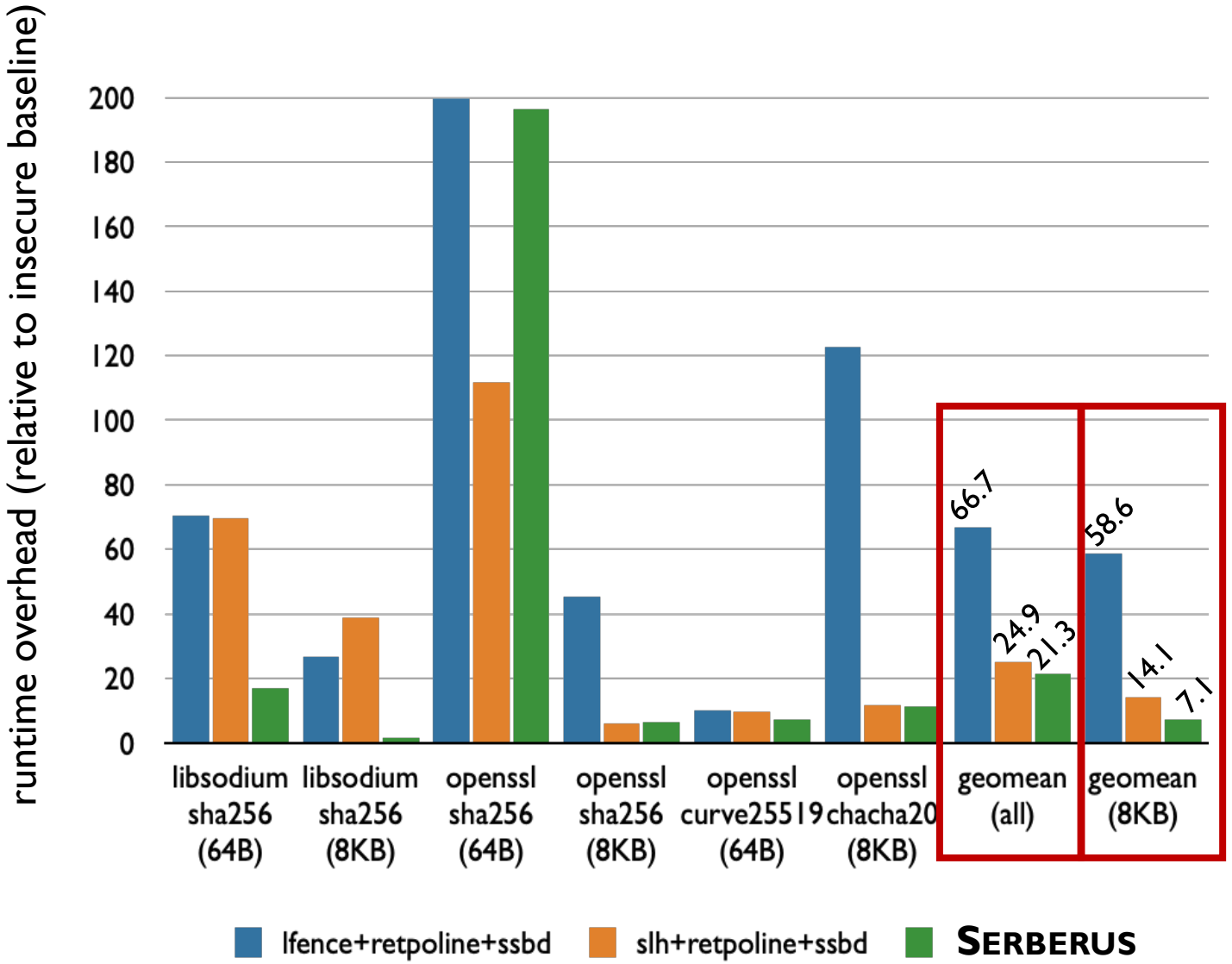
Transient CFG



Mitigated Procedure

```
while (t0 = p->valid,
       LFENCE(),
       t0){
  process(p);
  p = p->next;
  LFENCE();
}
```

SERBERUS' Performance



Evaluated Mitigations

mitigation	PHT	BTB	RSB	STL	PSF
insecure baseline					
lfence+retpoline+ssbd	✓	✓		✓	✓
slh+retpoline+ssbd	~	✓		✓	✓
SERBERUS (this paper)	✓	✓	✓	✓	✓

SERBERUS outperforms state-of-the-art mitigations in the crypto primitives we evaluate while offering **stronger security guarantees**.

Comparison to Prior Work

Legend

⌫ disable speculation

● secure speculation

deployable mitigations targeting CT	**PHT** conditional branch pred	**BTB** indirect branch pred	**RSB** return prediction	**STL** store-to-load fwding pred	**PSF** predictive store fwding
Intel LFENCE	⌫				
UltimateSLH [Zhang+ USENIX'23]	●				
Blade [Vassena+ POPL'21]	●				
selSLH [Shivakumar+ S&P'23]	●				
Misspec. types [Shivakumar+ S&P'23]	●				
retpoline		⌫			
IPRED_DIS		⌫			
SSBD				⌫	⌫
PSFD					⌫
Securest SOTA	●	⌫		⌫	⌫
SERBERUS [Mosier+ S&P'24]	●	●	●	●	⌫

Extras in the Paper

- We introduce "**Abstract Speculative Processor" (ASP),** an operational semantics modeling transient execution on a speculative, out-of-order processor
- We **formalize CTS programming** for ASP programs
- We **formally define taint primitives** and **prove** the existence of the **four classes of taint primitives** in CTS programs on ASP
- We **formalize SERBERUS' three passes** as transformations on ASP programs
- We **prove the correctness and security guarantees** of SERBERUS' passes on ASP

Takeaways

- **SERBERUS is the first comprehensive, deployable defense** for protecting constant-time code against Spectre leakage.
- **SERBERUS outperforms state-of-the-art defenses** despite offering stronger security guarantees.
- **Proof of SERBERUS' security guarantees**: see paper
- Working on upstreaming into LLVM

Interested in learning more on how Serberus works?
→ **Read our paper!**

Interested in protecting your code with Serberus?
→ **Contact us!**

GitHub: https://github.com/nmosier/serberus

email: nmosier@stanford.edu