SERBERUS: Protecting Cryptographic Code from Spectres at Compile Time

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

October 30, 2025
Top Picks in Hardware and Embedded Security

[1] Stanford University [2] KTH Royal Institute of Technology

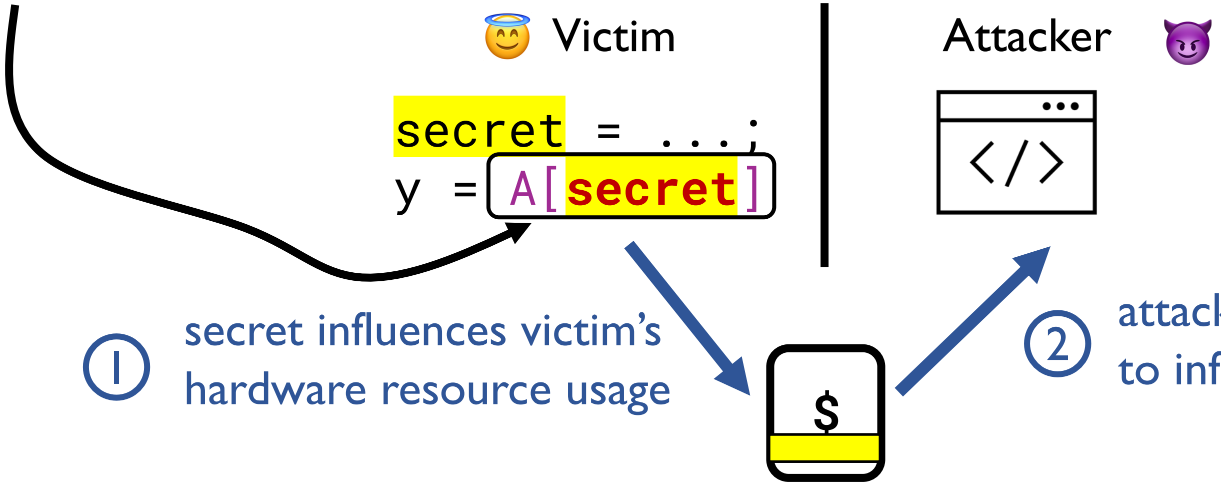
Original paper: N. Mosier, H. Nemati, J. C. Mitchell, and C. Trippel, "Serberus: Protecting cryptographic code from Spectres at compile-time," IEEE Symposium on Security and Privacy (S&P), 2024, pp. 4200–4219.

Hardware side-channel attacks

leak victim secrets to an attacker

transmitter

unsafe instruction whose execution exhibits
operand-dependent resource usage



😇 Victim

Attacker 😈

```
secret = ...;
y = A[secret]
```

① secret influences victim's hardware resource usage

② attacker observes usage to infer secret's value

Constant-time programming ensures secrets do not leak 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 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.

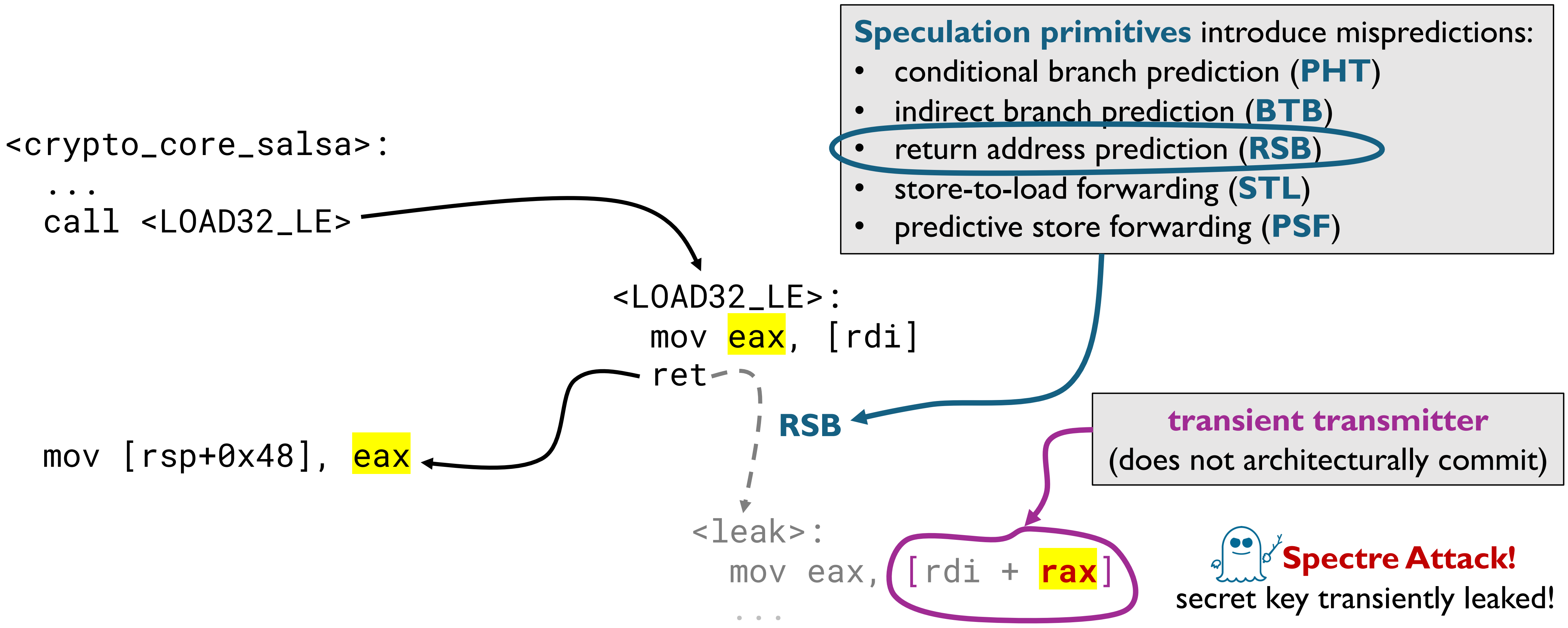
What about **transient execution**?

```
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**)

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

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

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

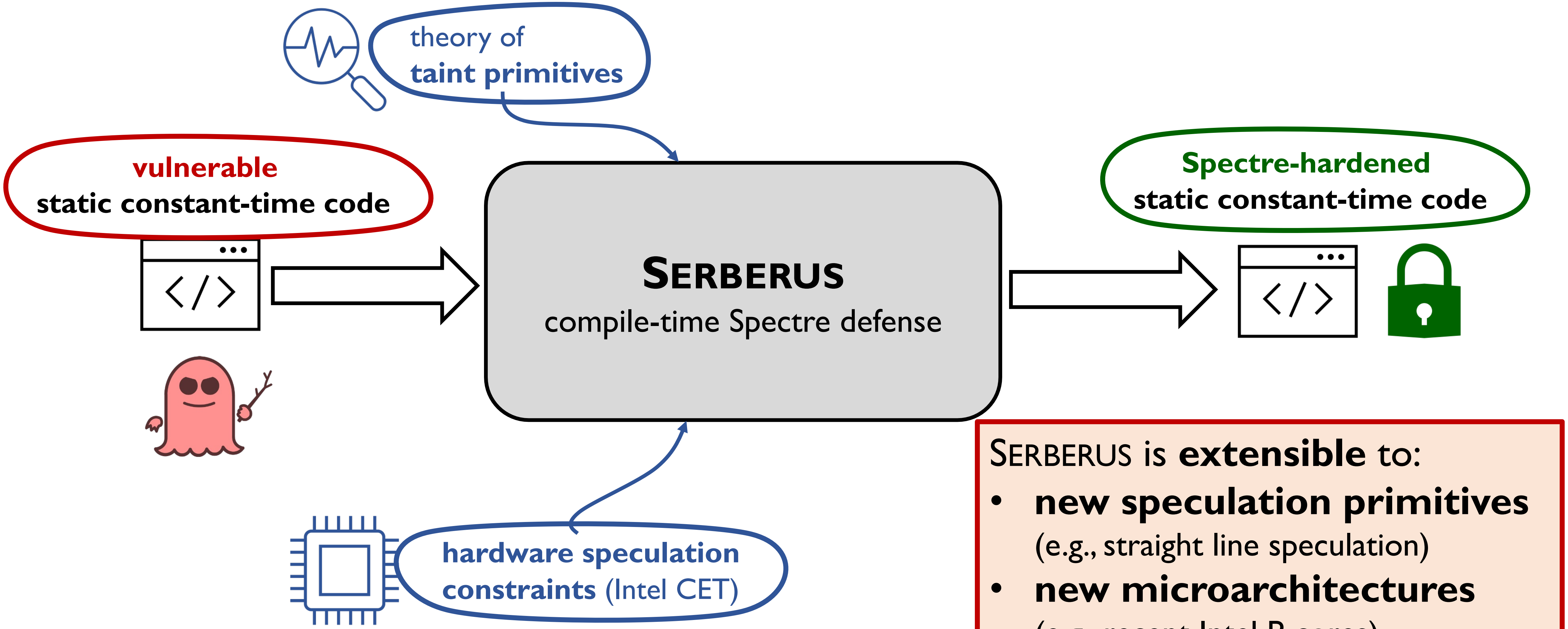
```
<leak>:
  mov eax, [rdi + rax]
  ...
```

transient transmitter
(does not architecturally commit)

Spectre Attack!
secret key transiently leaked!

SERBERUS

the first software Spectre defense to secure constant-time programs featuring PHT/BTB/RSB/STL/PSF speculation on existing hardware (recent Intel E-cores)



theory of **taint primitives**

vulnerable **static constant-time code**

SERBERUS
compile-time Spectre defense

Spectre-hardened **static constant-time code**

hardware speculation constraints (Intel CET)

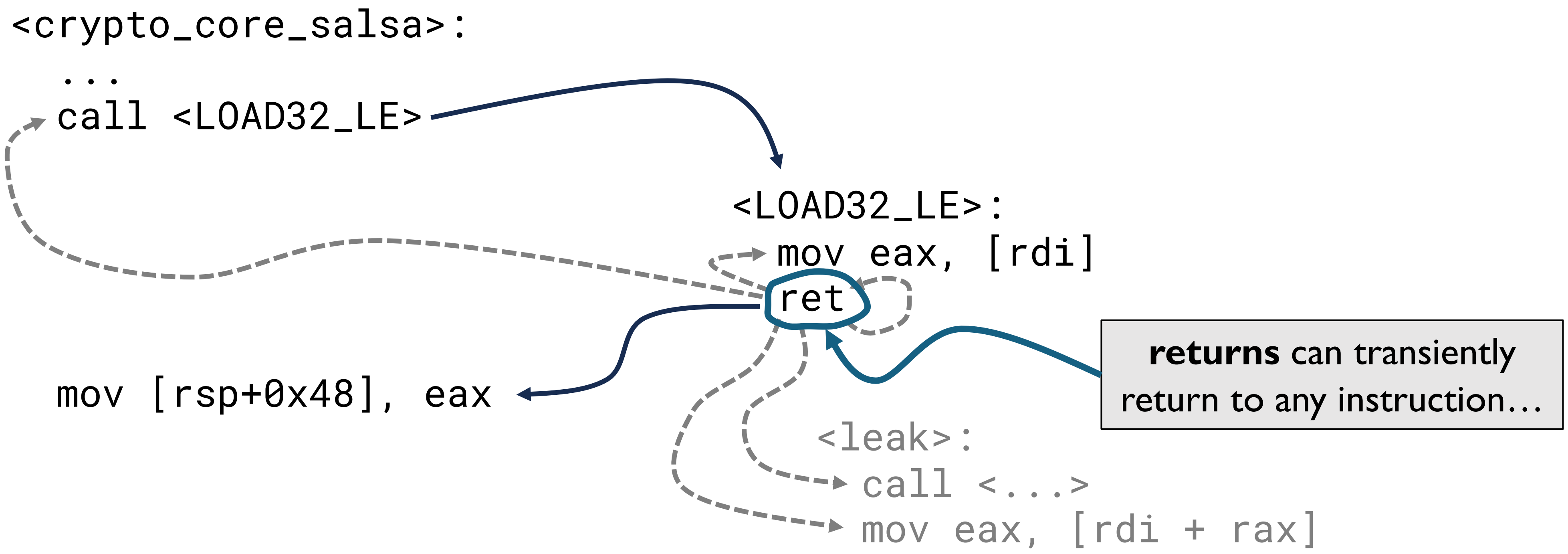
SERBERUS is **extensible** to:

- **new speculation primitives** (e.g., straight line speculation)
- **new microarchitectures** (e.g., recent Intel P-cores)

Outline

- Overcoming fundamental challenges for compile-time Spectre defenses
  1. Making transient control-flow analysis tractable
  2. Performantly securing code that passes secrets by value
  3. Root-causing Spectre leakage in constant-time code
- SERBERUS' design and evaluation
- SERBERUS' impact

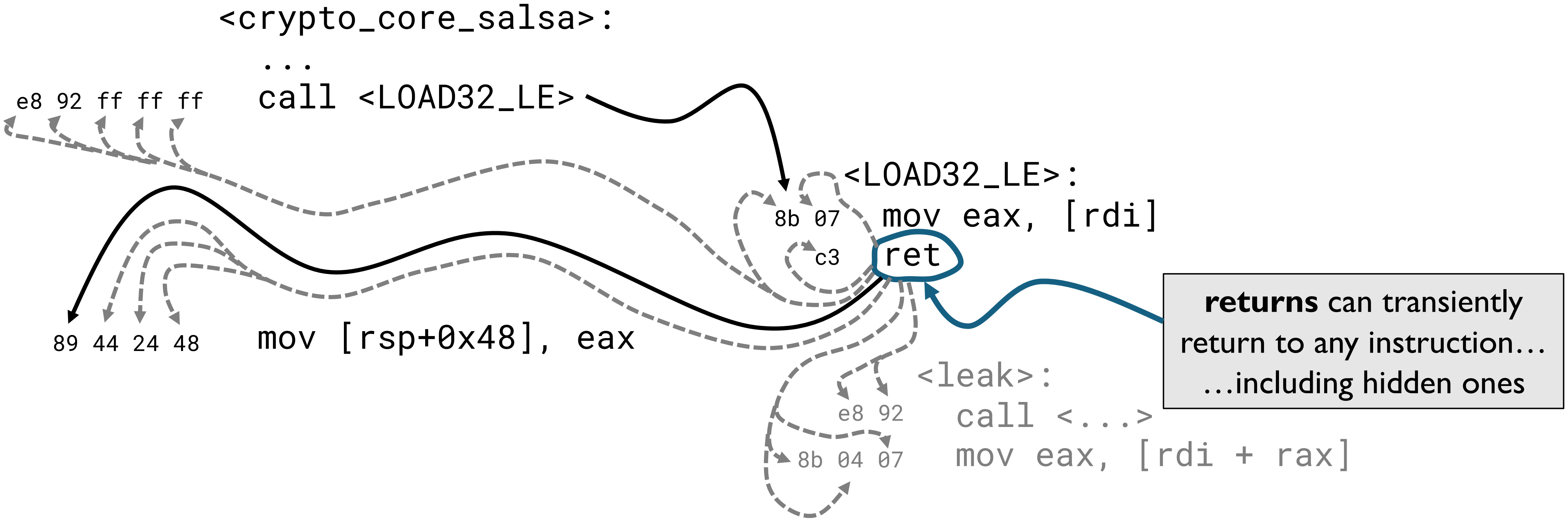
Challenge #1: Intractable to reason about unconstrained transient control-flow in software



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

                                   <LOAD32_LE>:
                                     mov eax, [rdi]
                                     ret

  mov [rsp+0x48], eax

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

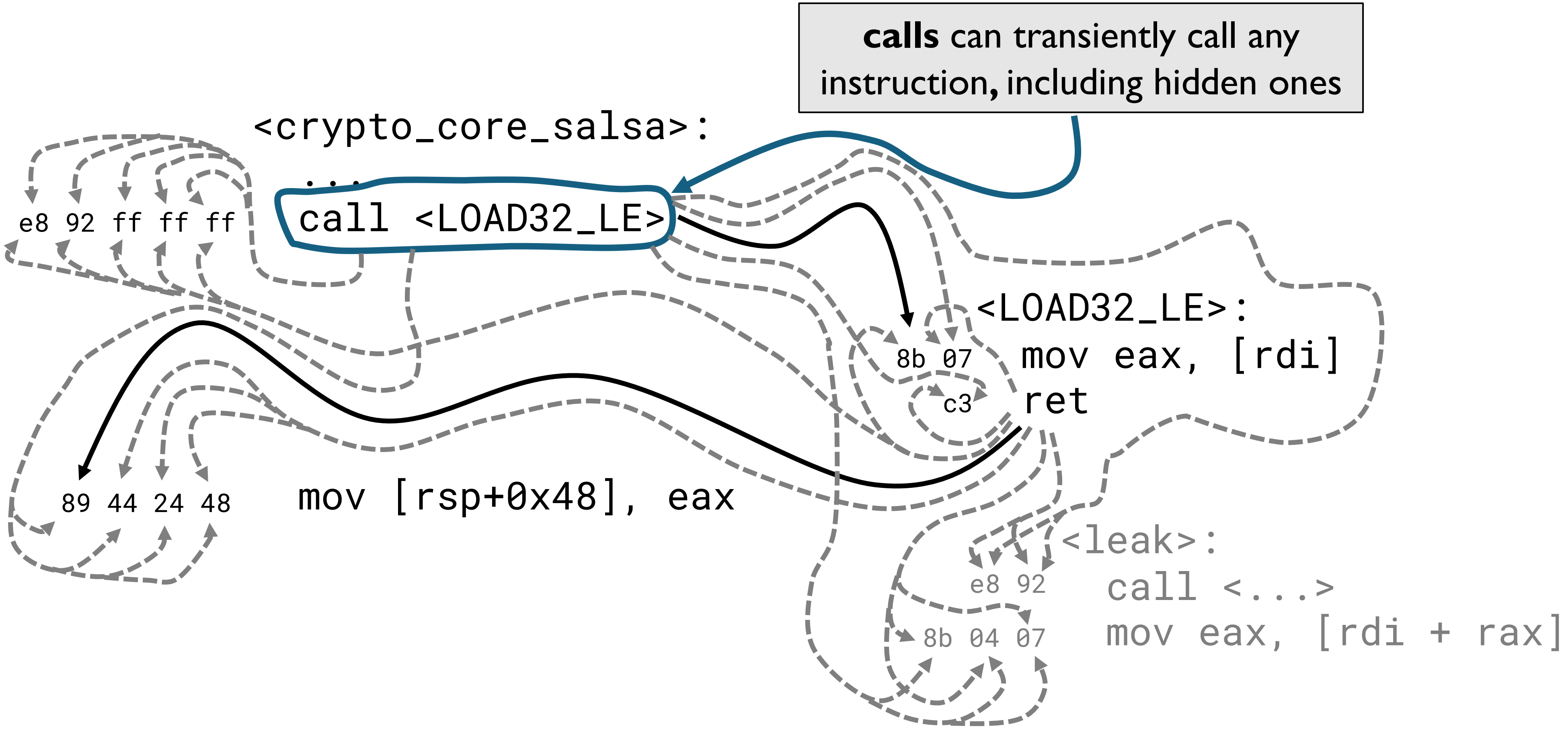
returns can transiently return to any instruction…

Challenge #1: Intractable to reason about unconstrained transient control-flow in software



returns can transiently return to any instruction…
…including hidden ones

Challenge #1: Intractable to reason about unconstrained transient control-flow in software

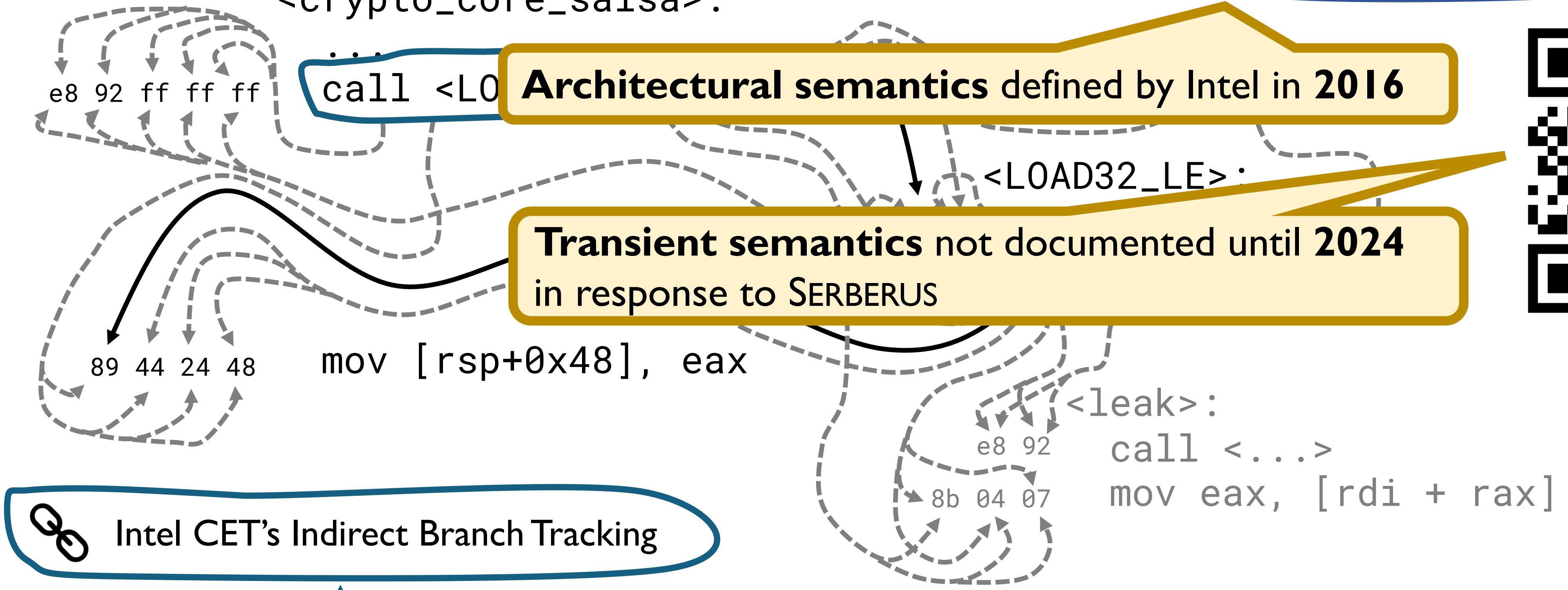


calls can transiently call any instruction, including hidden ones

```
                    <crypto_core_salsa>:
                      ...
e8 92 ff ff ff        call <LOAD32_LE>

                                              <LOAD32_LE>:
                                         8b 07  mov eax, [rdi]
                                            c3  ret

89 44 24 48           mov [rsp+0x48], eax

                                                 <leak>:
                                           e8 92  call <...>
                                        8b 04 07  mov eax, [rdi + rax]
```

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

Constrain transient control-flow with **Intel Control-Flow Enforcement Technology (CET)**, which provides **transient CFI** on E-core microarchitectures (e.g., *Gracemont*, *Crestmont*)



```
<crypto_core_salsa>:
...
e8 92 ff ff ff    call <LO

89 44 24 48       mov [rsp+0x48], eax

<LOAD32_LE>:

<leak>:
e8 92       call <...>
8b 04 07    mov eax, [rdi + rax]
```

Architectural semantics defined by Intel in **2016**

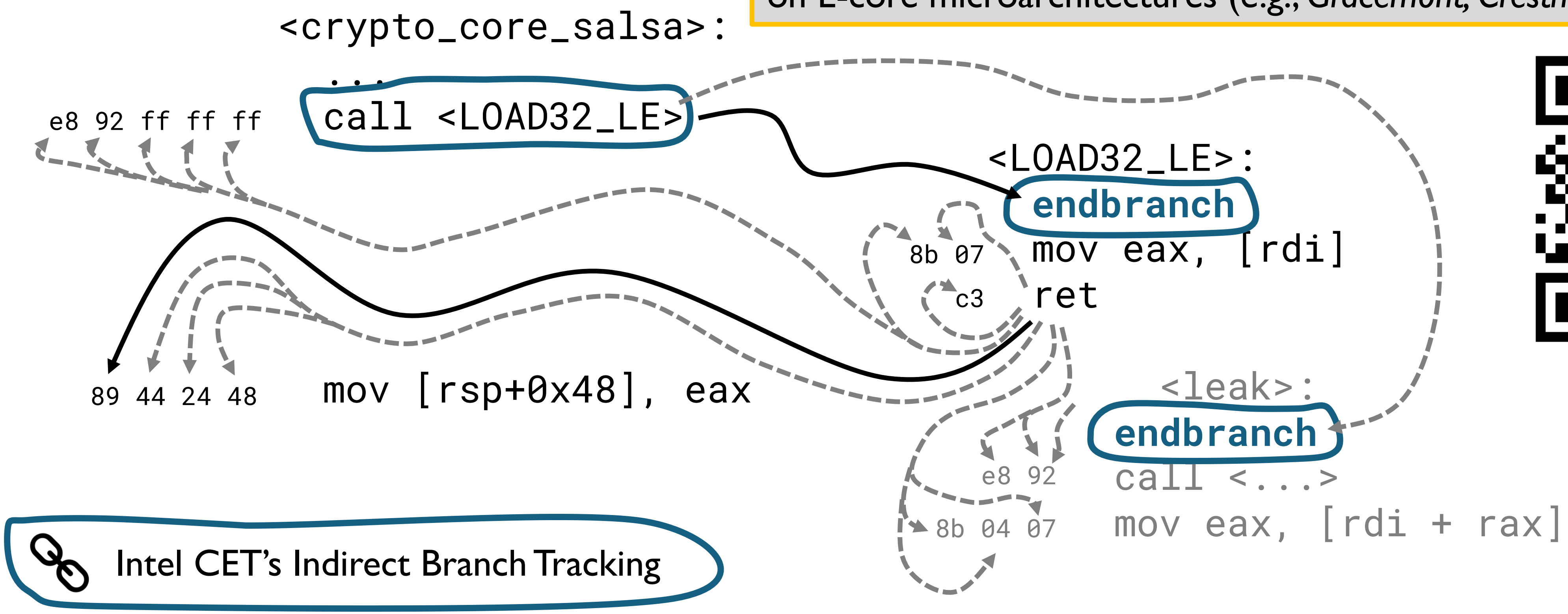
Transient semantics not documented until **2024** in response to SERBERUS

Intel CET's Indirect Branch Tracking

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

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

Constrain transient control-flow with **Intel Control-Flow Enforcement Technology (CET)**, which provides **transient CFI** on E-core microarchitectures (e.g., *Gracemont, Crestmont*)



```
<crypto_core_salsa>:
...
e8 92 ff ff ff    call <LOAD32_LE>

89 44 24 48       mov [rsp+0x48], eax
```

```
<LOAD32_LE>:
      endbranch
8b 07 mov eax, [rdi]
c3    ret
```

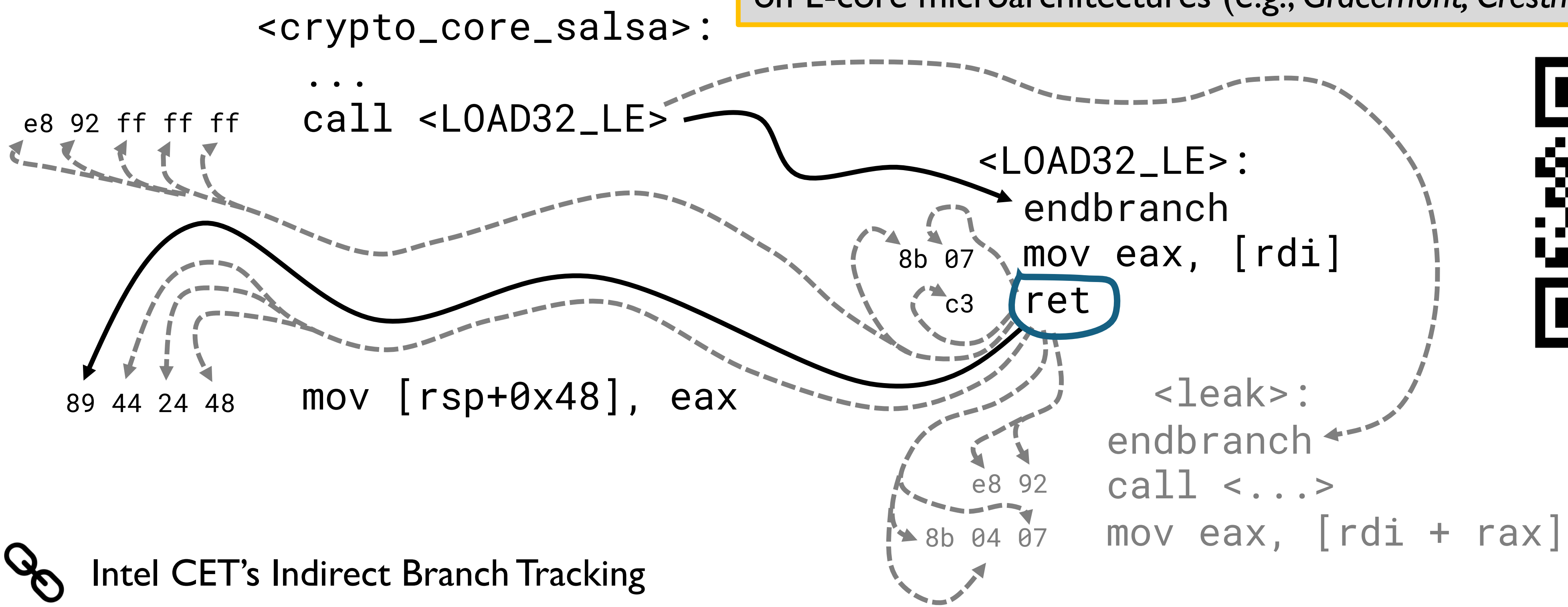
```
         <leak>:
         endbranch
e8 92    call <...>
8b 04 07 mov eax, [rdi + rax]
```

Intel CET's Indirect Branch Tracking

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

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

Constrain transient control-flow with **Intel Control-Flow Enforcement Technology (CET)**, which provides **transient CFI** on E-core microarchitectures (e.g., *Gracemont, Crestmont)*



```
<crypto_core_salsa>:
  ...
e8 92 ff ff ff   call <LOAD32_LE>

89 44 24 48      mov [rsp+0x48], eax
```

```
<LOAD32_LE>:
       endbranch
8b 07  mov eax, [rdi]
c3     ret
```

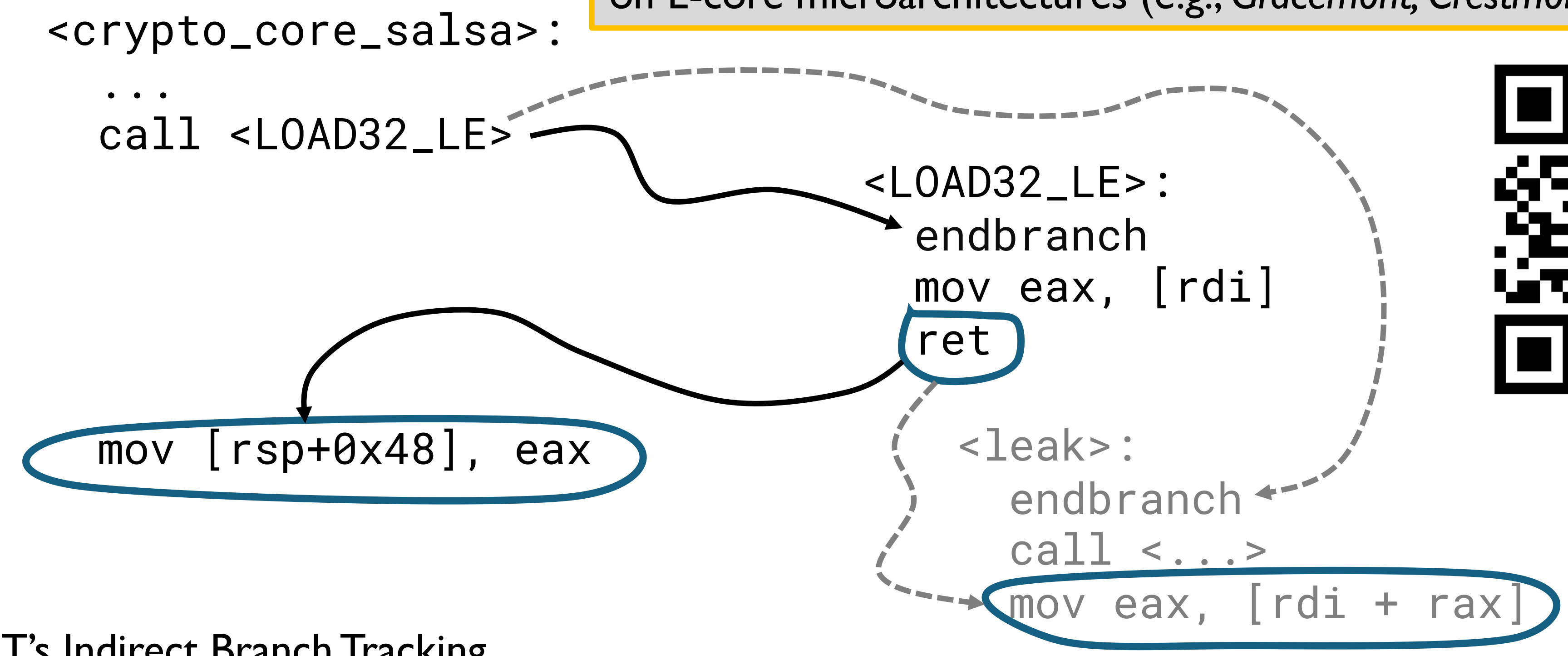
```
<leak>:
           endbranch
e8 92      call <...>
8b 04 07   mov eax, [rdi + rax]
```

- 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 transient control-flow

Constrain transient control-flow with **Intel Control-Flow Enforcement Technology (CET)**, which provides **transient CFI** on E-core microarchitectures (e.g., *Gracemont, Crestmont)*



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

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

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

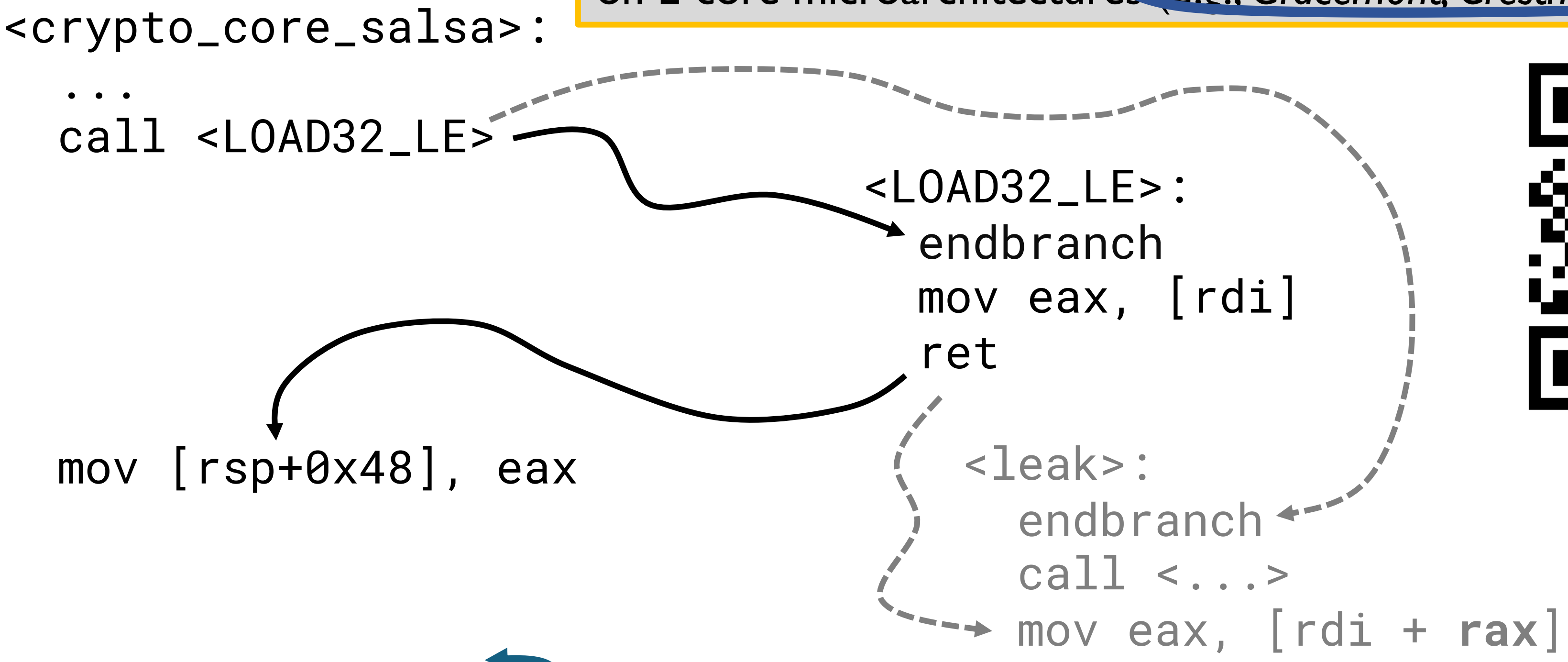
```
<leak>:
  endbranch
  call <...>
  mov eax, [rdi + rax]
```

- 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 transient control-flow

Constrain transient control-flow with **Intel Control-Flow Enforcement Technology (CET)**, which provides **transient CFI** on E-core microarchitectures (e.g., *Gracemont, Crestmont*)



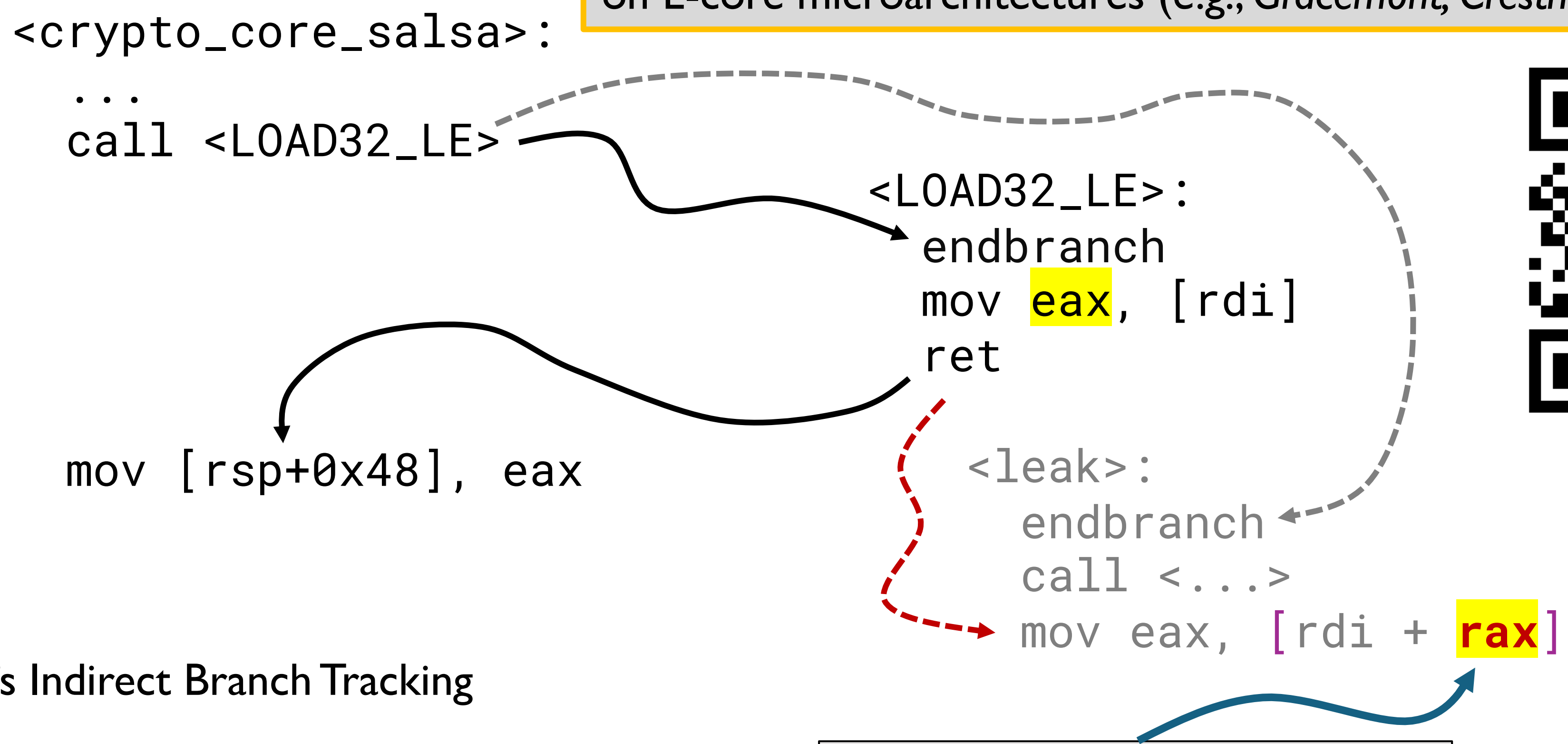
```
<crypto_core_salsa>:
  ...
  call <LOAD32_LE>
                             <LOAD32_LE>:
                               endbranch
                               mov eax, [rdi]
                               ret

  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 microarchitectures

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

Constrain transient control-flow with **Intel Control-Flow Enforcement Technology (CET)**, which provides **transient CFI** on E-core microarchitectures (e.g., *Gracemont, Crestmont)*

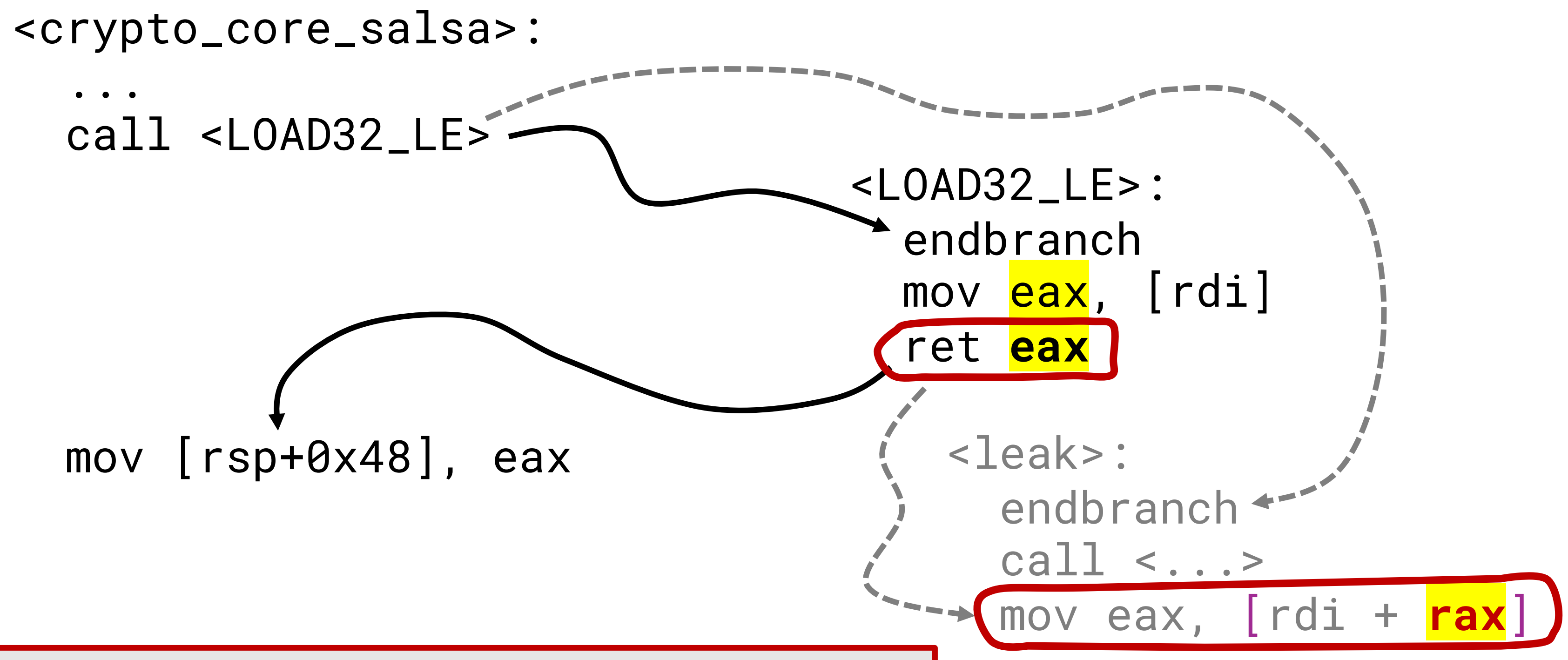


```
<crypto_core_salsa>:
  ...
  call <LOAD32_LE>

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

  mov [rsp+0x48], eax
                                  <leak>:
                                    endbranch
                                    call <...>
                                    mov eax, [rdi + rax]
```

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

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

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



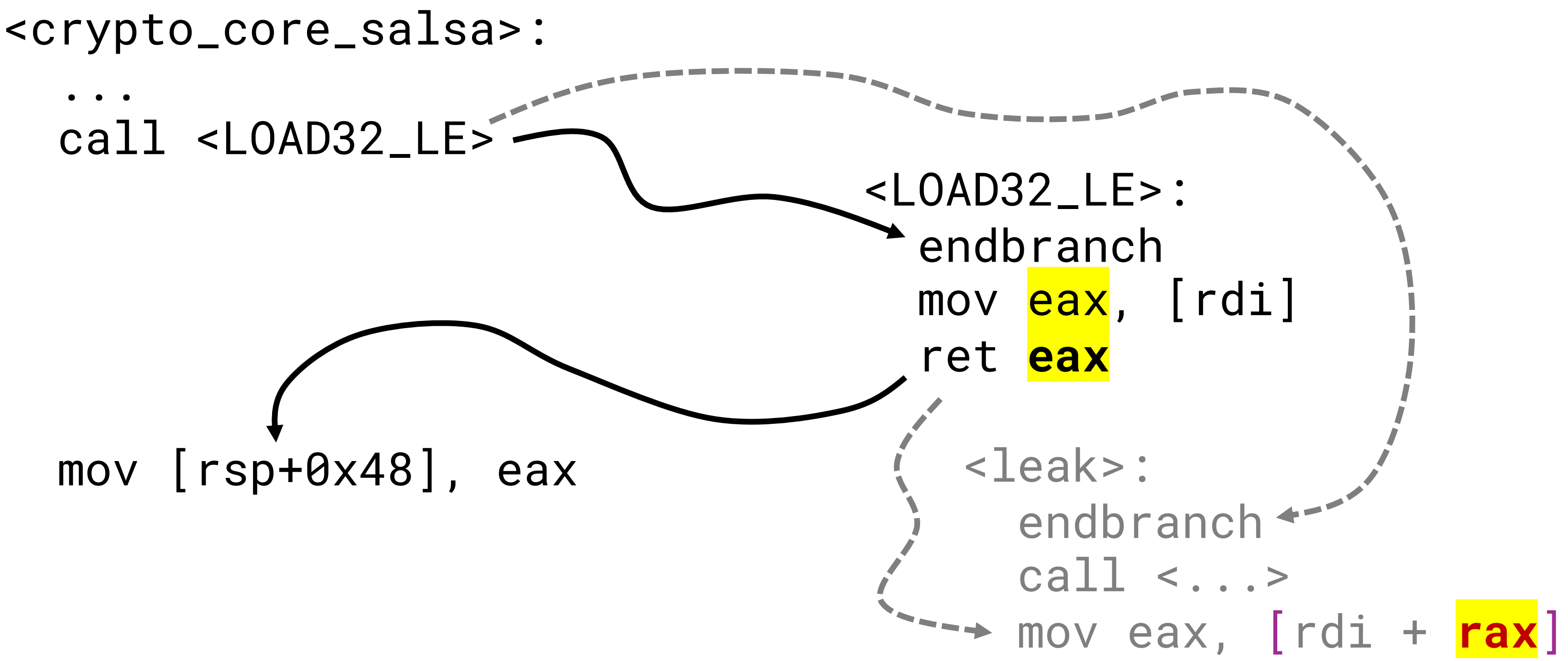
```
<crypto_core_salsa>:
  ...
  call <LOAD32_LE>

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

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

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

returning secrets by value is inherently vulnerable…

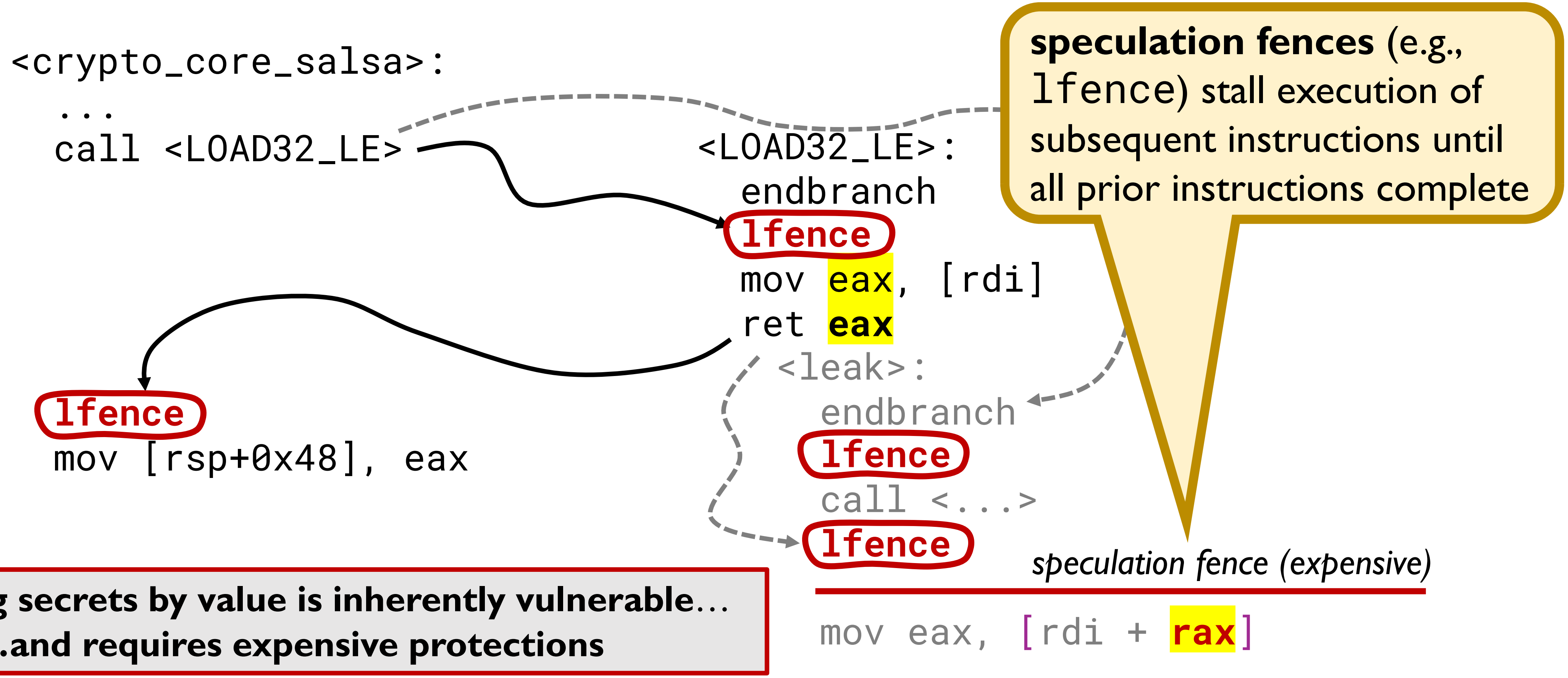
Challenge #2: Passing secrets by value is unsafe in the Spectre era



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

  mov [rsp+0x48], eax
                                       <leak>:
                                         endbranch
                                         call <...>
                                         mov eax, [rdi + rax]
```

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

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



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

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

```
<LOAD32_LE>:
  endbranch
  lfence
  mov eax, [rdi]
  ret eax
   <leak>:
     endbranch
     lfence
     call <...>
     lfence
```

speculation fence (expensive)

```
mov eax, [rdi + rax]
```

speculation fences (e.g., `lfence`) stall execution of subsequent instructions until all prior instructions complete

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:

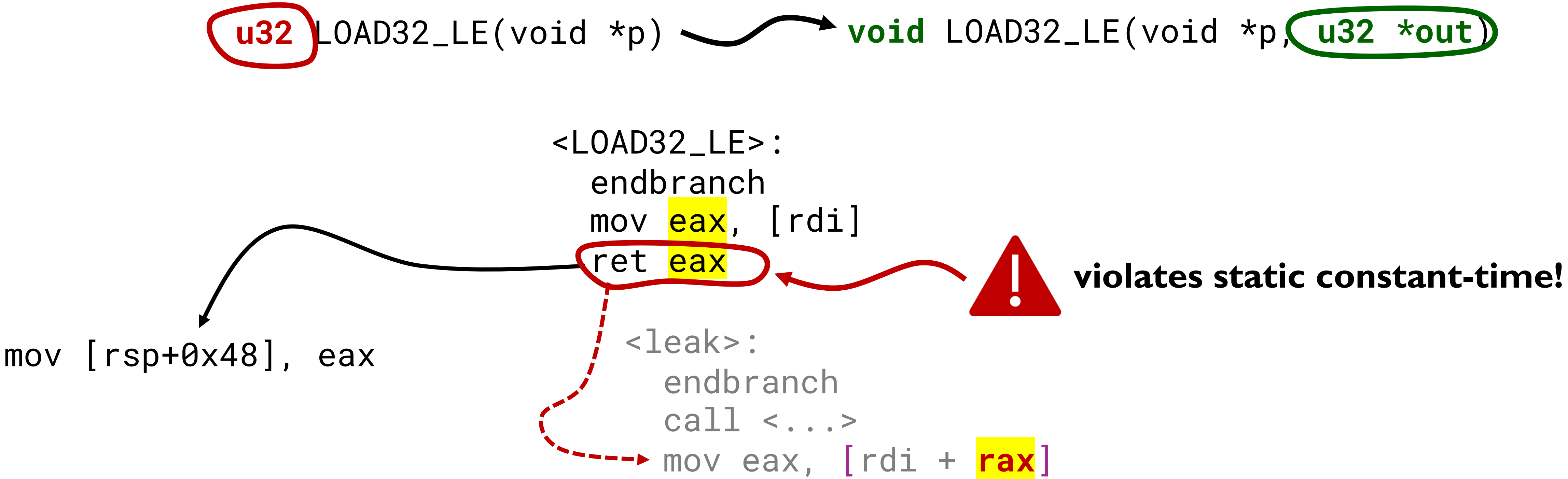
1 **implicit public/secret typing** of variables
(so that we can infer types at compile time without programmer annotations)

2 **pass** secret arguments by **reference**, not **value**
(so that we can scrub all secrets from registers on call/return)

Note: crypto routines generally satsify CTS out-of-the-box

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
```

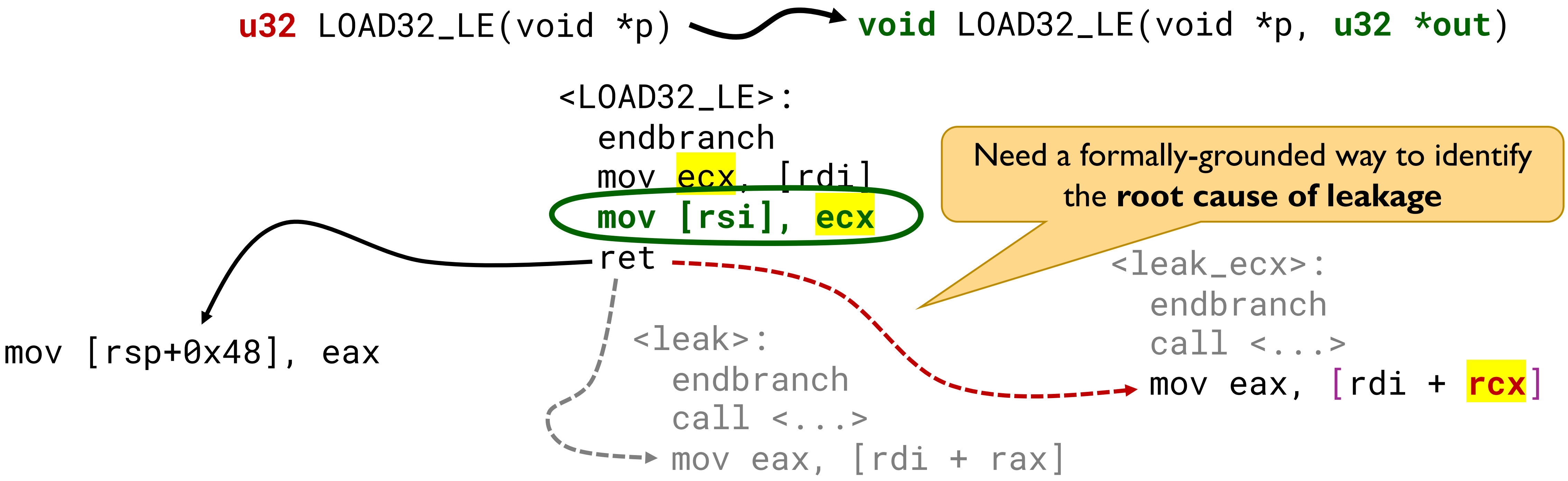
```
mov [rsp+0x48], eax
```

violates static constant-time!

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

Solution #2: Static Constant-Time Programming

2 **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
```

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

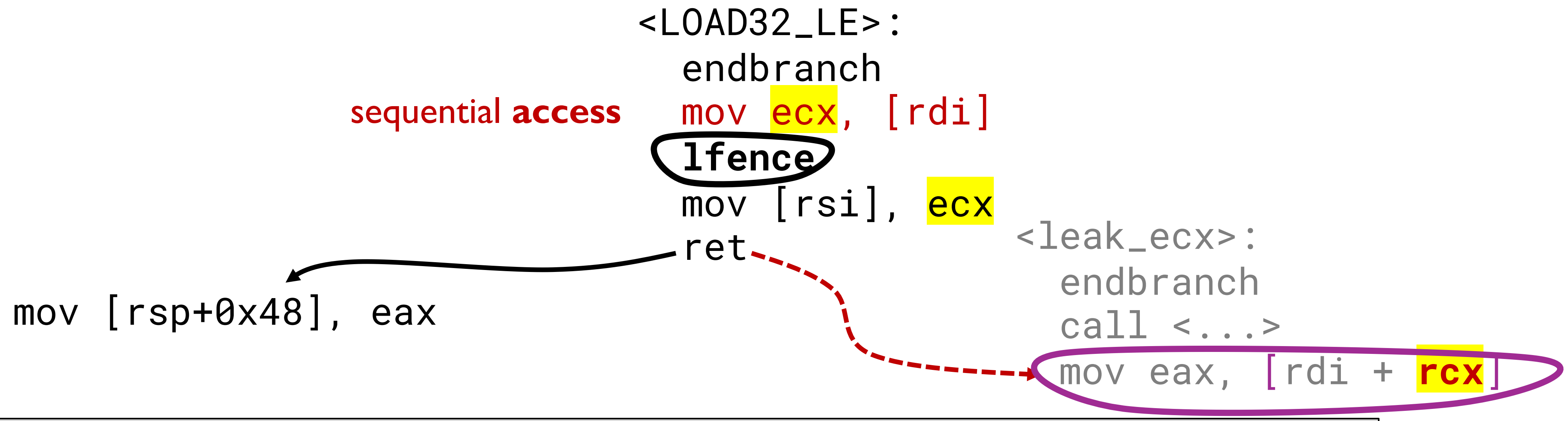
```
<leak>:
  endbranch
  call <...>
  mov eax, [rdi + rax]
```

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

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

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

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



sequential **access**

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

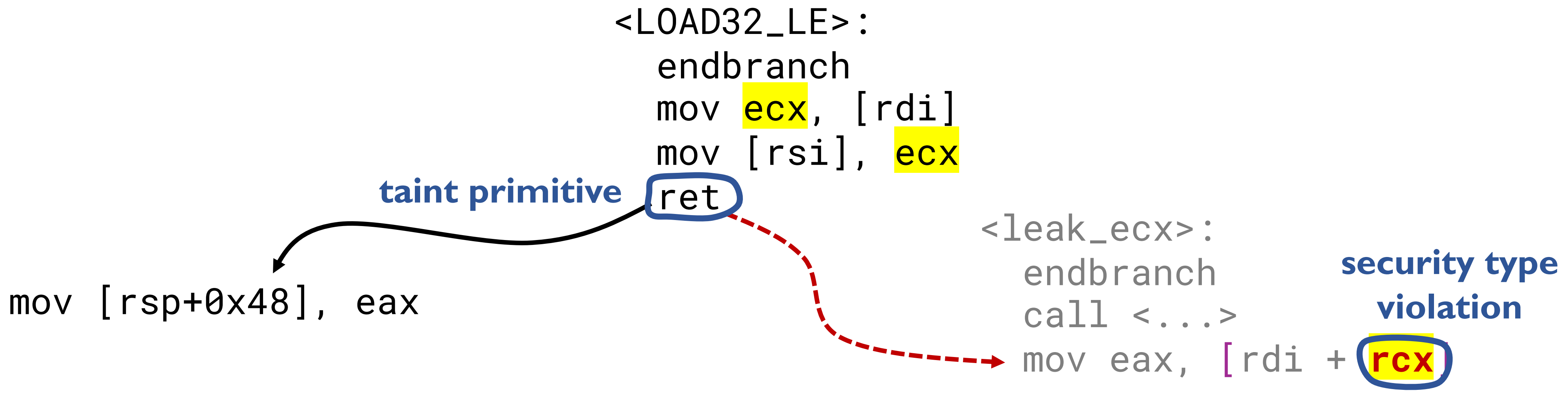
```
mov [rsp+0x48], eax
```

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

Inapplicable to software defenses: speculation fences do not always block sequentially accessed secrets from **transiently leaking**.

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.



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

taint primitive

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

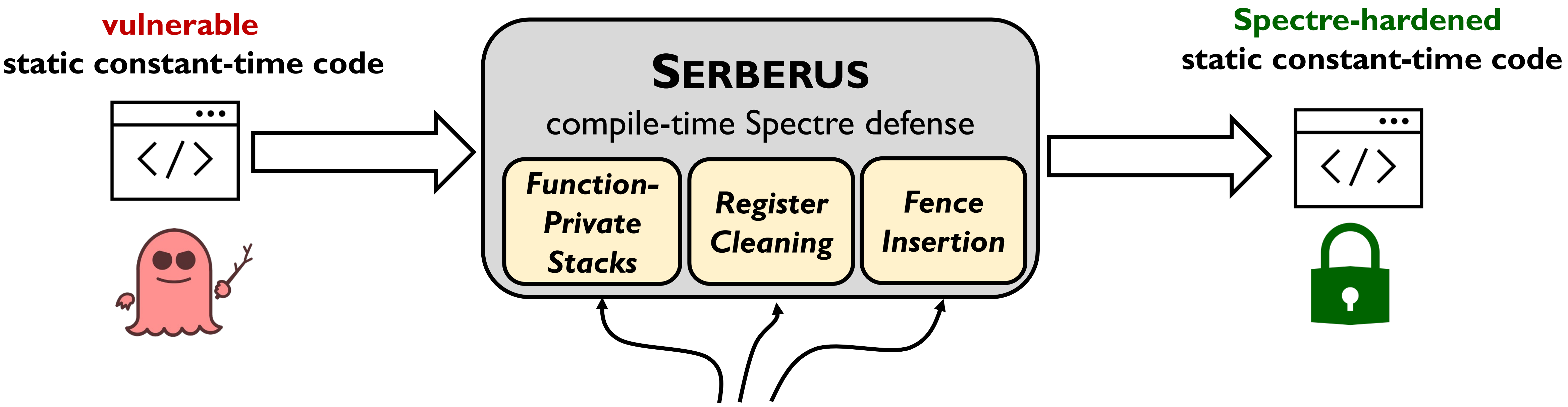
```
<leak_ecx>:
  endbranch
  call <...>
  mov eax, [rdi + rcx]
```

security type violation

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

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

*implemented for **LLVM***



<span style="color:red">**vulnerable**</span>
static constant-time code

SERBERUS

compile-time Spectre defense

Function-Private Stacks

Register Cleaning

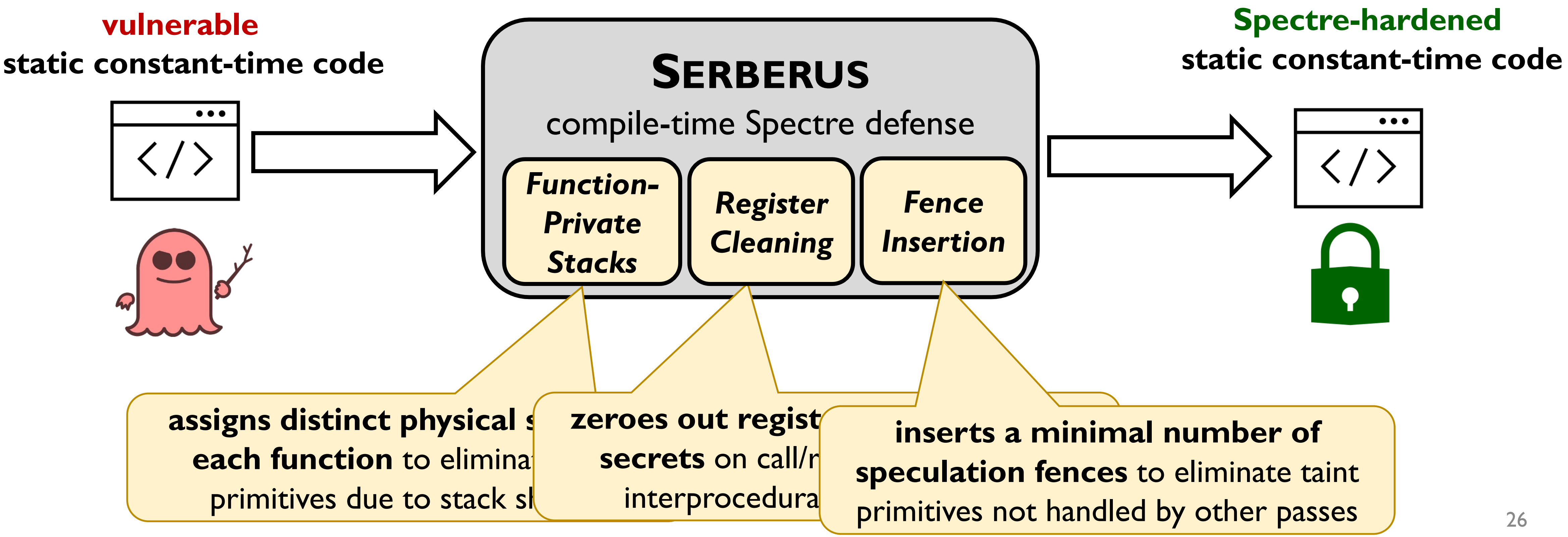
Fence Insertion

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

Register Cleaning

Fence Insertion

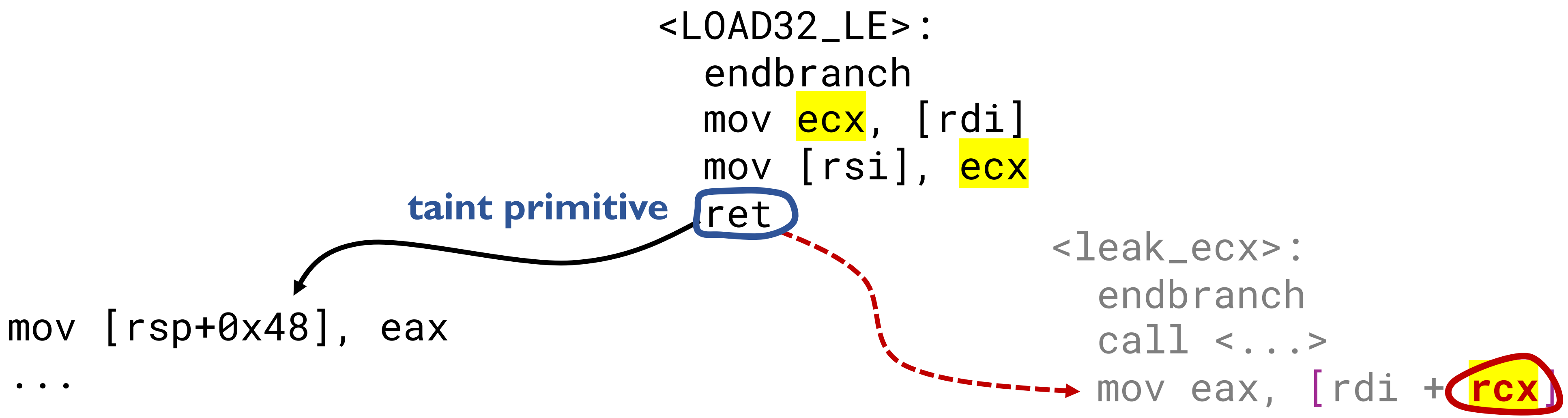
Spectre-hardened
static constant-time code

assigns distinct physical s
each function to elimina
primitives due to stack sh

zeroes out regist
secrets on call/r
interprocedura

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

SERBERUS' Register Cleaning Pass



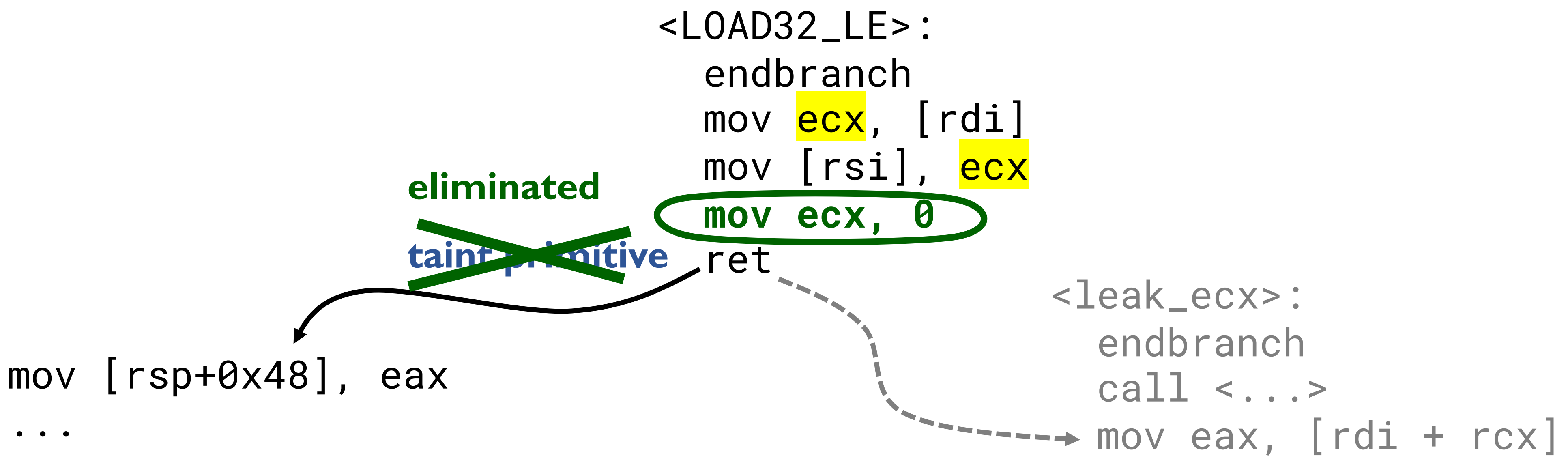
```
<LOAD32_LE>:
  endbranch
  mov ecx, [rdi]
  mov [rsi], ecx
  ret
```

taint primitive

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

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

SERBERUS' Register Cleaning Pass



```
<LOAD32_LE>:
  endbranch
  mov ecx, [rdi]
  mov [rsi], ecx
  mov ecx, 0
  ret
```

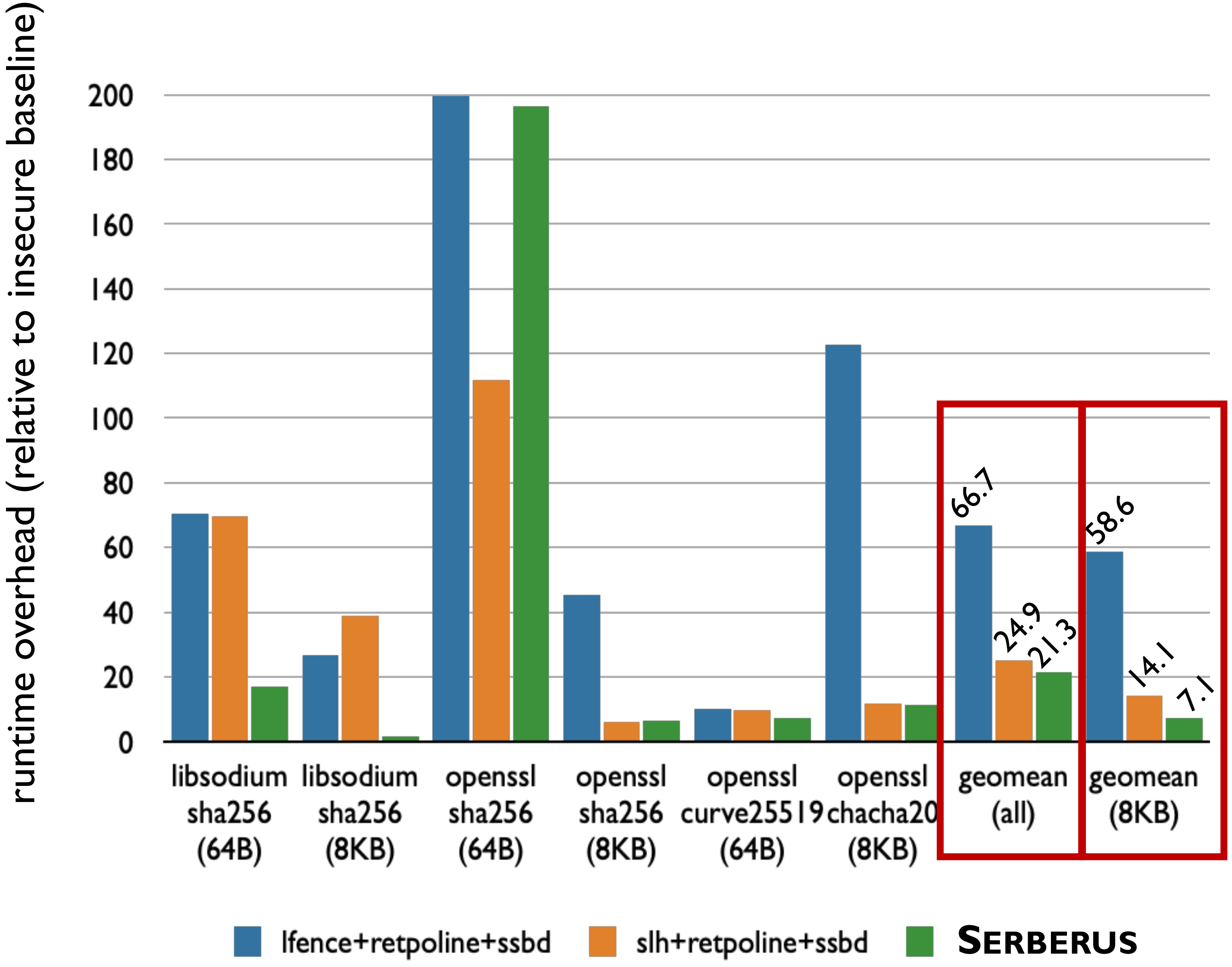
eliminated

~~taint primitive~~

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

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

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 with Existing Software Defenses
for Unannotated Constant-Time Code

Legend

⌫ disable speculation

● secure speculation

defense	**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]	●				
retpoline		⌫			
Switchpoline [Bauer+ AsiaCCS'24]		⌫			
IPRED_DIS		⌫			
SSBD					⌫
PSFD				⌫	⌫
Securest SOTA	●	⌫		⌫	⌫
SERBERUS (this paper)	●	●	●	●	⌫

LOAD32_LE's transient leakage

SERBERUS' Impact

- **for users**: offering the first way to securely run CTS crypto code on existing HW
- **for vendors**: prompting industry documentation of transient CFI semantics
- **for developers**: Spectre-aware programming guidelines
- **for researchers**: a new paradigm for identifying and eliminating transient leakage
- **for architects**: the first HW-SW codesign of its kind

Impact for **users**: only way to securely run CTS crypto code on existing HW

Past

No prior defense prevents secrets from transiently leaking via PHT/BTB/RSB/STL/PSF on existing hardware.

Serberus

- Serberus remains the **only comprehensive defense for all CTS crypto code** on **existing Intel E-cores**.
- Serberus' open-source LLVM fork secures any CTS code that LLVM can compile
- Widely cited as the new SOTA

Future

- **Serberus prompted Intel to signal** that **its "long-term direction"** is to enforce transient CFI on all future hardware.[1]
- Serberus will become **more widely deployable over time**.
- In progress: **upstreaming Serberus into LLVM**.

deployable defenses targeting CT	**PHT** conditional branch pred	**BTB** indirect branch pred	**RSB** return prediction	**STL** store-to-load fwding pred	**PSF** predictive store fwding	**Overhead** (8KB benchmarks)
Prior work	●	⌫		⌫	⌫	**14.1%**

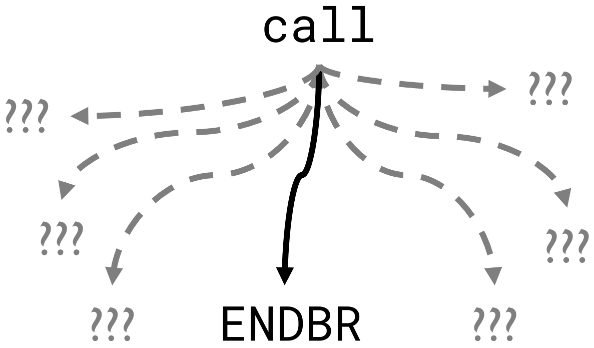
[1] https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/hardware-behavior-related-to-speculative-execution.html#intel-cet

Impact for **vendors**: standardization of transient CFI semantics

Past → SERBERUS → Future

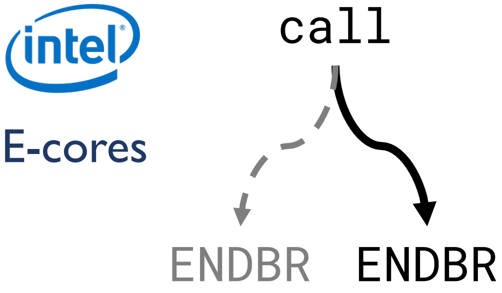
Past

- No transient semantics for CFI on any major ISA/CPU



SERBERUS

- Spurred Intel to publish the transient semantics of Intel CET on existing platforms



Future

- To our knowledge, has spurred discussions about how to retrofit Arm for speculative CT programming
- Possible domino effect, leading to standardization for RISC-V too
- Future defenses may build on Intel's strong transient CFI semantics

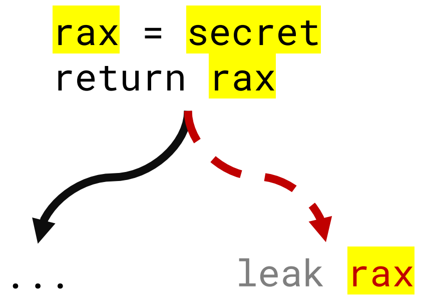
	Intel CET	**Arm**	**RISC-V**
Pre-SERBERUS	unknown	unknown	unknown

[1] https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/hardware-behavior-related-to-speculative-execution.html#intel-cet

Impact for **developers**: Spectre-aware programming guidelines

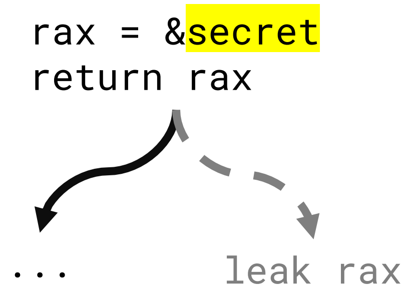
Past
Serberus
Future

Traditional constant-time programming permits inherently dangerous programming patterns



```
rax = secret
return rax

...          leak rax
```

Static constant-time (CTS) programming defines concrete, easy-to-follow guidelines for crypto programming in the Spectre era



```
rax = &secret
return rax

...          leak rax
```

- Many routines in widely used crypto libraries (OpenSSL, libsodium, HACL*) already uphold CTS

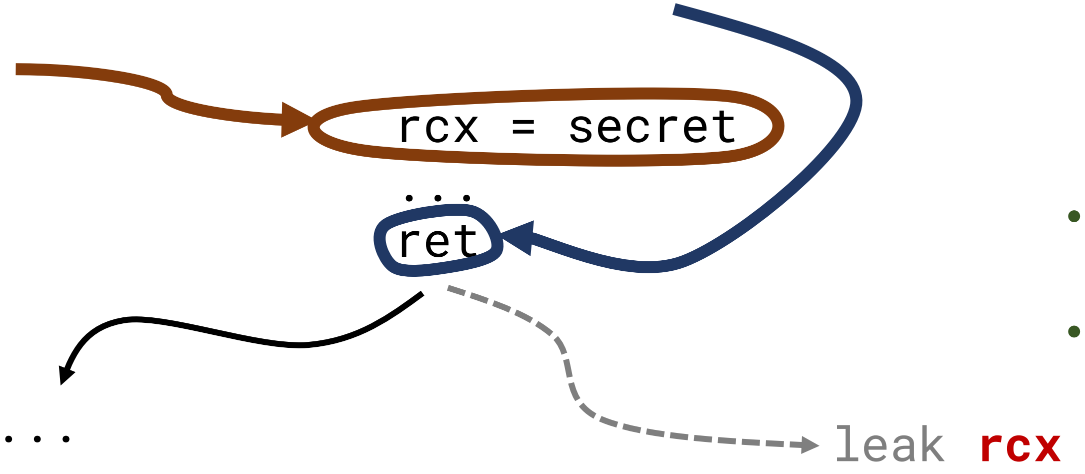
Future adoption of CTS principles will reduce Spectre attack surface, even if no defense is enabled

Impact for **researchers**: new way to identify and eliminate transient leakage of secrets

Past → Serberus → Future

Access instructions do not **root-cause transient leakage** in CTS code.

Taint primitives precisely root-cause transient leakage in CTS code.



- Researchers have already suggested **combining access-based defenses with Serberus' taint-primitive-based defense** to extend their security guarantees to CTS code *[Schmitz+ AsiaCCS'25]*
- Unpublished work **generalizes taint primitives to all code**
- Unpublished work efficiently **recognizes taint primitives at runtime in HW**

Impact for **architects**: a new kind of HW-SW codesign

Past
SERBERUS
Future

While performant, prior HW-SW codesigns require **invasive HW modifications** to track what data may be secret.

- **Heavy lifting in HW, not SW**
- **High HW complexity**
- **Unknown viability**: not implemented in existing HW; no sign of adoption in future HW

SERBERUS shows how **low-cost transient control-flow and data-flow restrictions** in HW can enable efficient compile-time defenses in SW.

- **Offloads heavy lifting from HW to SW**
- **Low HW complexity**
- **Viable**: sufficient constraints implemented in some existing HW

Motivates architects to implement **more low-cost restrictions** in future HW to enable even **more efficient SW defenses**.

Takeaways

- **SERBERUS is the first comprehensive, deployable defense** for protecting constant-time code against Spectre leakage on existing HW.
- SERBERUS has prompted **industry to document transient CFI semantics**.
- SERBERUS can be extended to…
  - **handle new speculation primitives** (e.g., SLS)
  - **run on new microarchitectures** (e.g., Intel P-cores or Arm)
  - **complement other defenses** (e.g., sandboxing defenses [Schmitz+ AsiaCCS'25])
- SERBERUS provides…
  - **crypto developers** with new guidelines for hardening their code against Spectre.
  - **researchers** with a new technique for root-causing and eliminating Spectre leakage.
  - **architects** with low-cost hardware changes to enable performant software Spectre defenses.

GitHub: https://github.com/nmosier/serberus
Email: nmosier@stanford.edu