Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing

Sangho Lee Ming-Wei Shih Prasun Gera
Taesoo Kim Hyesoon Kim Marcus Peinado

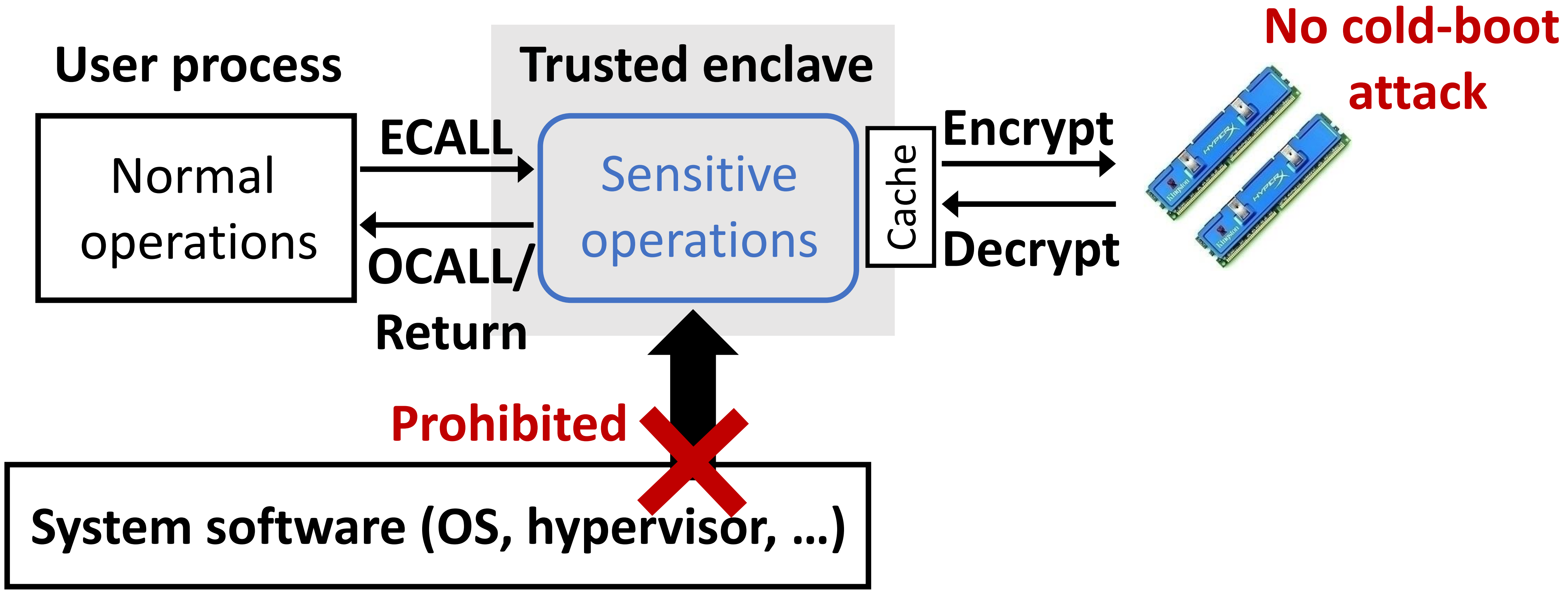
26th USENIX Security Symposium

August 17, 2017

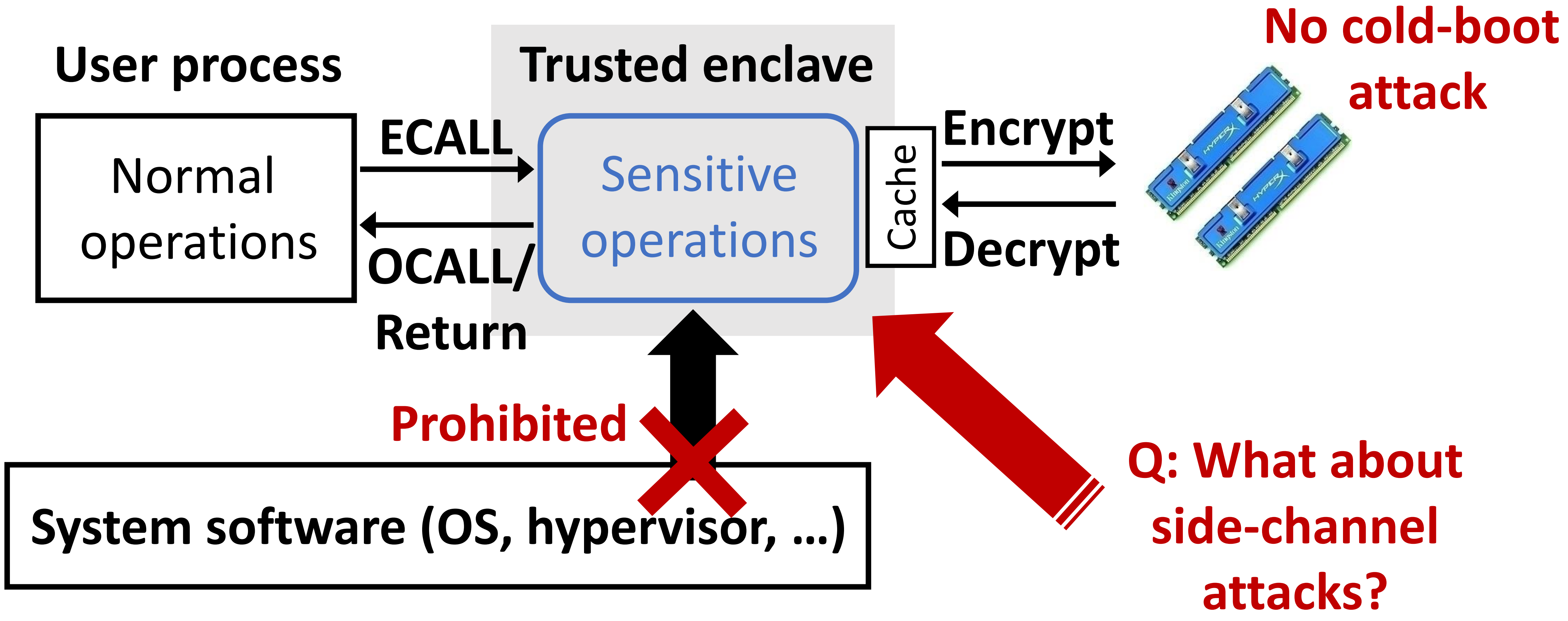
Georgia Institute of Technology

Microsoft Research

Intel Software Guard Extension (SGX)



Intel Software Guard Extension (SGX)



Side-channel attacks against Intel SGX are getting attention

Monitor page-fault or page-access sequence
(Oakland15, ASIACCS16, Security17)

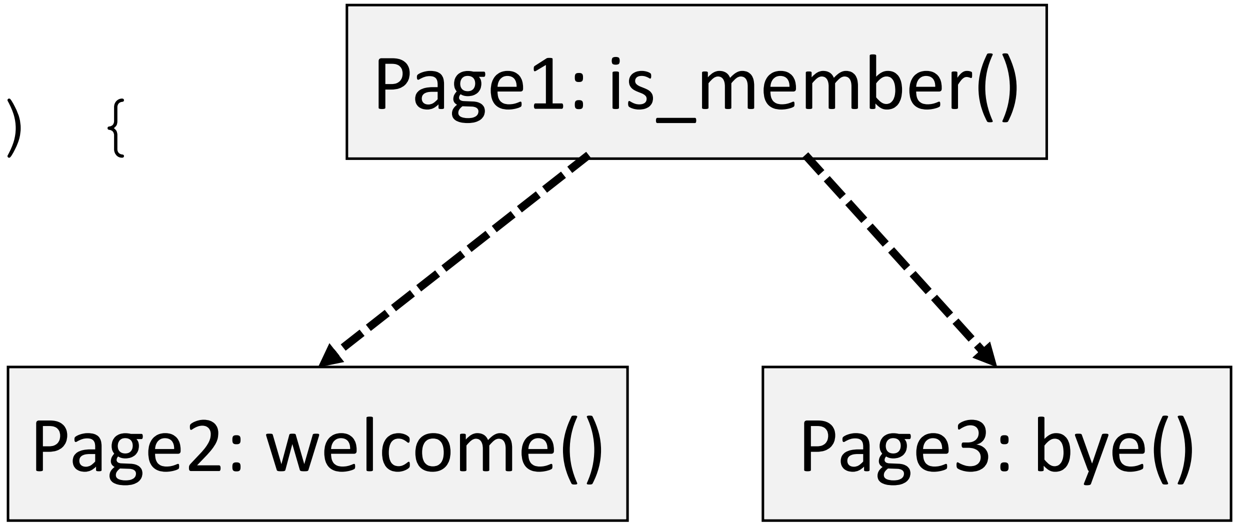
- Noise-free, but coarse-grained (page address)

Measure cache hit/miss timing
(EuroSec17, DIMVA17, ATC17, WOOT17)

- Fine-grained (cache line), but noisy

Page-fault side channel (Oakland15)

```
if (is_member(person)) {
   welcome();
} else {
   bye();
}
```

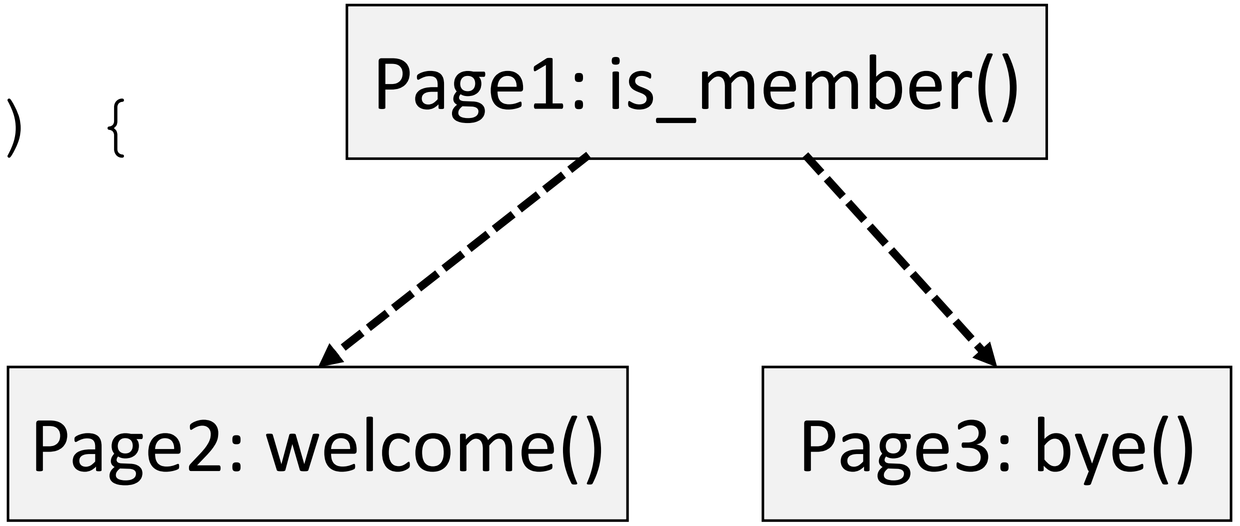


Unmap all pages and monitor page fault sequences

- Page 1->Page 2: A member
- Page 1->Page 3: Not a member

Page-fault side channel (Oakland15)

```
if (is_member(person)) {
    welcome();
} else {
    bye();
}
```



Does not work when a sensitive control flow change occurs within the same page (or cache line)

Branch shadowing: A fine-grained side-channel attack against Intel SGX

- Can attack each branch instruction
  - Neither page nor cache-line granularity
- Deterministically identify branch history
  - Either taken or not taken
  - Not about timing difference
- Achieve high attack success rate
  - Recover 66% of a 1024-bit RSA private key from a single run

Observation:
SGX does not clear branch history!

CPU caches how each branch instruction has been executed for later prediction, even for SGX.

- Either **taken** or **not taken**, as well as its **target address**

Does an attacker have a reliable way to extract branch history from SGX?

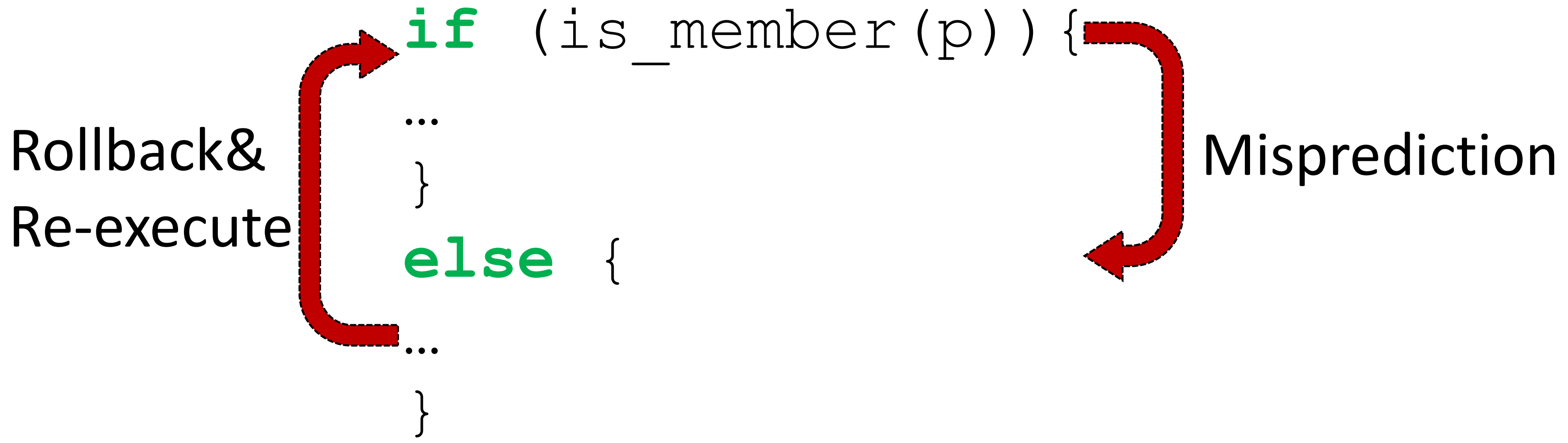
Performance monitoring unit (PMU) is prohibited

- PMUs to profile branch history
  - Last branch record (LBR) and processor trace (PT)
  - Prediction results (success/failure), target address, …
- Anti side channel inference (ASCI)
  - SGX doesn't publish hardware performance events to PMUs.
- **Malicious OS cannot directly use PMUs to get SGX's branch history.**

Branch collision timing attack works for SGX but has limitations

Mispredicted branch takes longer than a correctly predicted branch.

- But, we cannot directly time a target branch inside SGX.

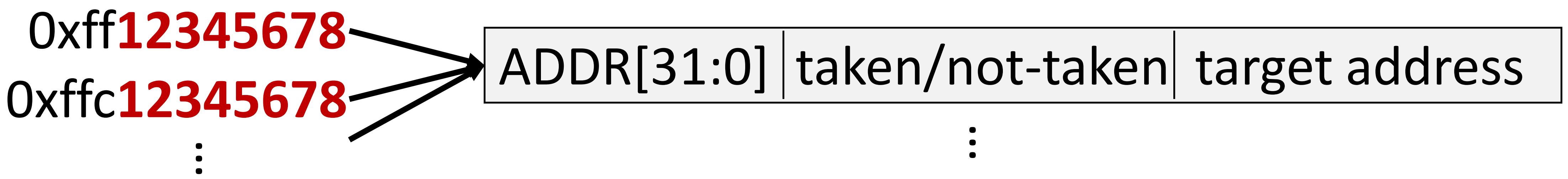


```
if (is_member(p)){
…
}
else {
…
}
```

Branch collision timing attack works for SGX but has limitations

Colliding branches affect each other's prediction (MICRO16).

- e.g., if a branch has been taken, CPU will predict other colliding branches will also be taken.



Branch instructions with colliding addresses
(CPU truncates higher bits to reduce storage overhead.)

Branch collision timing attack works for SGX but has limitations

Branch execution inside SGX affects colliding branches outside of SGX (***shadow branch***).

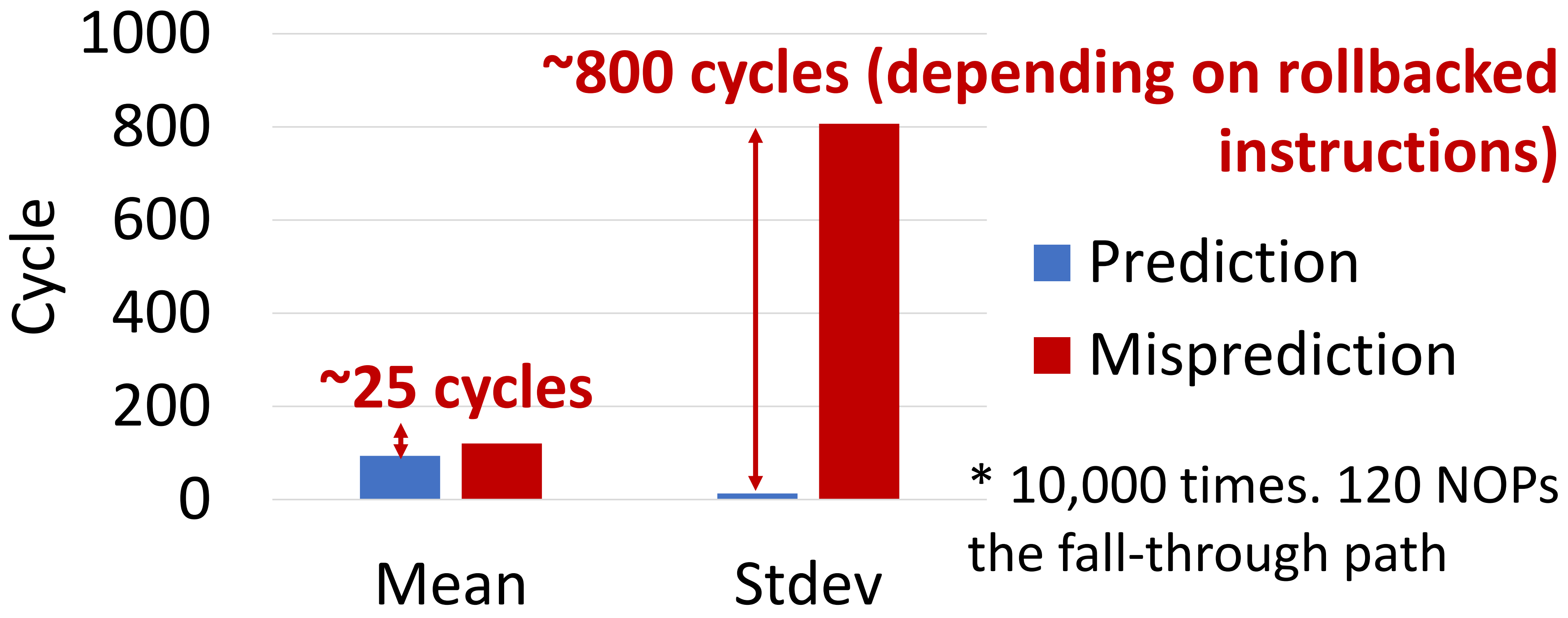
- We can time a shadow branch instead of the actual target to know whether it has been mispredicted, but…

This attack has two critical limitations.

- Suffer from high measurement noise
- Difficult to synchronize target and shadow branches

Limitation 1: High measurement noise

Mispredicted branch takes long to do rollback while suffering from **high variance**.



* 10,000 times. 120 NOPs at the fall-through path

Limitation 2:
Difficulty in synchronization

We need to time a shadow branch **right after** a target has been executed to avoid overwriting.

- e.g., Skylake's branch target buffer: 4 ways x 1,024 sets
- Worst case: Five branch executions would overwrite the target branch history.

Synchronization is difficult because SGX does not allow **single-stepping**.

How does branch shadowing overcome the two limitations?

Apply LBR to a shadow branch to identify branch prediction results instead of timing

- No ASCI because a shadow branch is outside of SGX
- Deterministic: Either correctly predicted or mispredicted

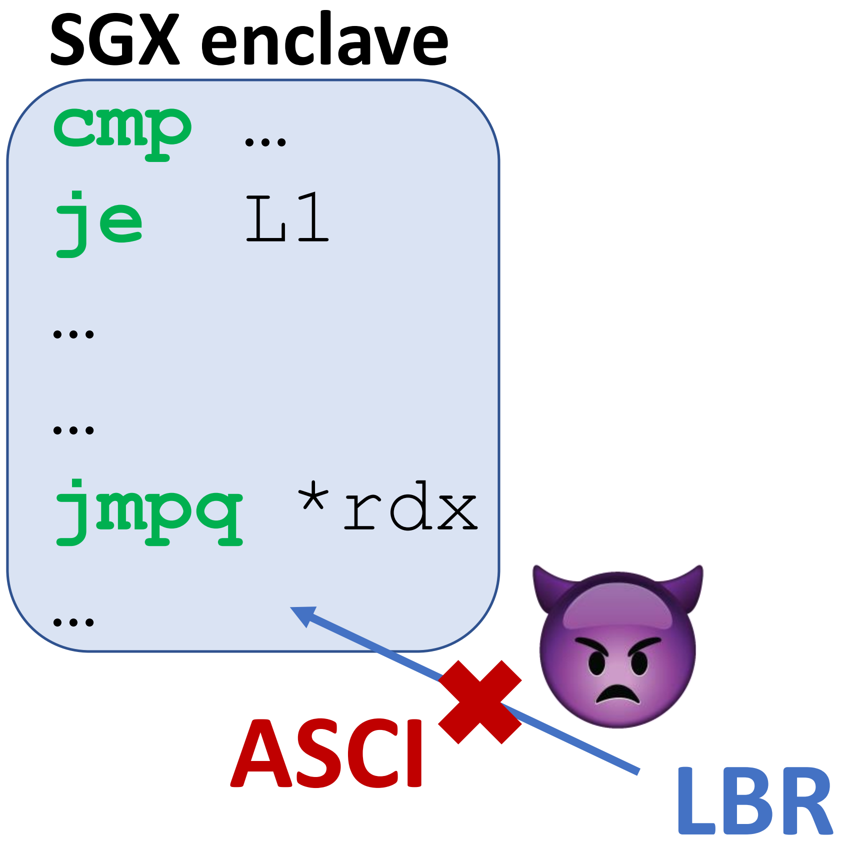
Realize near single-stepping by increasing timer interrupt frequency and disabling the cache

- Can interrupt SGX enclaves for every ~5 cycles

Threat model

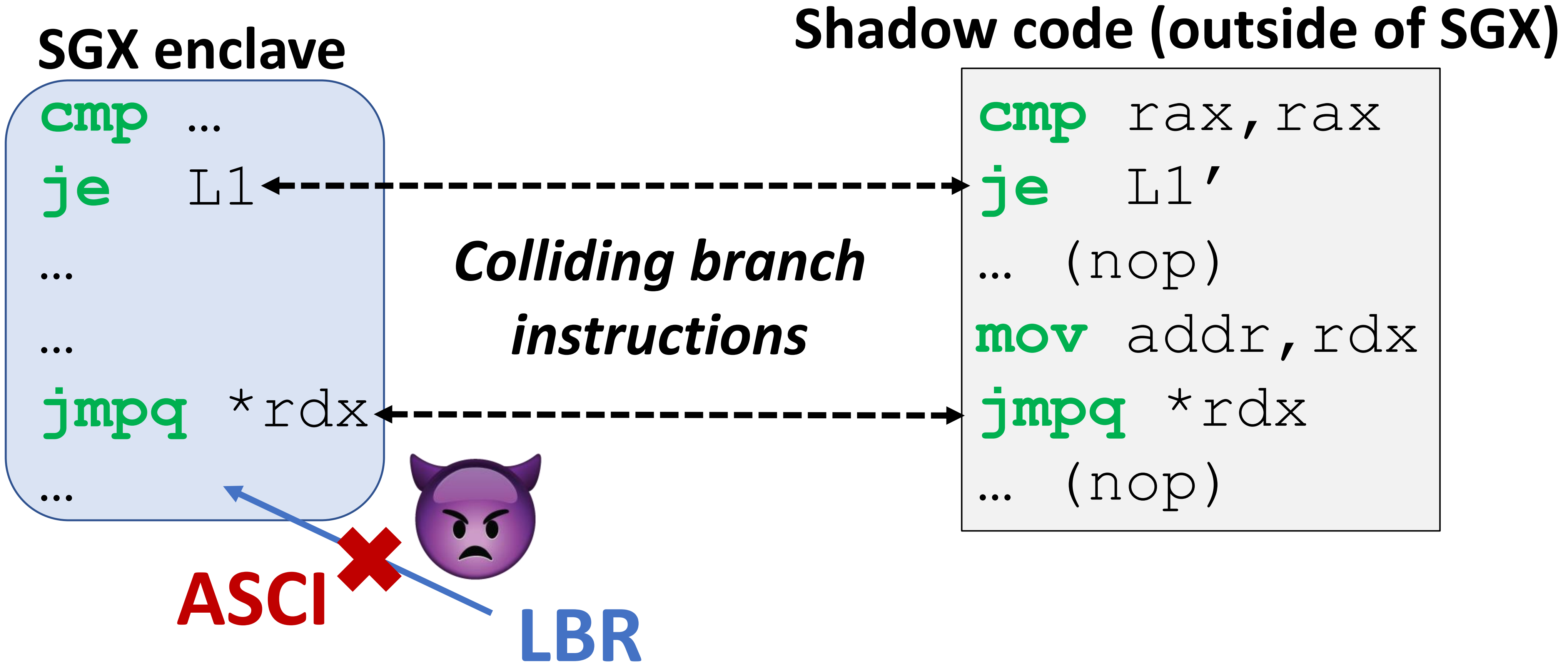
- Attacker knows the source code or binary of a target enclave.
- Attacker can frequently interrupt the target enclave’s execution to execute attack code.
- Attacker prevents or disrupts the target enclave from accessing a trusted time source.

Step 1: Prepare a shadow copy of an SGX program to monitor it with LBR



```
cmp  …
je   L1
…
…
jmpq  *rdx
…
```

Step 1: Prepare a shadow copy of an SGX program to monitor it with LBR



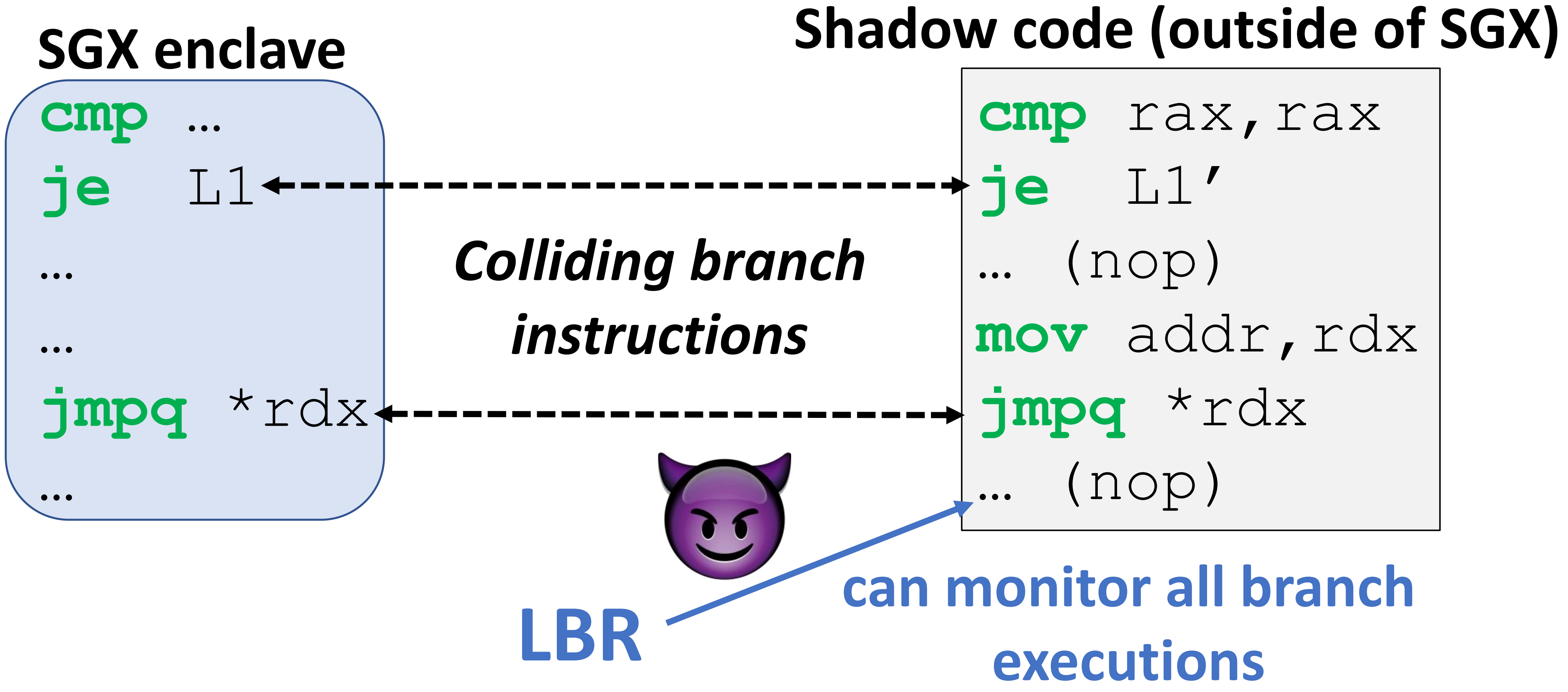
SGX enclave

```
cmp  …
je   L1
…
…
jmpq  *rdx
…
```

Shadow code (outside of SGX)

```
cmp  rax,rax
je   L1'
…  (nop)
mov  addr,rdx
jmpq  *rdx
…  (nop)
```

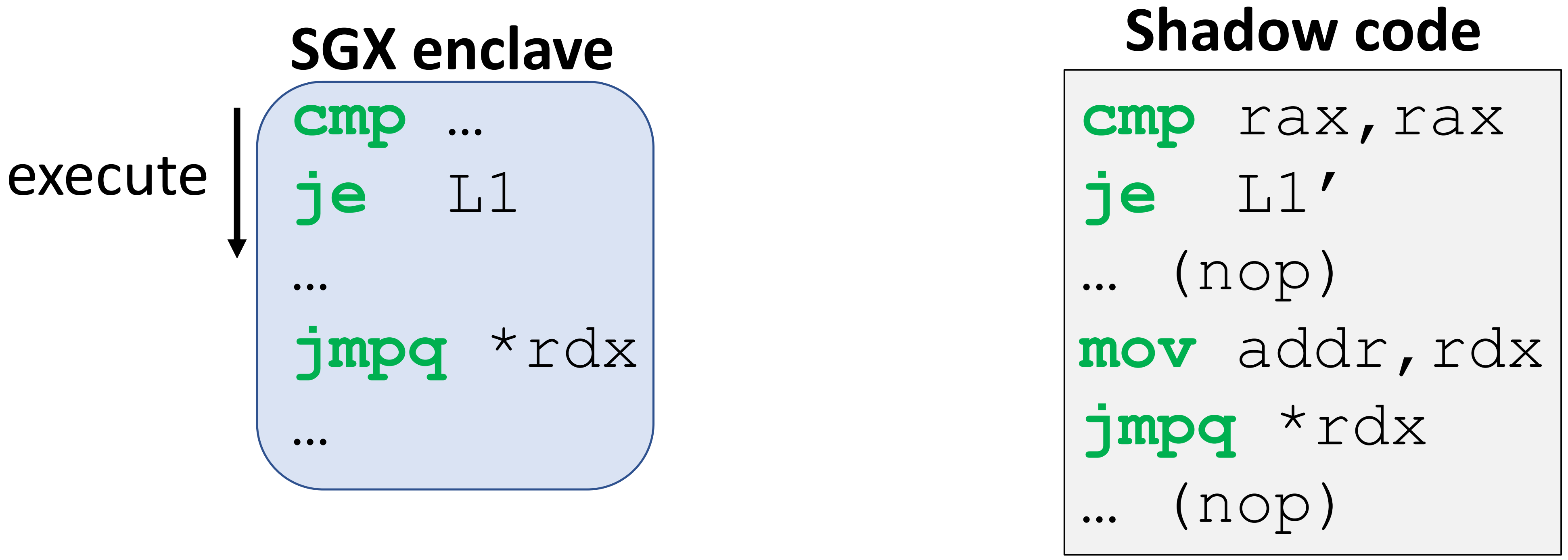
Step 1: Prepare a shadow copy of an SGX program to monitor it with LBR


SGX enclave
Shadow code (outside of SGX)
Colliding branch instructions
LBR
can monitor all branch executions

```
cmp  …
je   L1
…
…
jmpq *rdx
…
```

```
cmp  rax,rax
je   L1'
… (nop)
mov  addr,rdx
jmpq *rdx
… (nop)
```

Step 2: Interrupt SGX execution and monitor shadow code with LBR



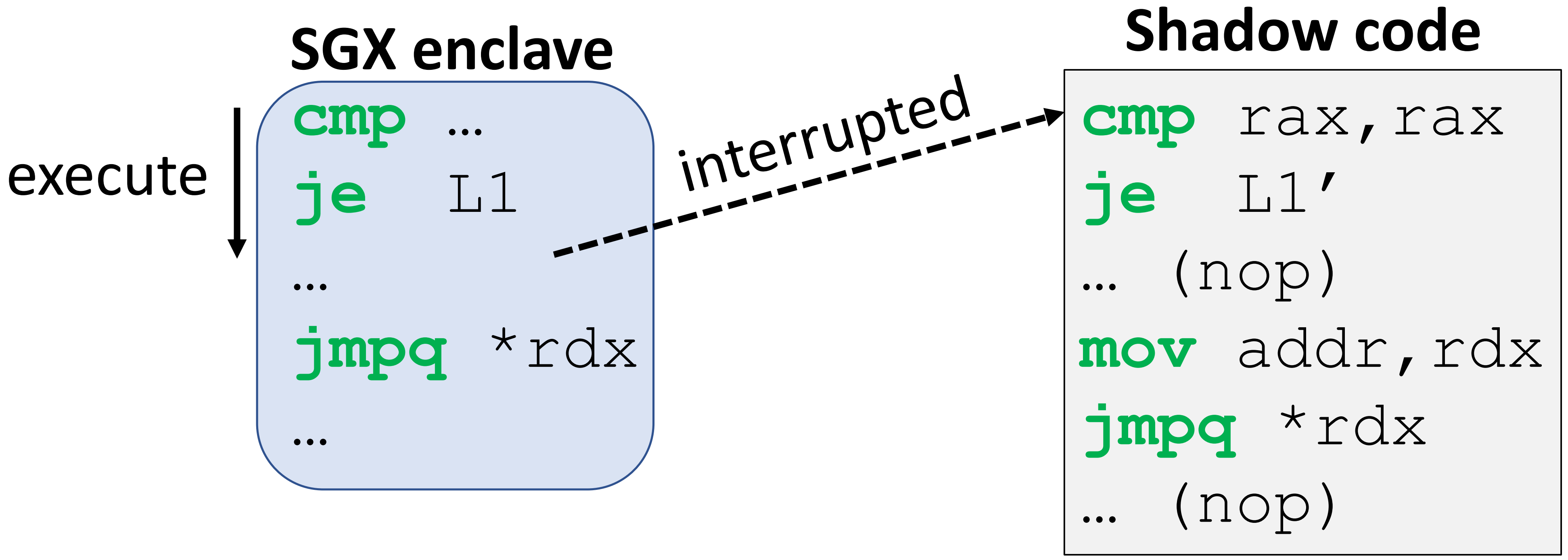
SGX enclave

```
cmp   …
je    L1
…
jmpq  *rdx
…
```

Shadow code

```
cmp  rax,rax
je   L1'
… (nop)
mov  addr,rdx
jmpq  *rdx
… (nop)
```

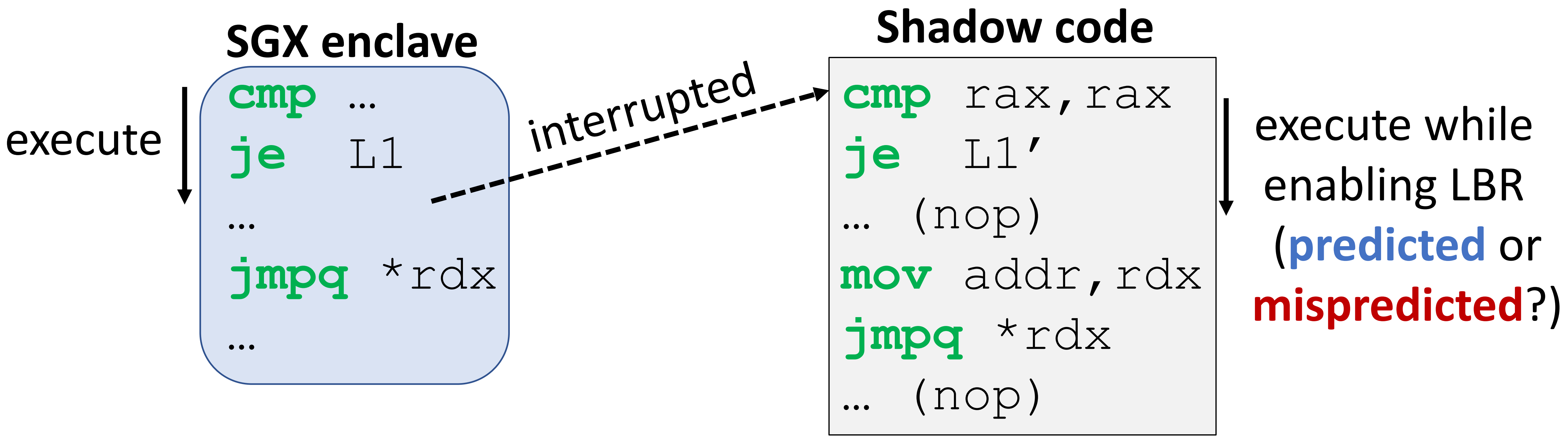
Step 2: Interrupt SGX execution and monitor shadow code with LBR



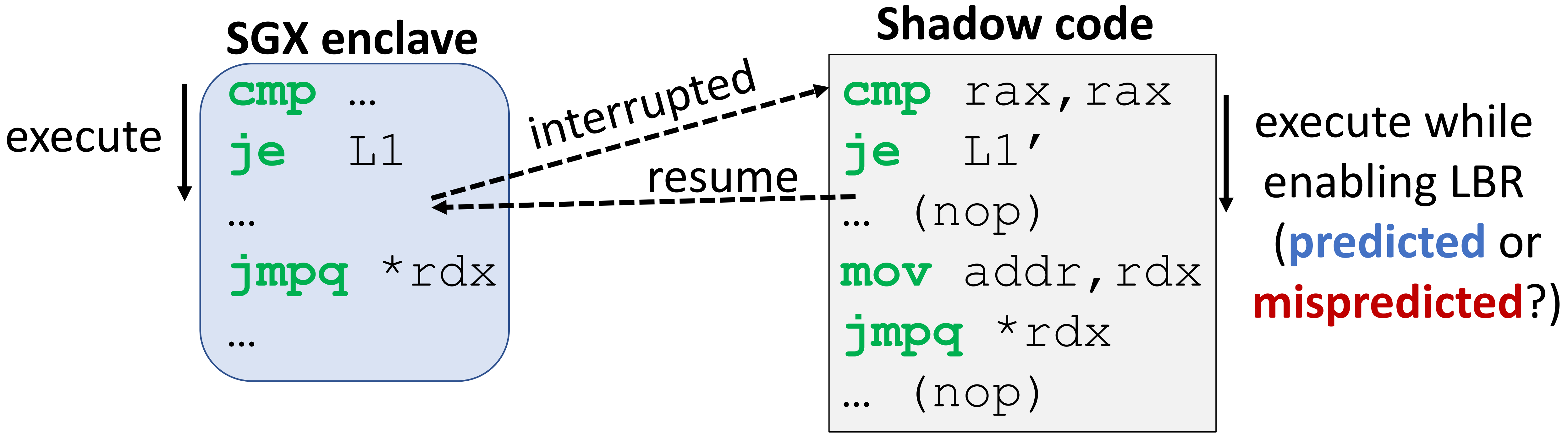
```
cmp   …
je    L1
…
jmpq  *rdx
…
```

```
cmp   rax,rax
je    L1'
…  (nop)
mov   addr,rdx
jmpq  *rdx
…  (nop)
```

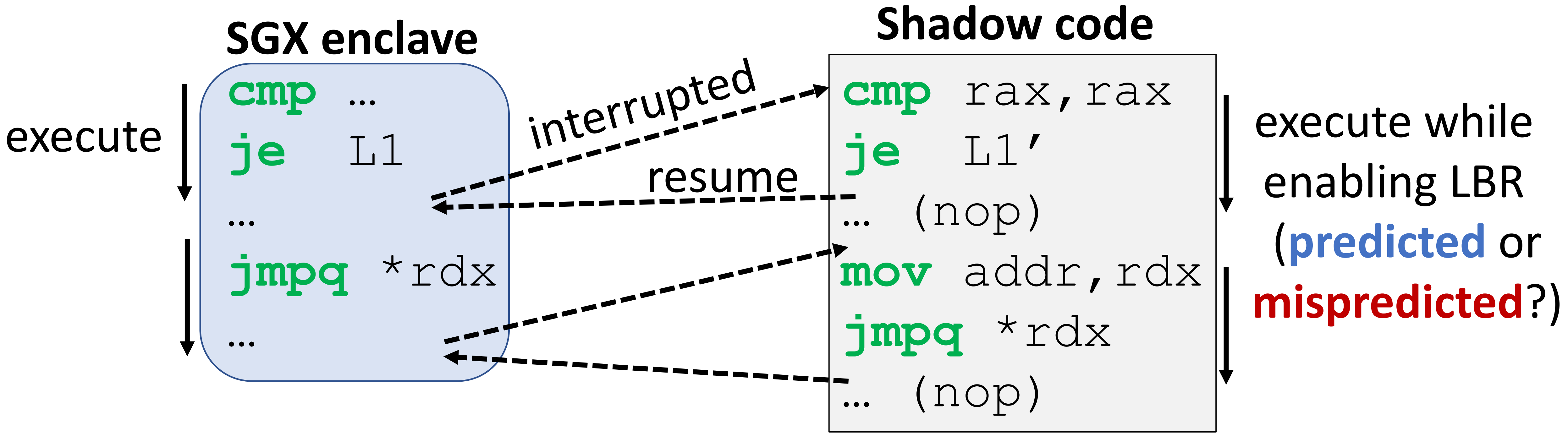
Step 2: Interrupt SGX execution and monitor shadow code with LBR



Step 2: Interrupt SGX execution and monitor shadow code with LBR

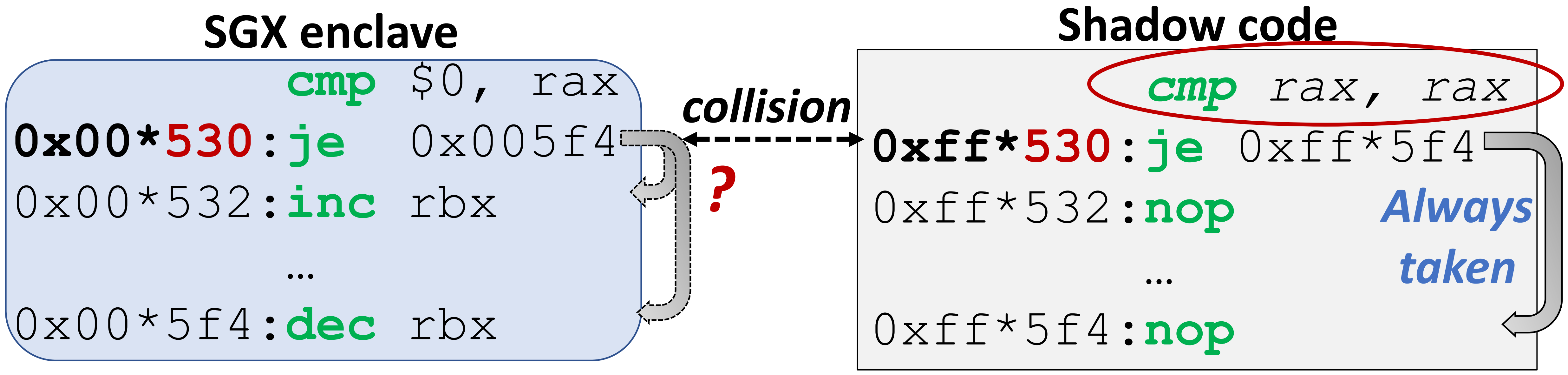


Step 2: Interrupt SGX execution and monitor shadow code with LBR



Whether or not shadow branches were correctly predicted reveals the history of target branches.

Shadow conditional branch and prediction result



```
                cmp $0, rax
0x00*530:je  0x005f4
0x00*532:inc rbx
            …
0x00*5f4:dec rbx
```

```
                cmp rax, rax
0xff*530:je 0xff*5f4
0xff*532:nop
            …
0xff*5f4:nop
```

LBR does not report not-taken branches, so we make our shadow branch be **always taken**.

Shadow conditional branch and prediction result

- Our shadow branch should be taken, but how does CPU predict it with target branch's history?
- If the target branch has been **taken**
  - ➢LBR: The shadow branch has been **correctly predicted**.
- If the target branch has been **not taken**
  - ➢LBR: The shadow branch has been **mispredicted**.

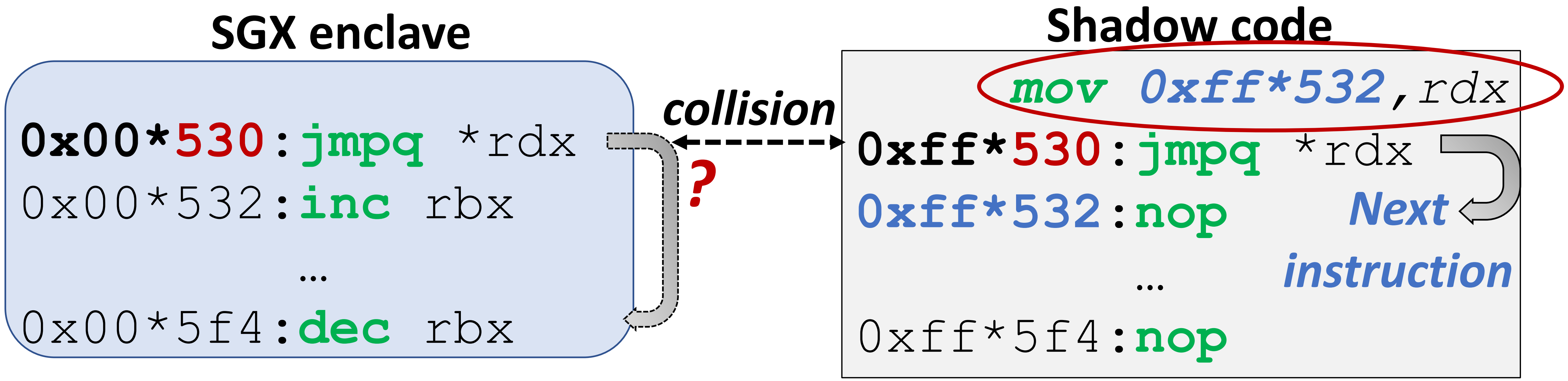
Shadow conditional branch and prediction result

- Our shadow branch should be taken, but how does CPU predict it with target branch's history?

Deterministically identify whether a target conditional branch has been taken or not taken

➢LBR: The shadow branch has been **mispredicted**.

Shadow indirect branch and prediction result



```
0x00*530:jmpq *rdx
0x00*532:inc rbx
        …
0x00*5f4:dec rbx
```

```
     mov 0xff*532,rdx
0xff*530:jmpq *rdx
0xff*532:nop
        …
0xff*5f4:nop
```

For an indirect branch, LBR reports a target prediction result.

We use its default target: **Next instruction**.

Shadow indirect branch and prediction result

- Our shadow branch will be correctly predicted unless the target branch updates cached destination.
- If the target branch has been **executed**
  - ➢LBR: The shadow branch has been **mispredicted**.
- If the target branch has been **not executed**
  - ➢LBR: The shadow branch has been **correctly predicted**.

Shadow indirect branch and prediction result

- Our shadow branch will be correctly predicted unless the target branch updates cached

Deterministically identify whether a target indirect branch has been executed or not

- If the target branch has been **not executed**
  - ➢LBR: The shadow branch has been **correctly predicted**.

Near single-stepping: Frequent timer and disabled cache

Increase timer interrupt frequency

- Adjust the timestamp counter value of the local APIC timer using a model-specific register, MSR_IA32_TSC_DEADLINE

Disable the CPU cache

- CD bit of the CR0 register (code?)

Near single-stepping: Frequent timer and disabled cache

Increase timer interrupt frequency **~50 cycles**

- Adjust the timestamp counter value of the local APIC timer using a model-specific register, MSR_IA32_TSC_DEADLINE

Disable the CPU cache

- CD bit of the CR0 register (code?)

Near single-stepping: Frequent timer and disabled cache

Increase timer interrupt frequency

- Adjust the timestamp counter value of the local APIC timer using a model-specific register, MSR_IA32_TSC_DEADLINE

Disable the CPU cache

~5 cycles

- CD bit of the CR0 register (code?)

Attack evaluation: Sliding-window exponentiation

```
/* X = A^E mod N */
mbedtls_mpi_exp_mod(X, A, E, N, _RR) {
  …
  while (1) {
    // i-th bit of exponent
    ei = (E->p[nblimbs] >> bufsize) & 1;

    if (ei == 0 && state == 0) continue;
    if (ei == 0 && state == 1)
      mpi_montmul(X, X, N, mm, &T);
    …
  } …
```

taken only when ei is one

Attack evaluation: Sliding-window exponentiation

```
/* X = A^E mod N */
mbedtls_mpi_exp_mod(X, A, E, N, _RR) {
  …
  while (1) {
```

We can recover 66% of a 1024-bit RSA private key from a *single run* (~10 runs are enough to fully recover it).

```
    if (ei == 0 && state == 1)
      mpi_montmul(X, X, N, mm, &T);
    …
  } …
```

Attack demo

https://youtu.be/jf9PanlF374

Hardware countermeasure: Flush branch history at SGX mode switch

Most effective, but need hardware modification

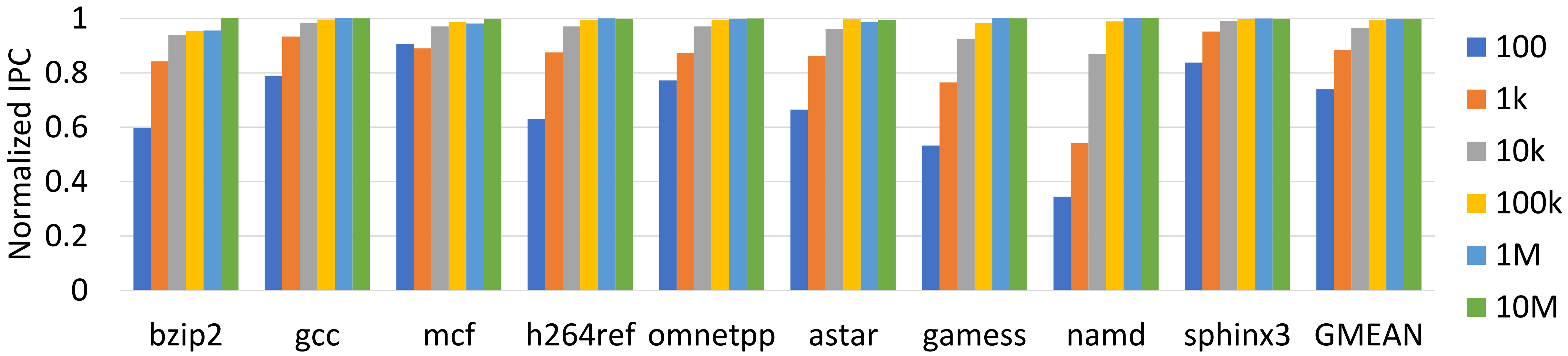
- It would not be realized by microcode update.

Overhead depends on how frequently SGX mode switch occurs.

Simulation result

Overhead was ~2% when mode switching occurs at every 100k cycles.

- Ten times frequent than the timer interrupt of Windows 10 (generated for every 1M cycles @ 4GHz CPU)



Software mitigation: Branch obfuscation

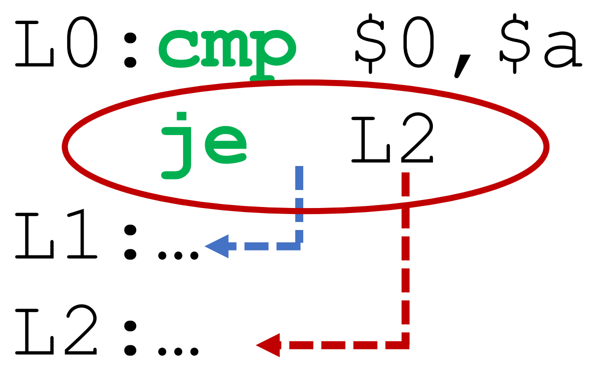
Replace a set of branches with a single indirect branch plus conditional move instructions

- Indirect branch only reveals when and whether it has been executed, not its target.
- Conditional move is used to conditionally update the indirect branch's target.

Modify LLVM for automatic transformation

- Average overhead: Below 1.3x (nbench)

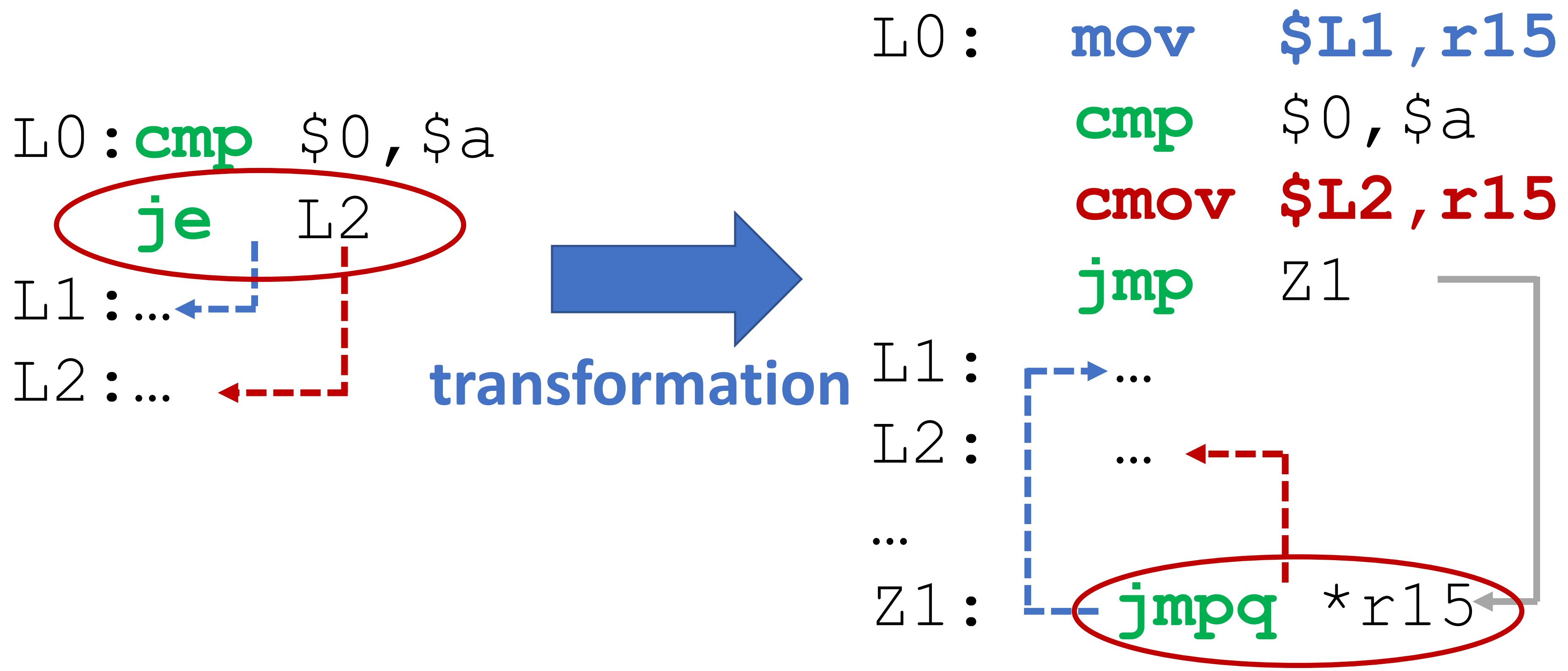
Example of branch obfuscation



```
L0:cmp $0,$a
   je  L2
L1:…
L2:…
```

Can identify whether L1 or L2 has been executed

Example of branch obfuscation



```
L0:cmp  $0,$a
   je   L2
L1:…
L2:…
```

```
L0:  mov   $L1,r15
     cmp   $0,$a
     cmov  $L2,r15
     jmp   Z1
L1:   …
L2:   …
…
Z1:  jmpq  *r15
```

Can identify whether L1 or L2 has been executed

Can identify whether Z1 has been executed but not its target

Conclusion

Branch shadowing: Fine-grained and deterministic side-channel attack on SGX

- Reveal direction and/or execution of individual branch instrs

Proposed hardware- and software-based countermeasures

- Branch history flushing and obfuscation

Thanks for listening!
Sangho Lee (sangho@gatech.edu)