ECE 382N-Sec (FA25):

L4: Data-Oblivious Computation

Neil Zhao

neil.zhao@utexas.edu

Previously on ECE 382N: Partitioning, Randomization, Detection

- **Resource Partitioning ⇒ Limit the sharing**
  - Way/Set partitioning for caches
  - Temporal partitioning
  - Dynamic partitioning
- **Randomization ⇒ Obfuscate the resource usage**
  - Randomize the address→set mapping
  - Make eviction set construction more difficult
  - Mimic a fully-associative cache
- **Detection ⇒ Catch the offender**
  - The false negative/positive concerns
  - Cyclic interference

Previously on ECE 382N: Partitioning, Randomization, Detection

- **Resource Partitioning ⇒ Limit the sharing**
  - Way/Set partitioning for caches
  - Temporal partitioning
  - Dynamic partitioning
- **Randomization ⇒ Obfuscate the resource usage**
  - Randomize the address→set mapping
  - Make eviction set construction more difficult
  - Mimic a fully-associative cache
- **Detection ⇒ Catch the offender**
  - The false negative/positive concerns

All these solutions require hardware or system software changes

Data-Oblivious/Constant-Time Programming

Informal definition[1]: A program is data-oblivious if for a fixed public input, its observable execution behavior is indistinguishable under any distinct secret input

Depends on:
- The environment: CPU microarchitecture, OS, runtime, …
- The attacker model:
  - Remote? End-to-end execution time
  - Co-location and share caches? Code and data accesses
  - Physical access? Power and EM patterns

[1] Refer to Definition III.1 from "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing" (NDSS '19) for the formal definition

Leaky String Comparison

A naïve implementation that compares strings s1 and s2

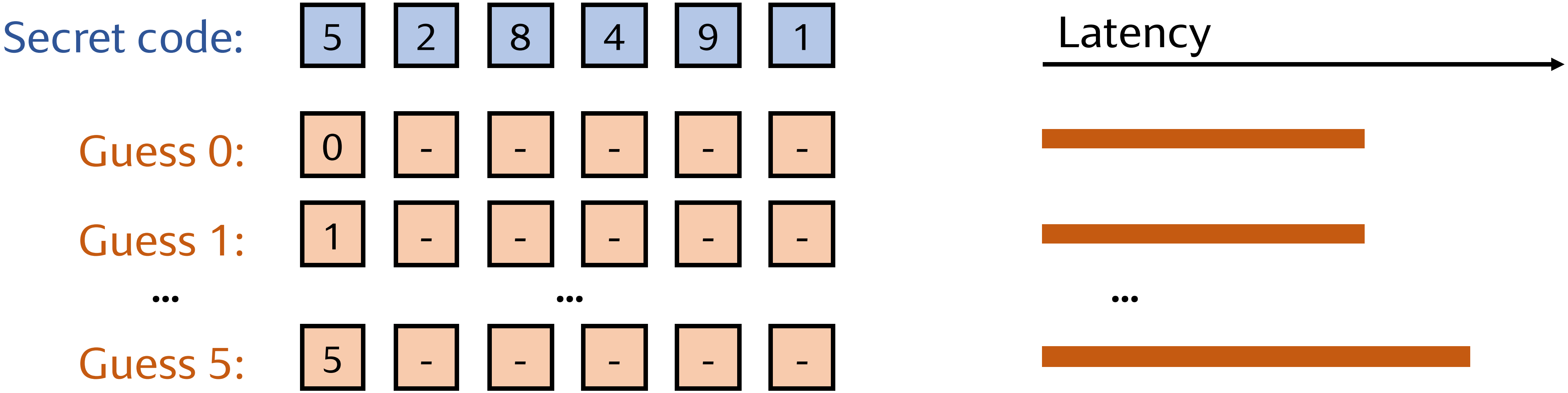
```
bool str_equal(const char *s1, const char *s2) {
    for (; *s1 && *s2; s1++, s2++) {
        if (*s1 != *s2) return false; // Early return
    }
    return *s1 == *s2;
}
```

More number of matching bytes between s1 and s2 → Longer execution time

Leaky String Comparison

Leaking a secret access code

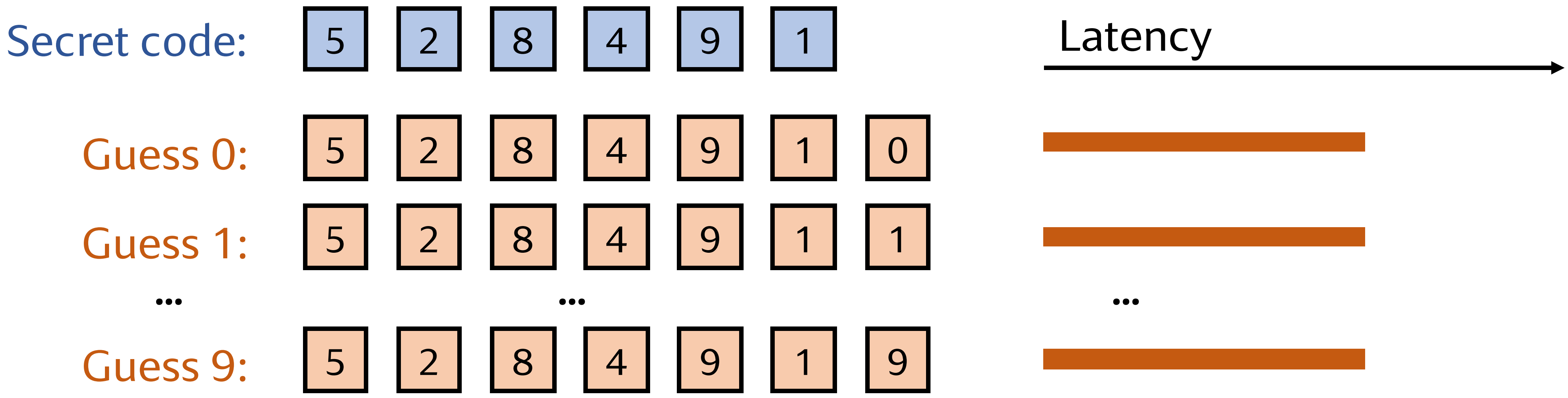
```c
bool check_access_code(const char *user_input) {
    return str_equal(user_input, SECRET);
}
```



Leaky String Comparison

Leaking the length of the secret access code

```
bool check_access_code(const char *user_input) {
    return str_equal(user_input, SECRET);
}
```



Fix? No Early Return

```
bool fixed_str_equal(const char *s1, const char *s2) {
    bool ret = 1;
    for (; *s1 && *s2; s1++, s2++) {
        ret &= (*s1 == *s2);
    }
    return ret & (*s1 == *s2);
}
```

The string length N still leaks, but it will need $O(10^N)$ instead of $O(N)$ guesses

Compiler Can Mess Things Up

```
bool fixed_str_equal(const char *s1, const char *s2) {
    bool ret = 1;
    for (; *s1 && *s2; s1++, s2++) {
        ret &= (*s1 == *s2);
    }
    return ret & (*s1 == *s2);
}
```

Compiler optimizations

```
...
    for (; *s1 && *s2; s1++, s2++) {
        ret &= (*s1 == *s2);
        if (!ret) return 0; // Early return again!
    }
...
```

Real-World Example: Balanced Branch → Constant E2E Timing

Montgomery ladder: Scalar multiplication on elliptic curves
Used in ECDSA (Elliptic Curve Digital Signature Algorithm)

```
BIGNUM x1, z1, x2, z2;
for k_i in k { // k is the secret nonce
  if (k_i) {
    Madd(&x1, &z1, &x2, &z2);
    Mdouble(&x2, &z2);
  } else {
    Madd(&x2, &z2, &x1, &z1);
    Mdouble(&x1, &z1);
  }
}
```

Secret-dependent ctrl. flow!
Call sites are on different cache lines ⇒ Flush+Reload[1]

Learning **k** and the corresponding signature ⇒ Private key used for signing

[1]Yarom et al., "Recovering OpenSSL ECDSA Nonces Using the Flush+Reload Cache Side-Channel Attack"

Real-World Example: Balanced Branch → Constant E2E Timing

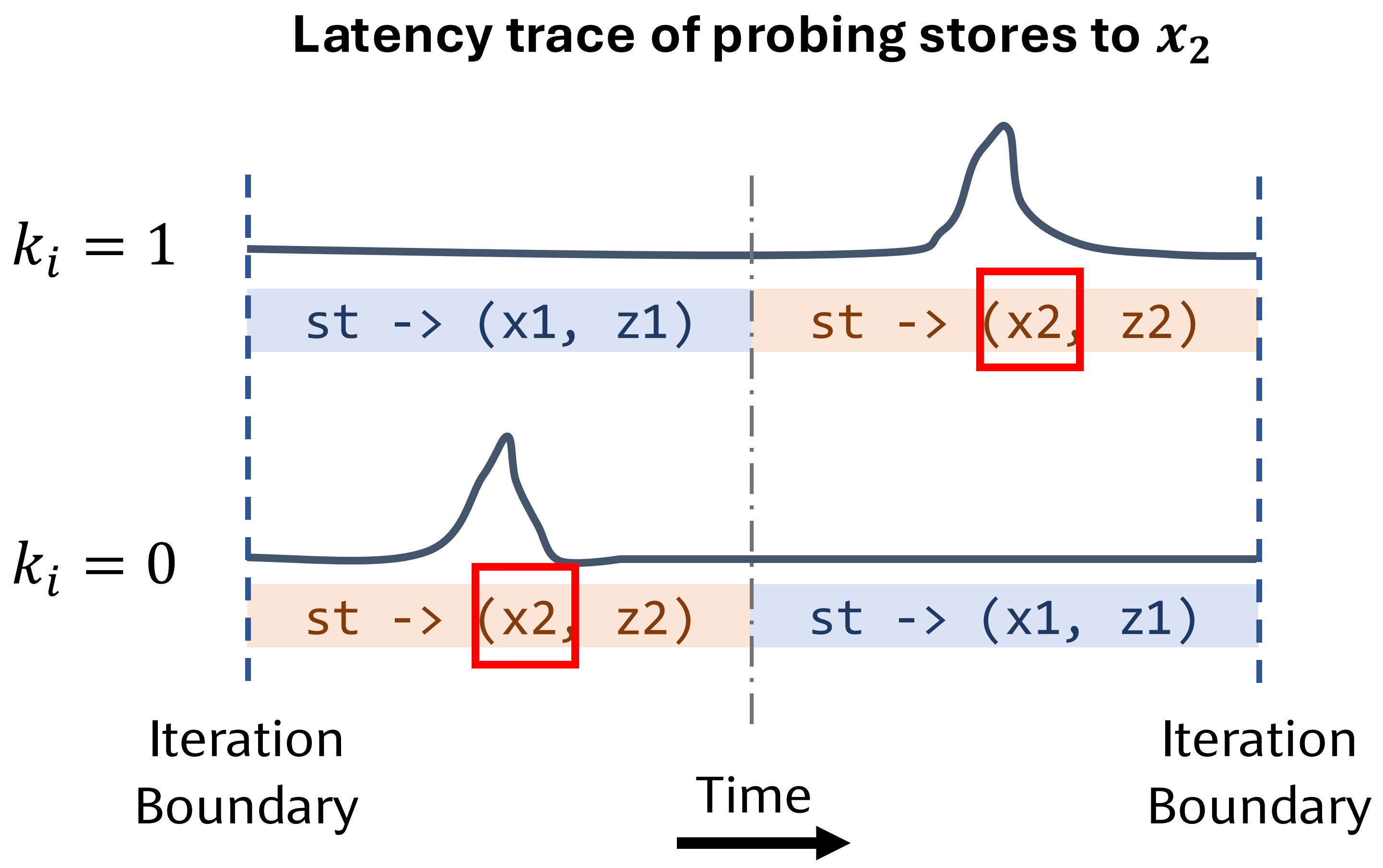
Montgomery ladder: Scalar multiplication on elliptic curves
Used in ECDSA (Elliptic Curve Digital Signature Algorithm)

```
BIGNUM x1, z1, x2, z2;
for k_i in k { // k is the secret nonce
  if (k_i) {
    Madd(&x1, &z1, &x2, &z2);    st -> (x1, z1)
    Mdouble(&x2, &z2);           st -> (x2, z2)
  } else {
    Madd(&x2, &z2, &x1, &z1);    st -> (x2, z2)
    Mdouble(&x1, &z1);           st -> (x1, z1)
  }
}
```

Secret-dependent memory accesses

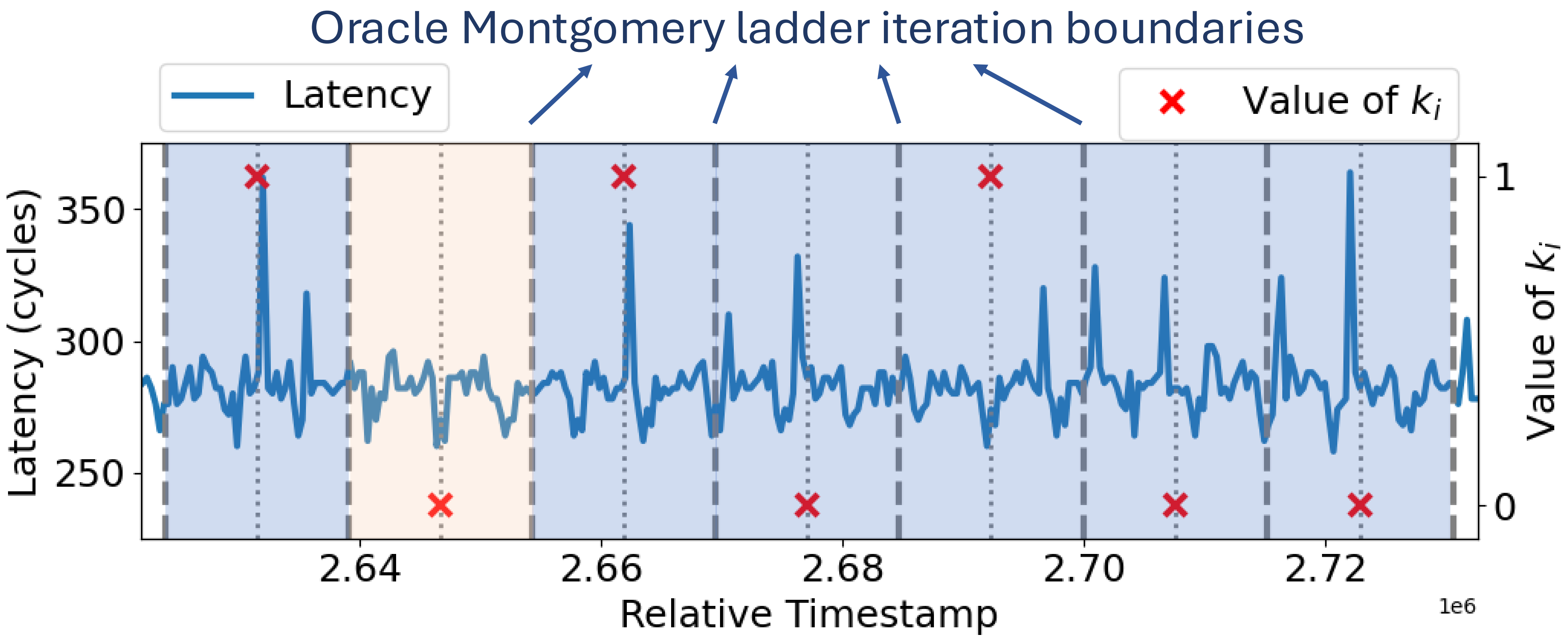
Learning **k** and the corresponding signature ⇒ Private key used for signing

Detect Memory Writes with Binoculars Attack[1]



[1]Zhao et al., "Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker"

Detect Memory Writes with Binoculars Attack[1]



[1]Zhao et al., "Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker"

A Naïve Fix Through CMOV

```
BIGNUM x1, z1, x2, z2;
BIGNUM x1_t, z1_t, x2_t, z2_t;

for k_i in k { // k is the secret nonce
  // true branch
  x1_t = x1; ... z2_t = z2;
  Madd(&x1_t, &z1_t, &x2_t, &z2_t);
  Mdouble(&x2_t, &z2_t);

  // false branch
  Madd(&x2, &z2, &x1, &z1);
  Mdouble(&x1, &z1);

  x1 = CMOV(k_i, x1_t); ... z2 = CMOV(k_i, z2_t);
}
```

There are better constant-time Montgomery ladder implementations. See "[Montgomery curves and the Montgomery ladder]" by DJB et al.

CMOV, Constant Time?

"CMOVcc loads data from its source operand into a temporary register **unconditionally** (regardless of the condition code and the status flags in the EFLAGS register). If the condition code associated with the instruction (cc) is satisfied, the data in the temporary register is then copied into the instruction's destination operand"
from "Intel® 64 and IA-32 Architectures Software Developer's Manual"

Yes, `cmove (%rax), %rbx` does load from the address stored in `%rax` regardless of the predicate, and it can fault

List instructions that are data-oblivious:

- Intel: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/data-operand-independent-timing-instructions.html
- ARM: Data-Independent Timing

CMOV, Constant Time?

"CMOVcc loads data from its source operand into a temporary register **unconditionally** (regardless of the condition code and the status flags in the EFLAGS register). If the condition code associated with the instruction (cc) is satisfied, the data in the temporary register is then copied into the instruction's destination operand"
from "Intel® 64 and IA-32 Architectures Software Developer's Manual"

Yes, `cmove (%rax), %rbx` does load from the address stored in `%rax` regardless of the predicate, and it can fault

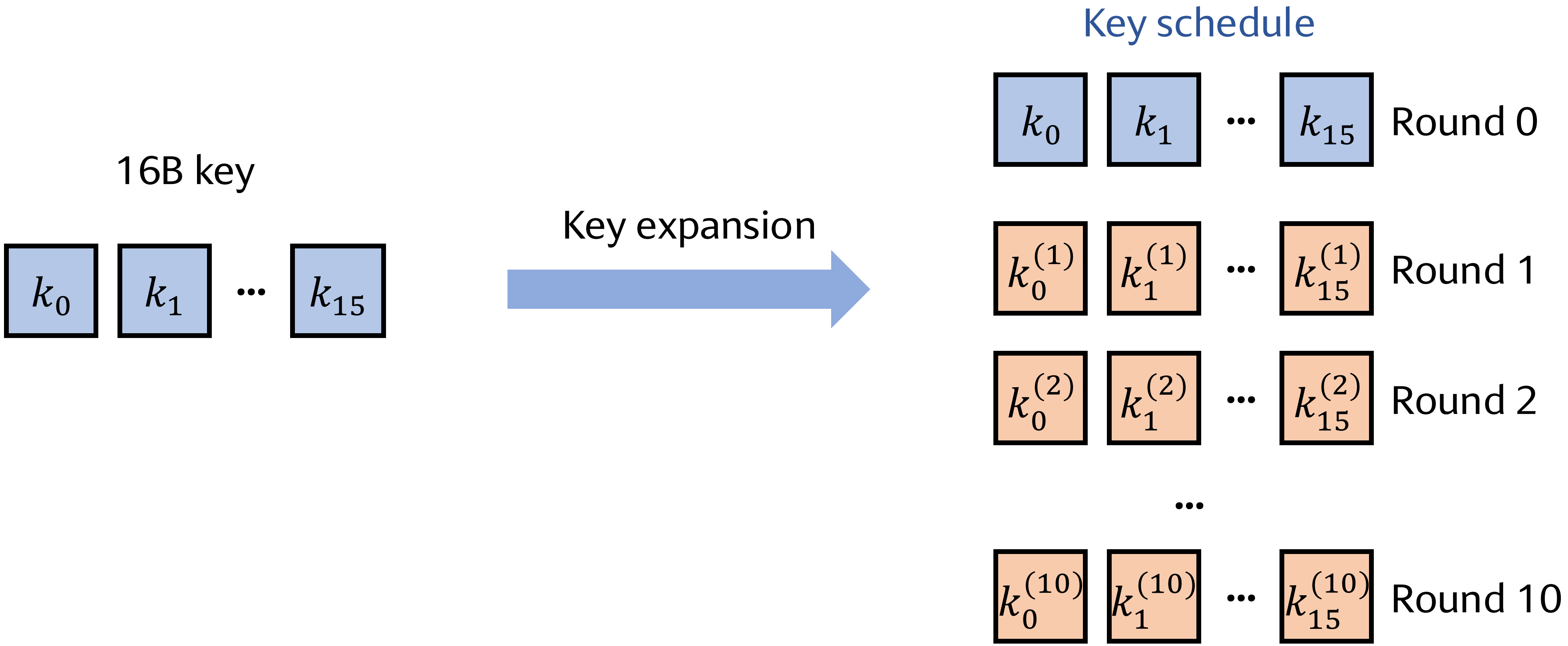
Side note: The bitmask select trick

```
uint64_t select_u64(uint64_t T, uint64_t F, bool predicate) {
    uint64_t mask = (uint64_t)0 - (uint64_t)(predicate != false);
    return (mask & T) | (~mask & F);
}
```

Secret-Dependent Accesses in AES

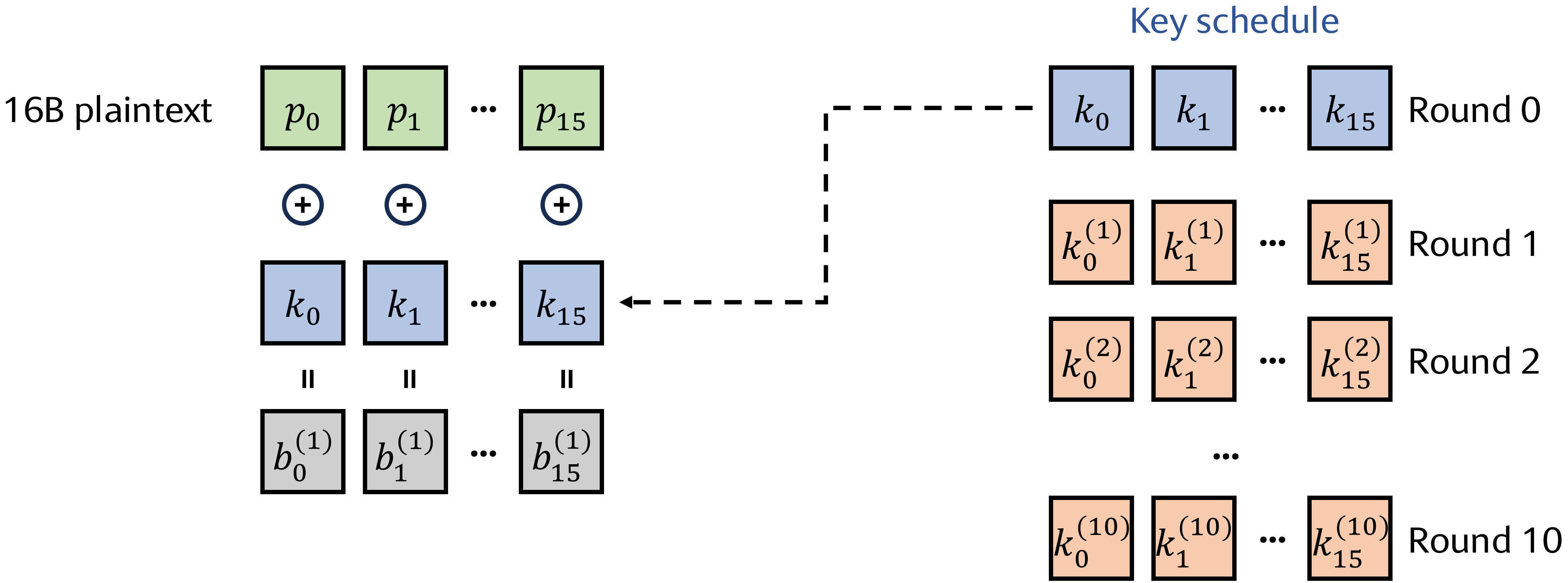
Symmetric block cipher: 16B plaintext → 16B ciphertext

AES-128: 16B key, 10 rounds



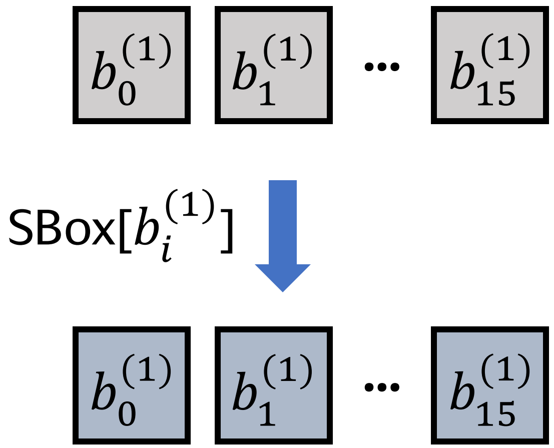
Secret-Dependent Accesses in AES

Round 0: AddRoundKey



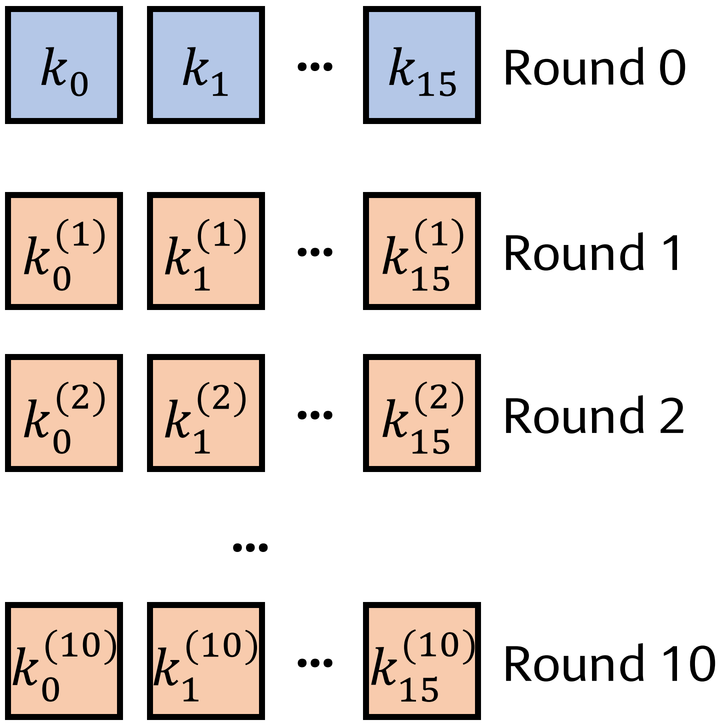
Secret-Dependent Accesses in AES

Round 1: SubBytes Step



The memory access pattern depends on $b_i^{(1)}$

Key schedule



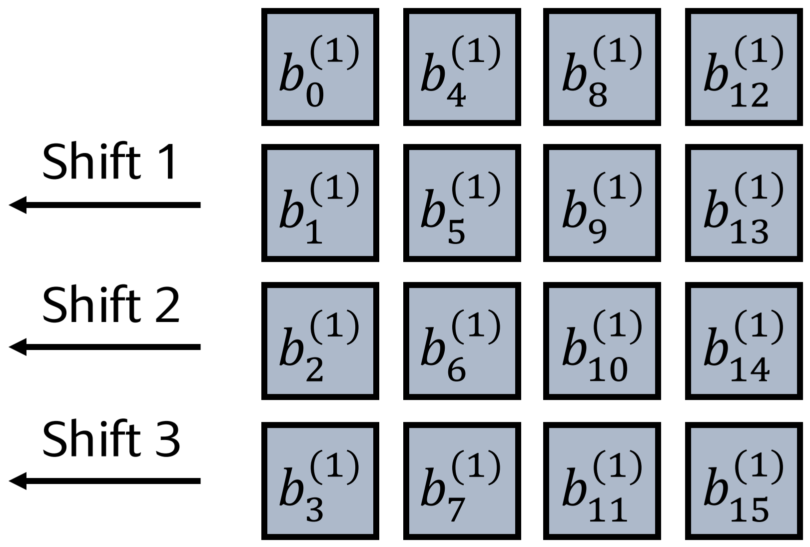
Secret-Dependent Accesses in AES

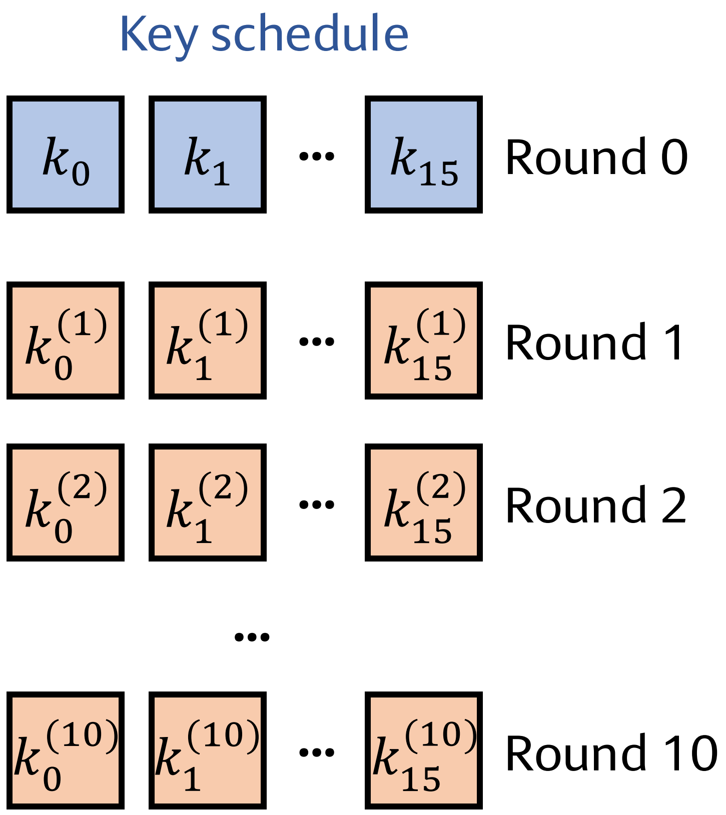
SBox is a 256B look-up table (occupies 4 cache lines)

```
static const uint8_t sbox[256] = {
  //0     1    2     3     4    5     6     7      8    9     A      B    C     D     E     F
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16};
```

Secret-Dependent Accesses in AES

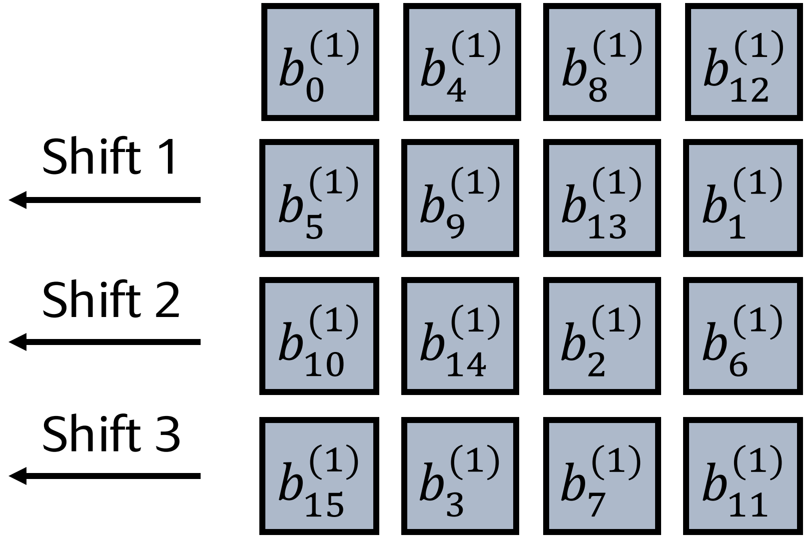
Round 1: ShiftRows Step

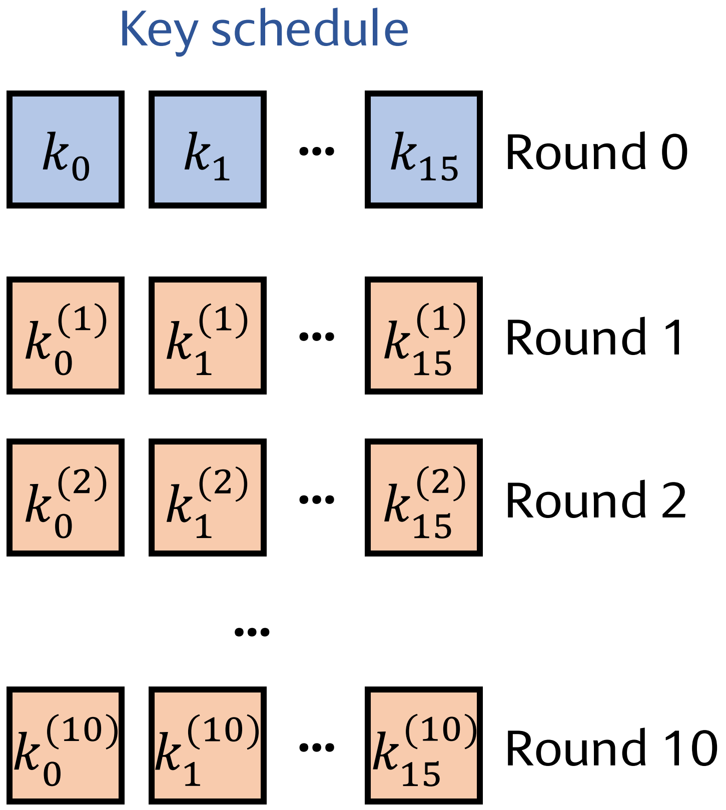




Secret-Dependent Accesses in AES

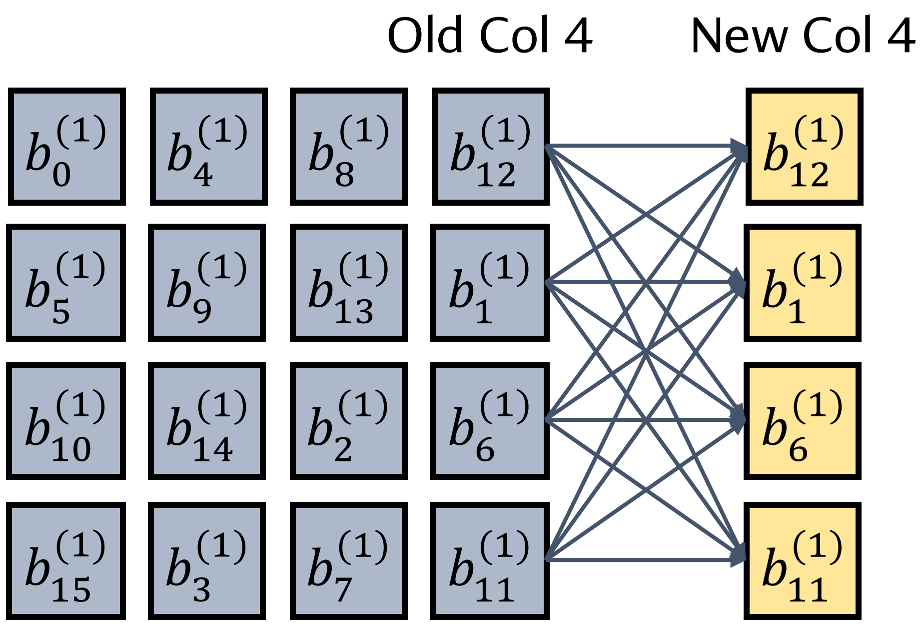
Round 1: ShiftRows Step



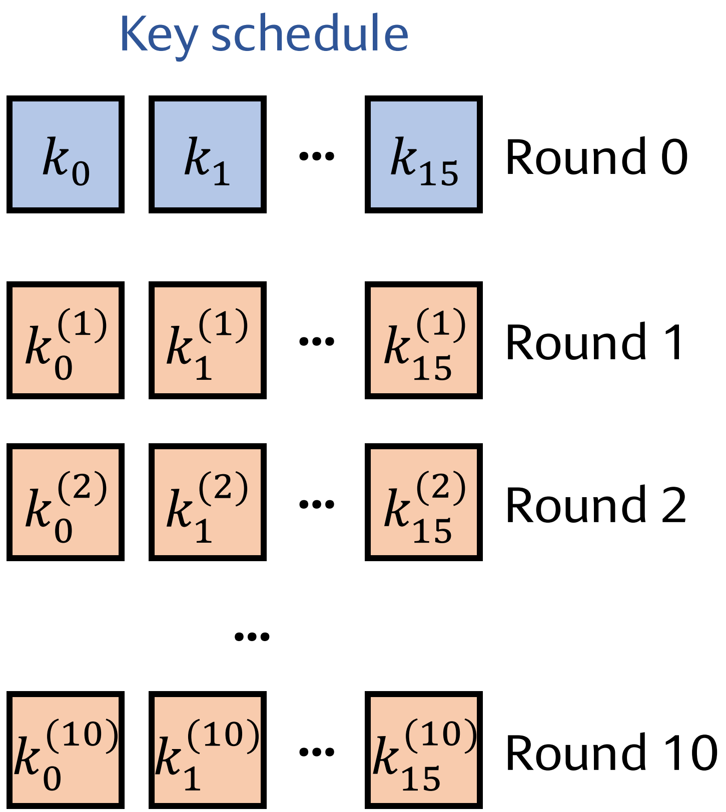


Secret-Dependent Accesses in AES

Round 1: MixColumn Step

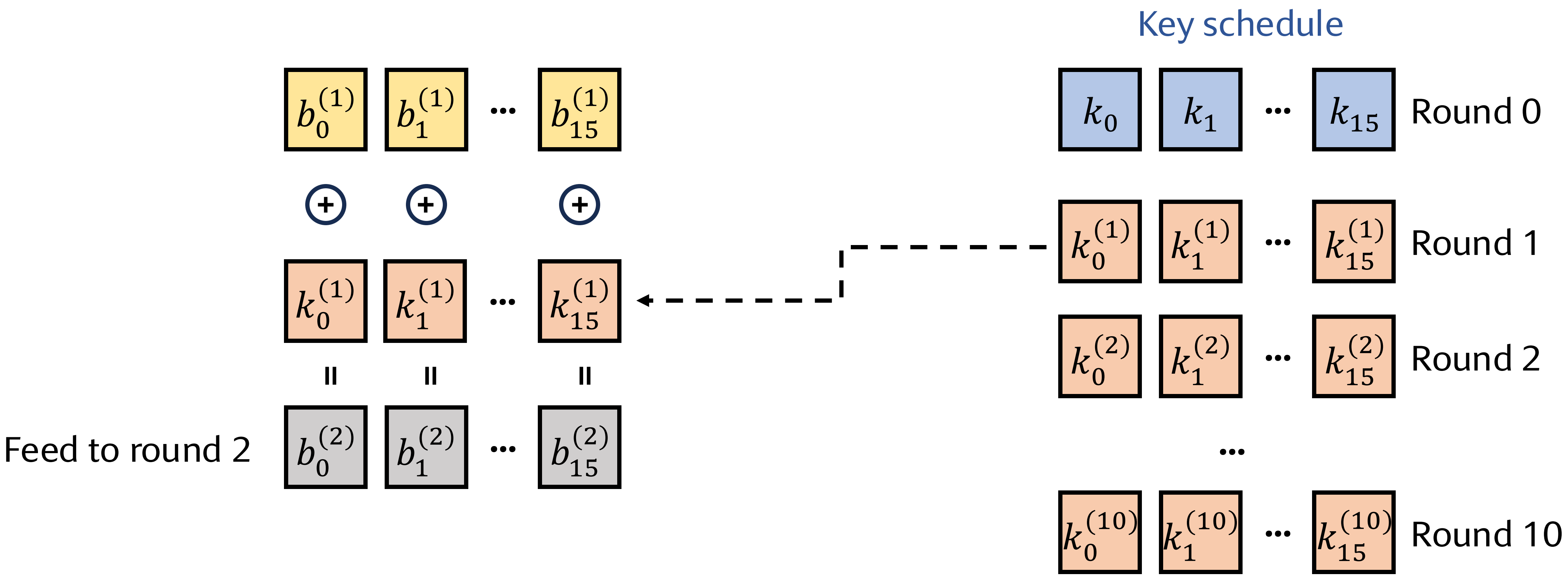


Repeat for each column

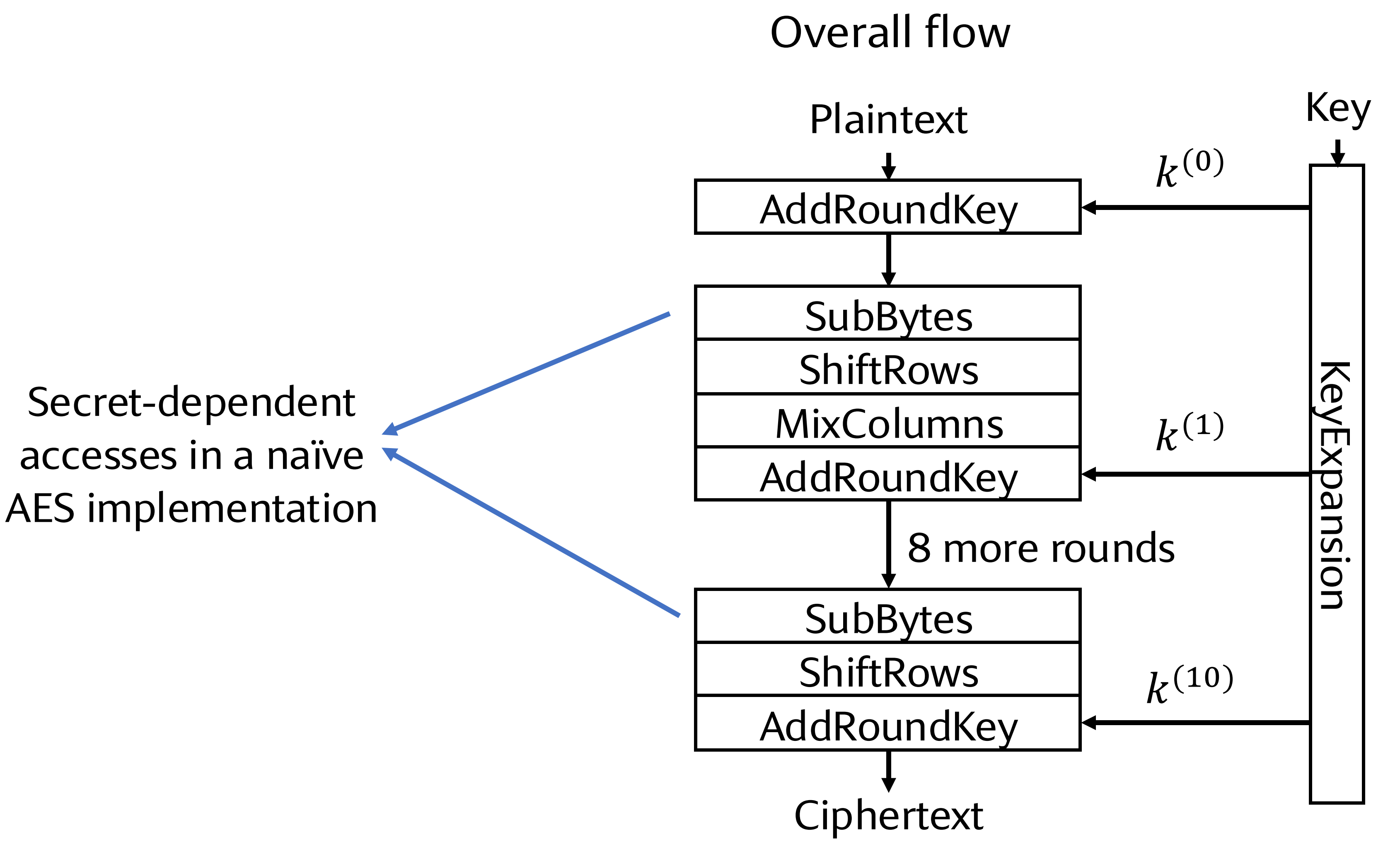


Secret-Dependent Accesses in AES

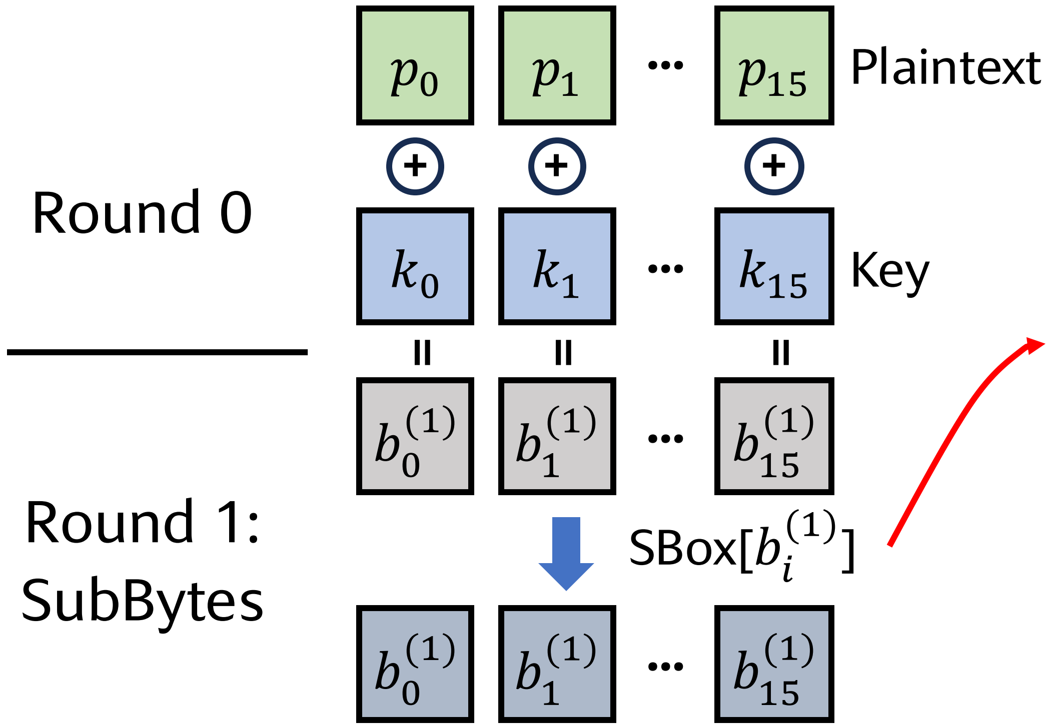
Round 1: AddRoundKey Step



Secret-Dependent Accesses in AES



First-Round Attack



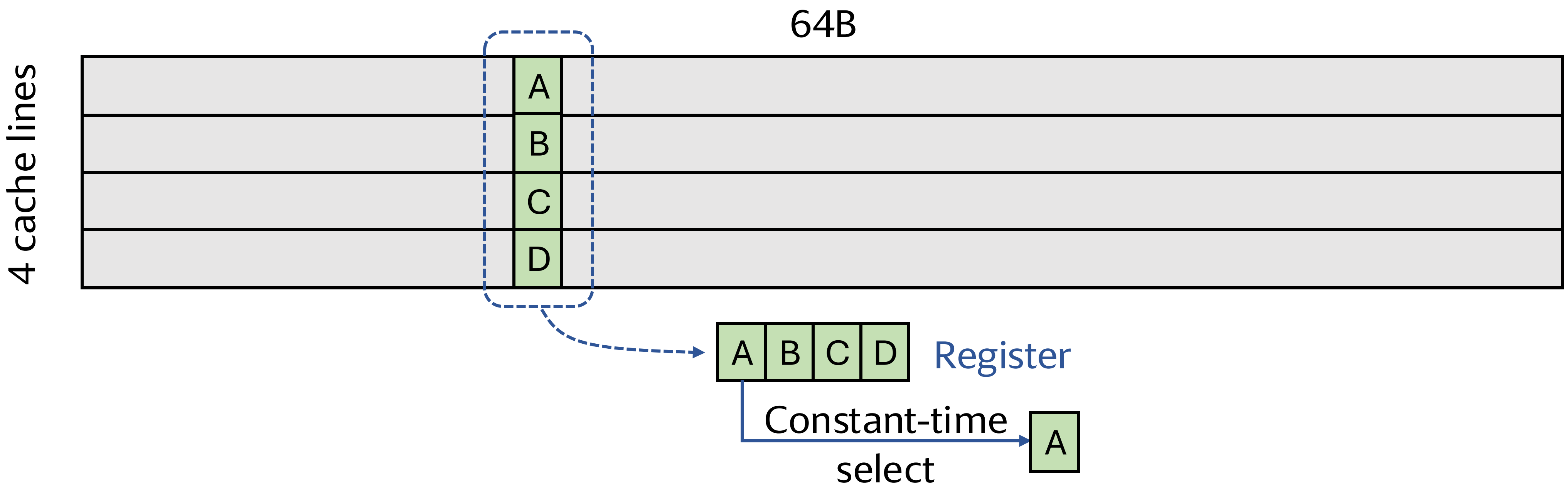
Leaks the upper 2 bits of $b_i^{(1)}$ with Flush+Reload or Prime+Probe

Since $b_i^{(1)} = k_i \oplus p_i$, if attacker controls the plaintext p
⇒ Learn the upper 2 bits of every key byte

An Attempt to Block the Leakage

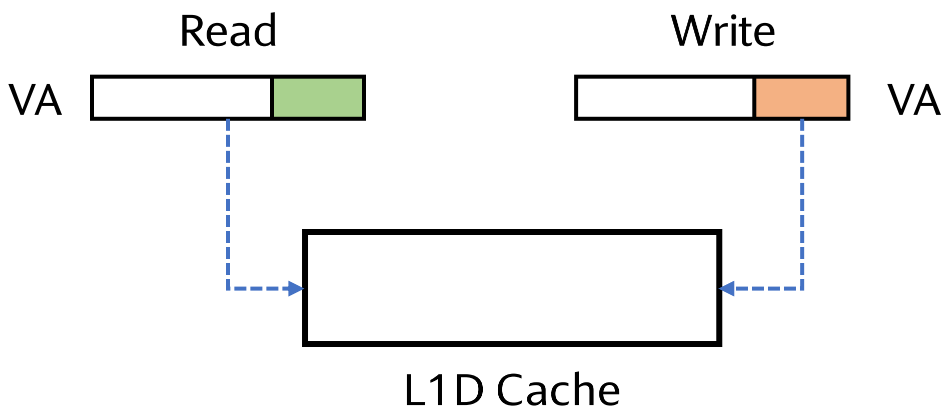
Intel Integrated Performance Primitives (IPP) Constant-Time AES

High-level idea: Cache side-channel can only observe memory accesses at a cache line granularity



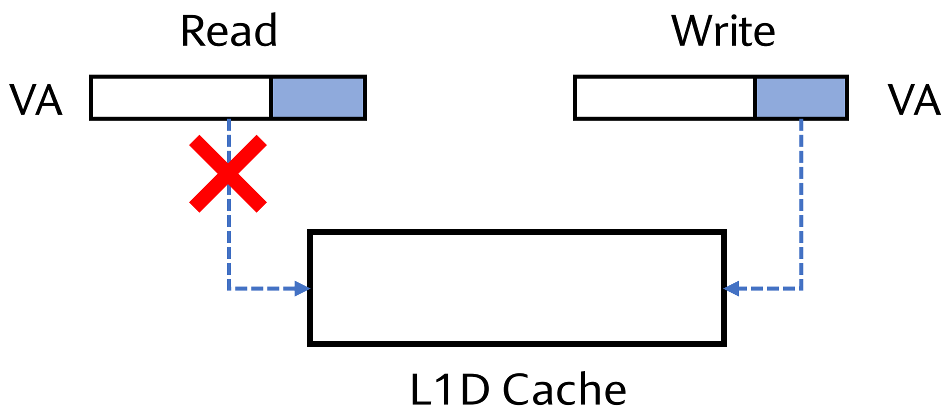
MemJam: False Dependencies in L1D

Different offsets (VA bits 11-0)



No dependence

Same offsets (VA bits 11-0)

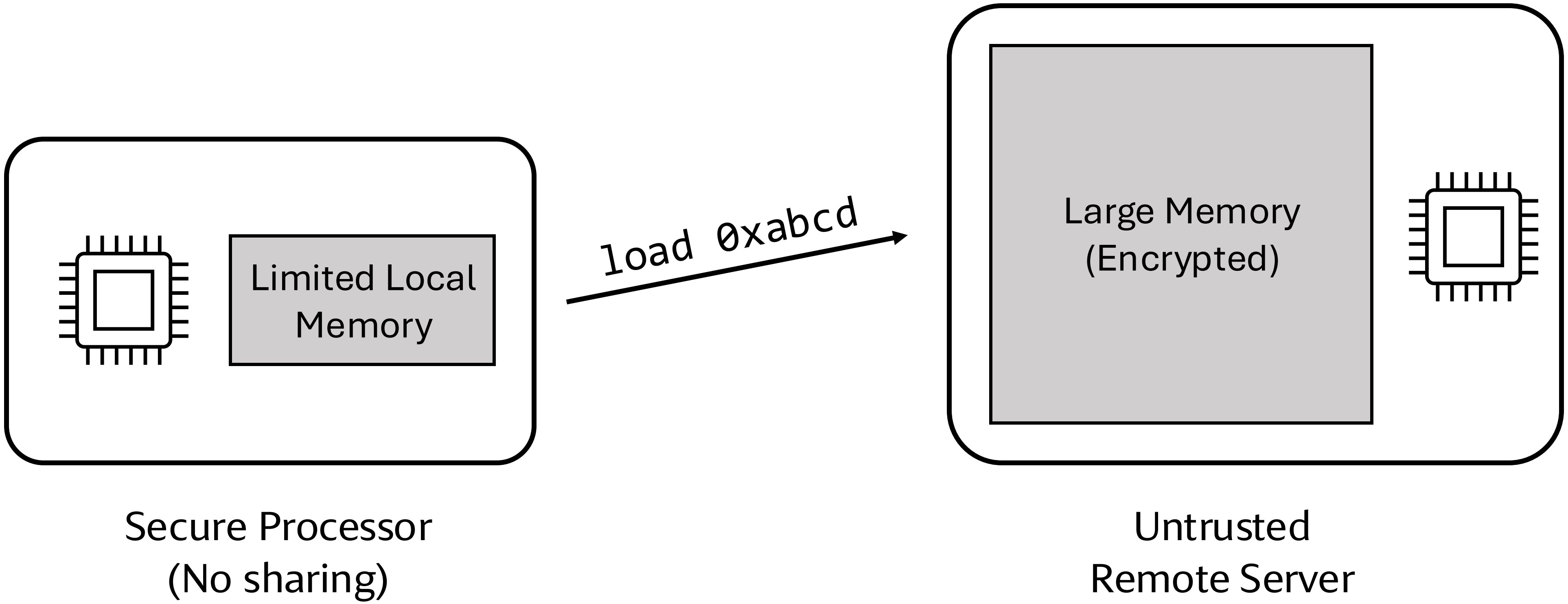


Potential dependence
(may be a *false dependence*)

One request is squashed and will retry

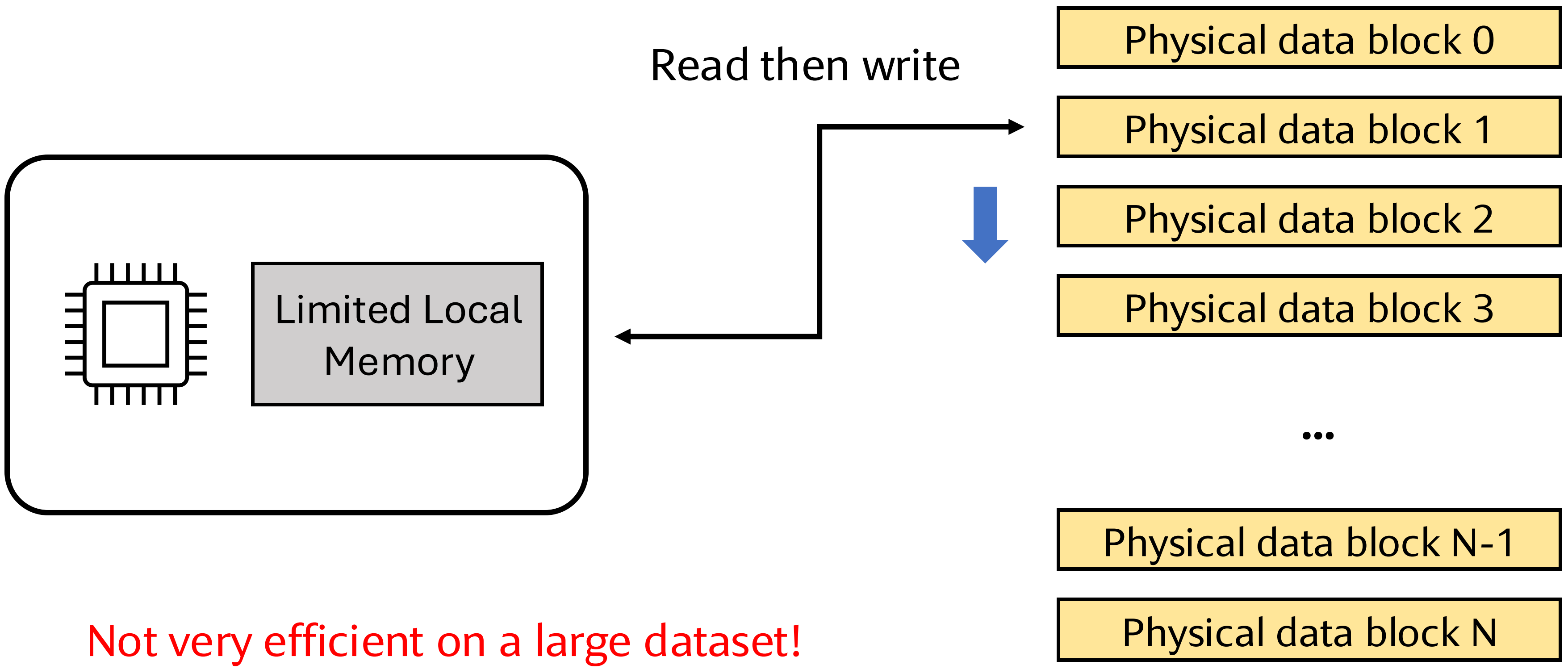
Implementation: Check can be done at a word granularity---i.e., two VAs only need to share bits 11-2 to be counted as potentially dependent (“4K-aliasing”)
⇒ Observe memory accesses at a sub-cacheline granularity

Oblivious RAM: Hide Data Access Pattern



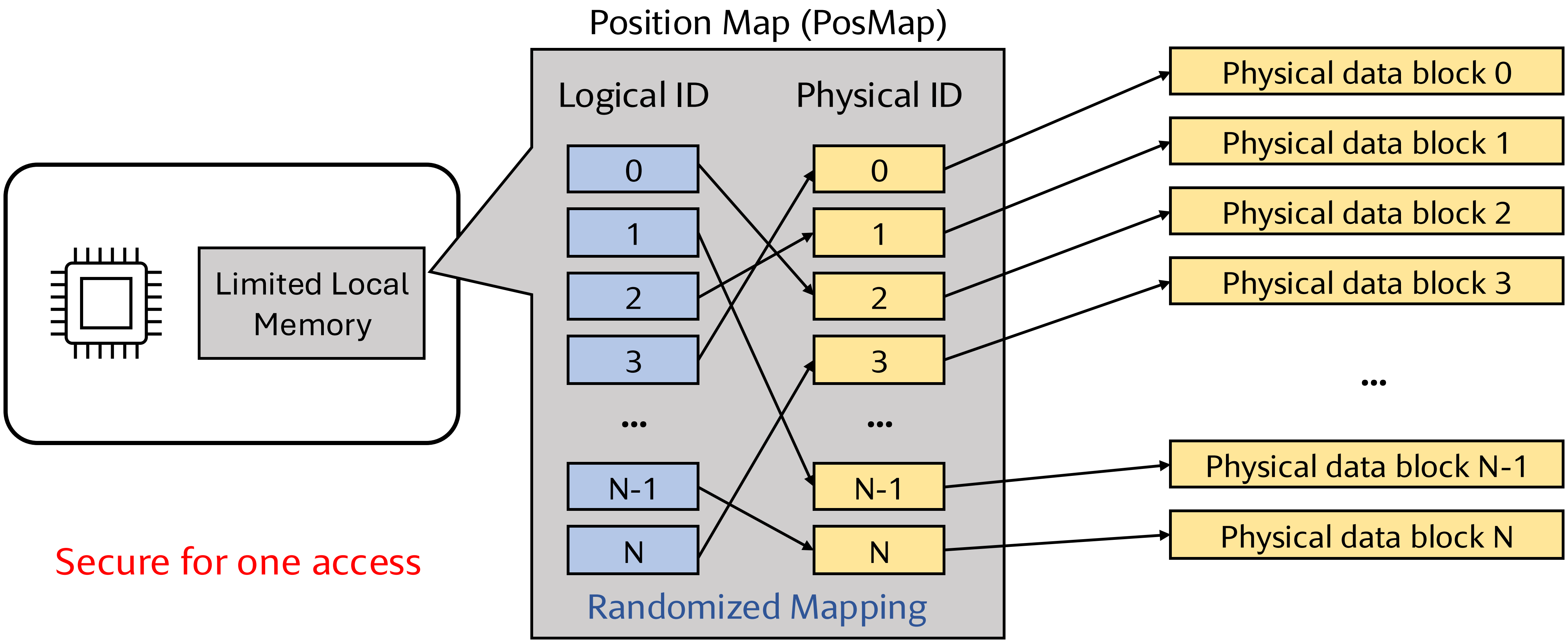
Data blocks are encrypted using a randomized encryption scheme
We want to hide our access pattern (e.g., which data block is accessed)

Linear Scan



Not very efficient on a large dataset!

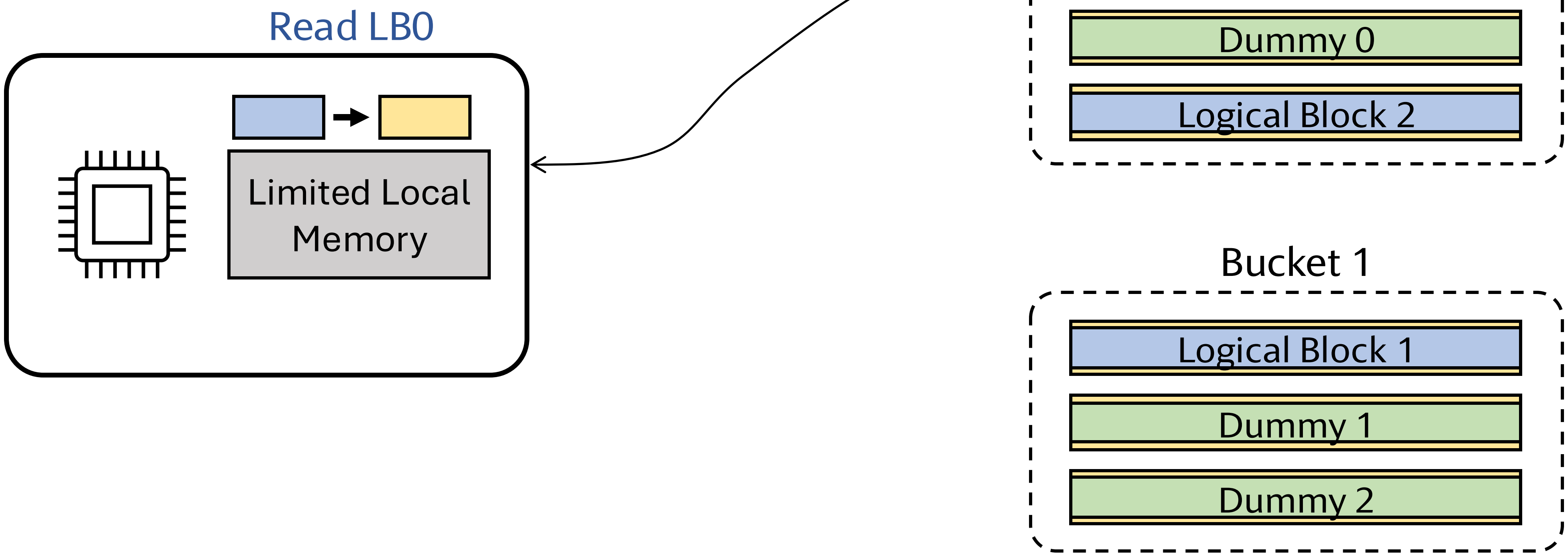
Randomize the Mapping



Secure for one access

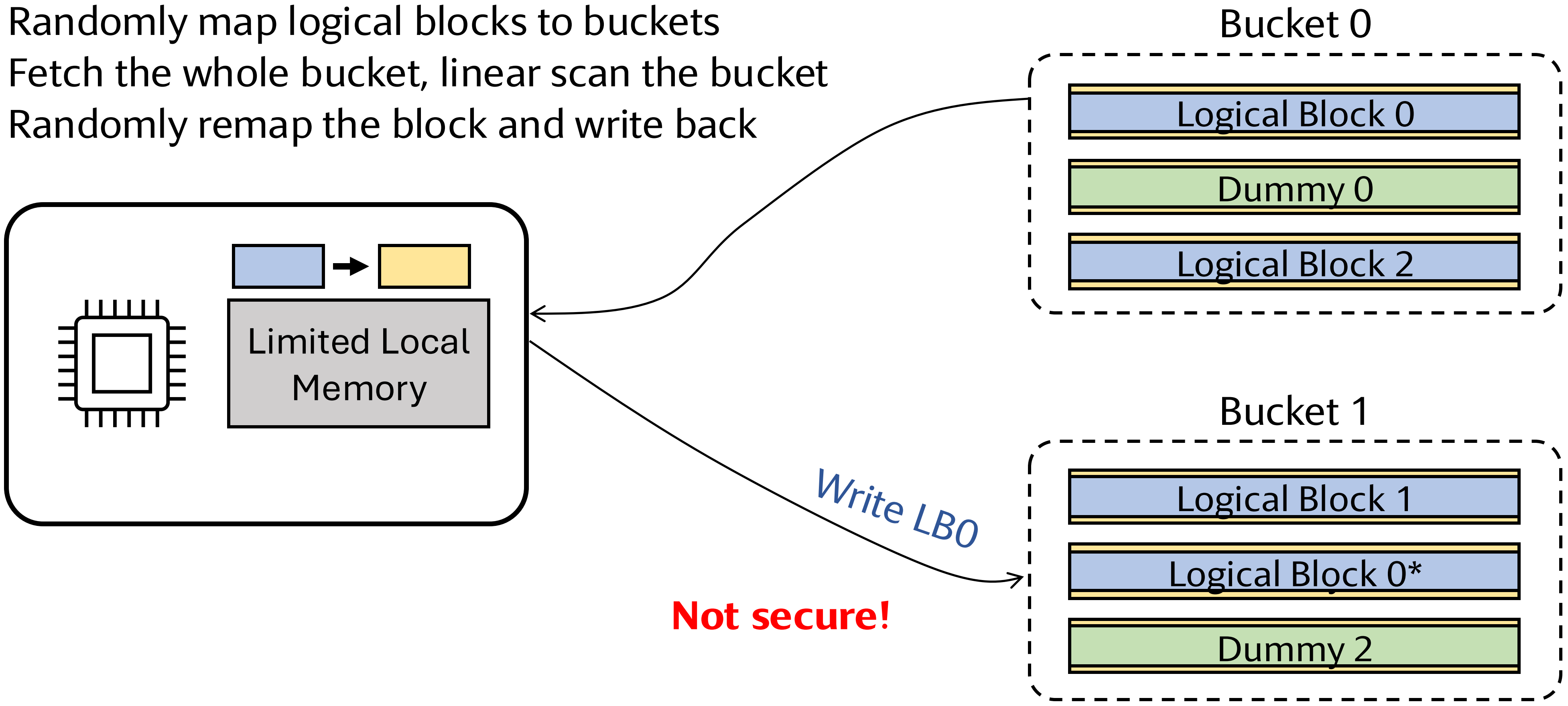
Remapping and Buckets

- Randomly map logical blocks to buckets
- Fetch the whole bucket, linear scan the bucket

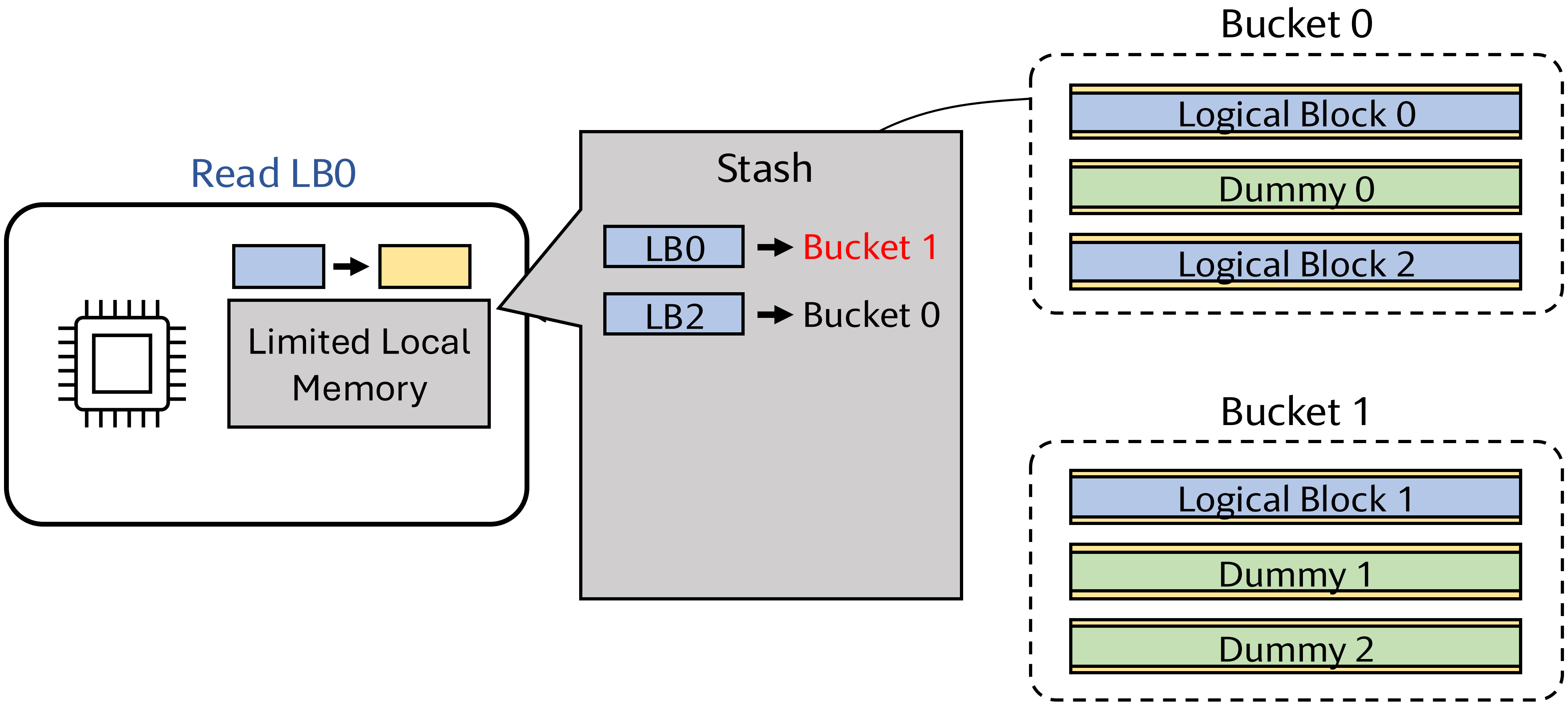


Remapping and Buckets

- Randomly map logical blocks to buckets
- Fetch the whole bucket, linear scan the bucket
- Randomly remap the block and write back

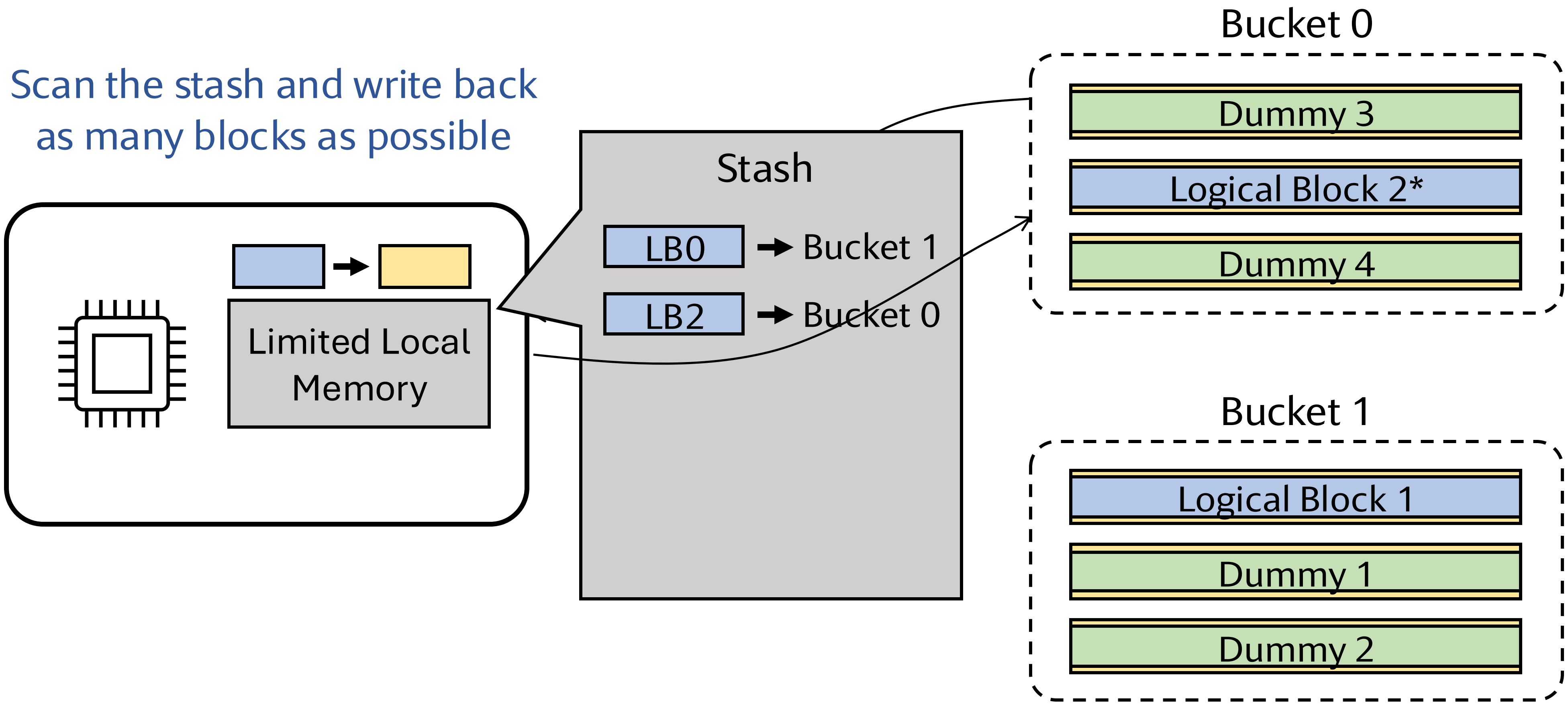


Hide the Shuffling with One Extra Writeback per Access

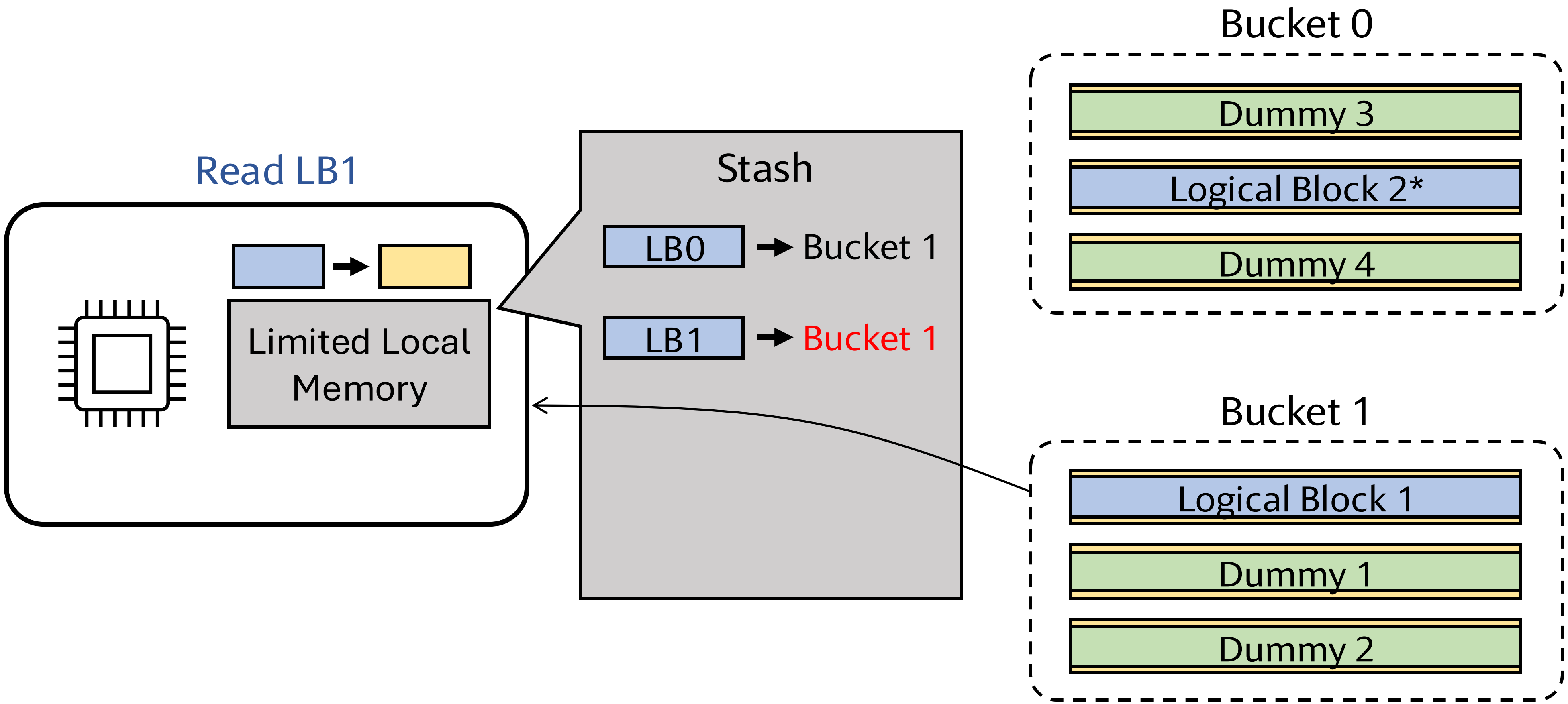


Hide the Shuffling with One Extra Writeback per Access

Scan the stash and write back as many blocks as possible

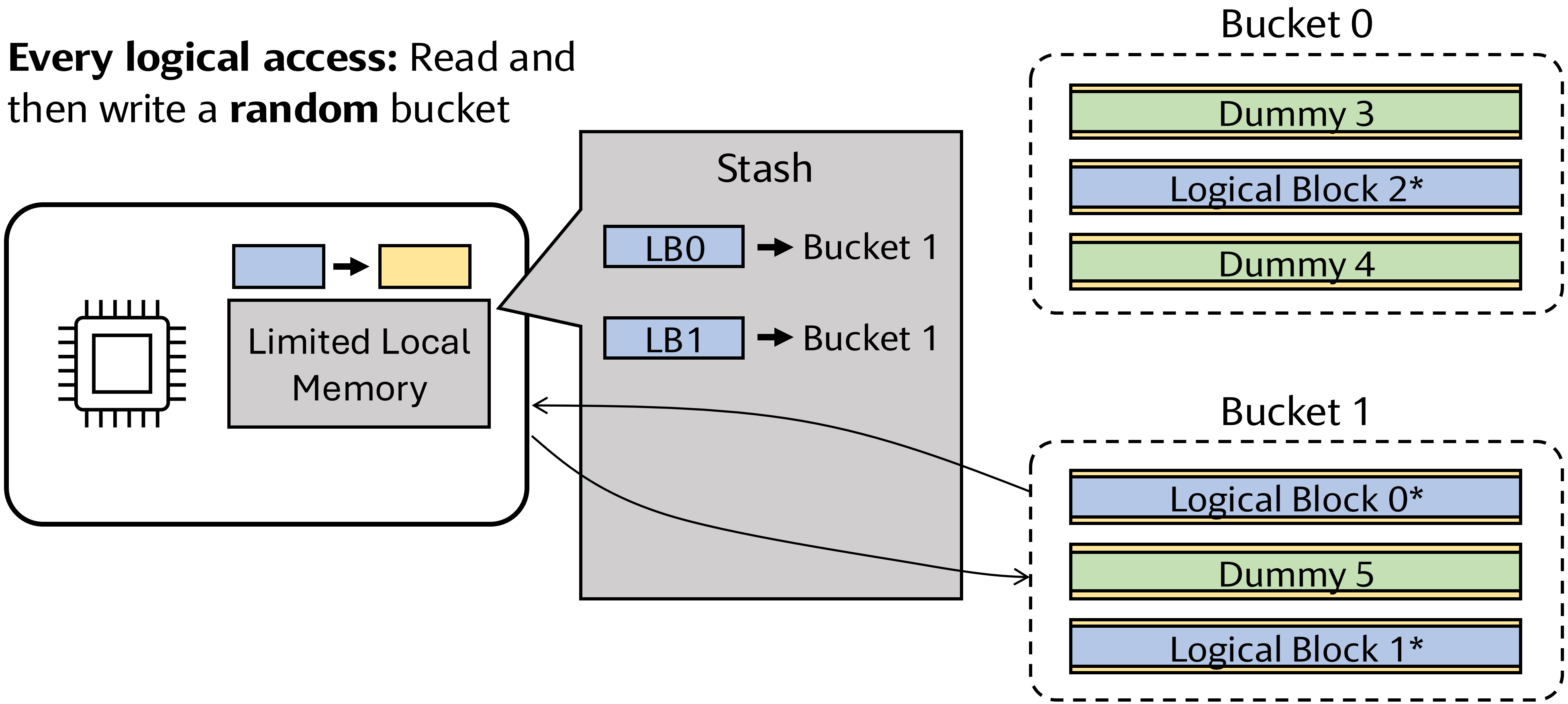


Hide the Shuffling with One Extra Writeback per Access

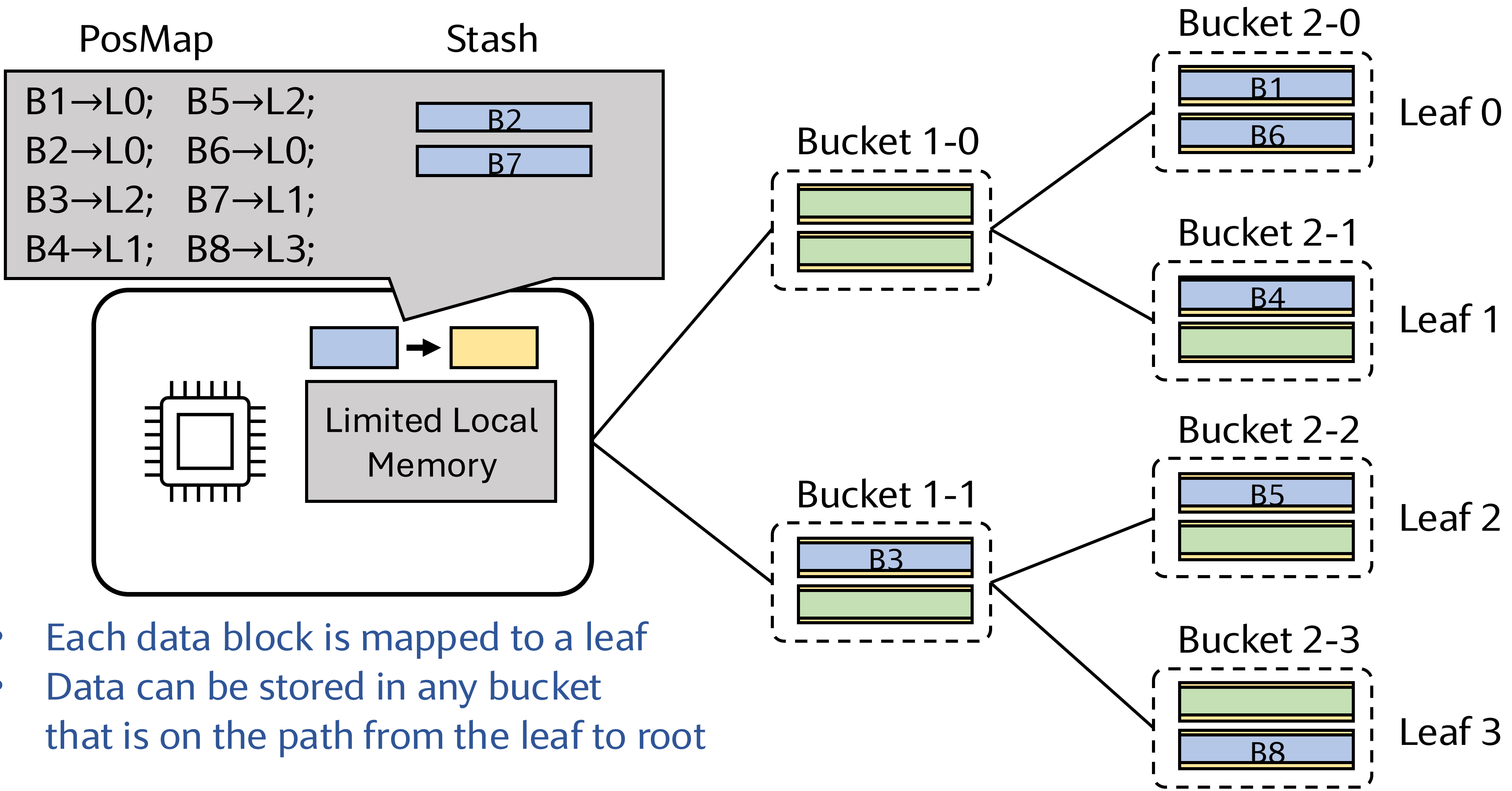


Hide the Shuffling with One Extra Writeback per Access

Every logical access: Read and then write a **random** bucket

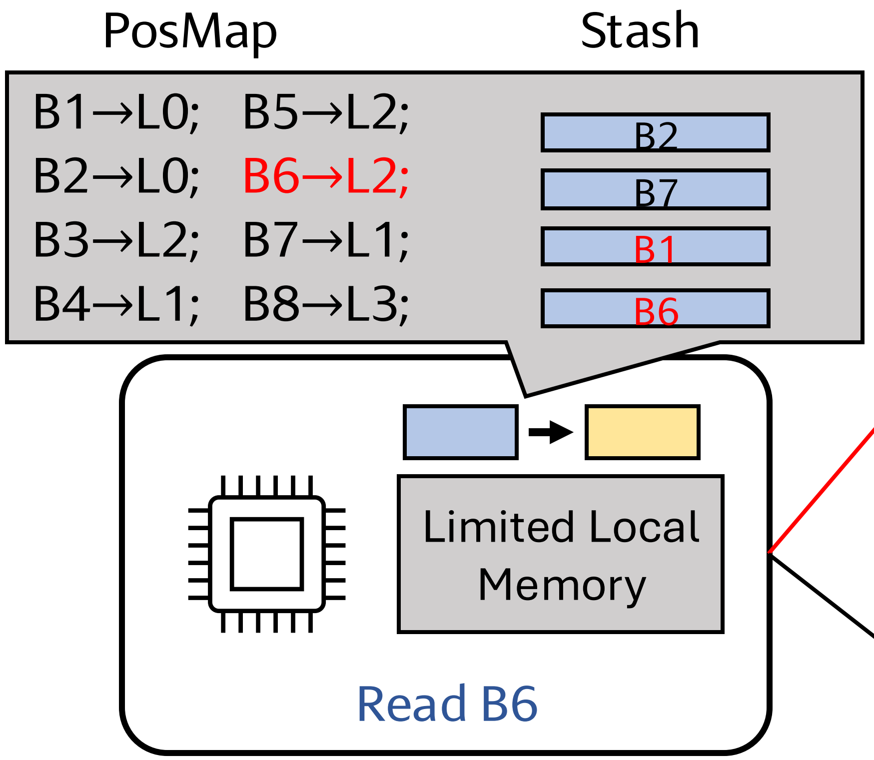


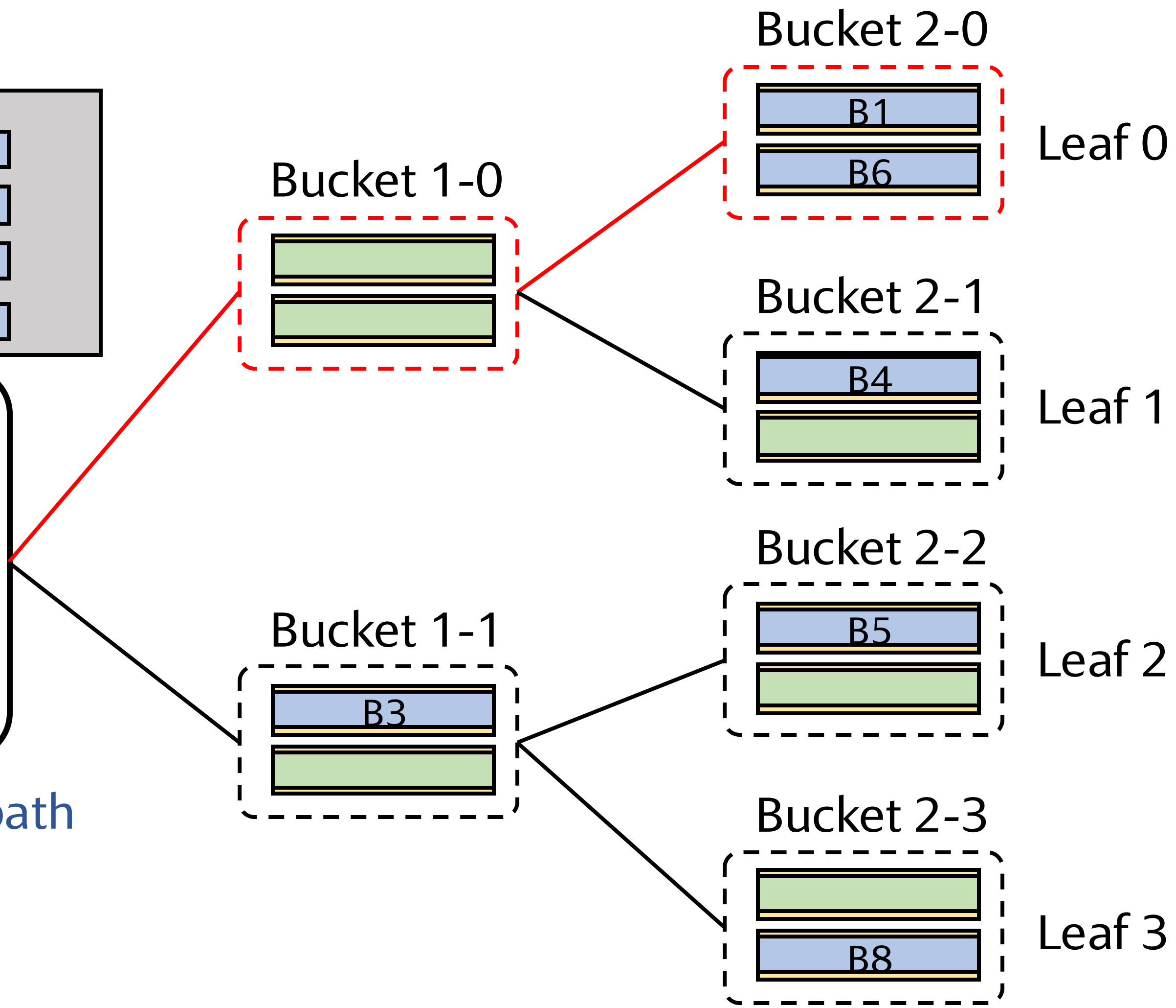
PathORAM



- Each data block is mapped to a leaf
- Data can be stored in any bucket that is on the path from the leaf to root

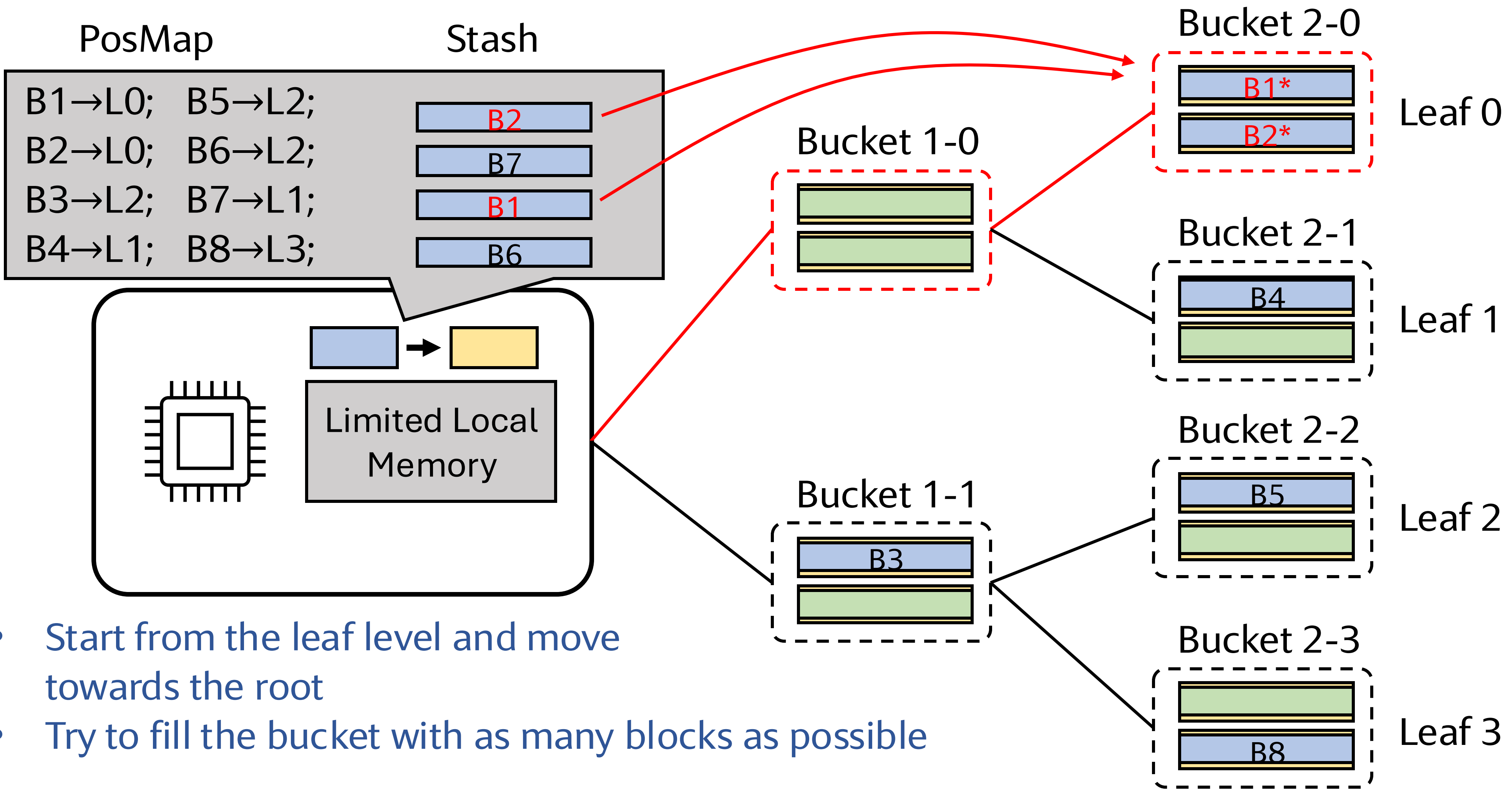
PathORAM: Read B6





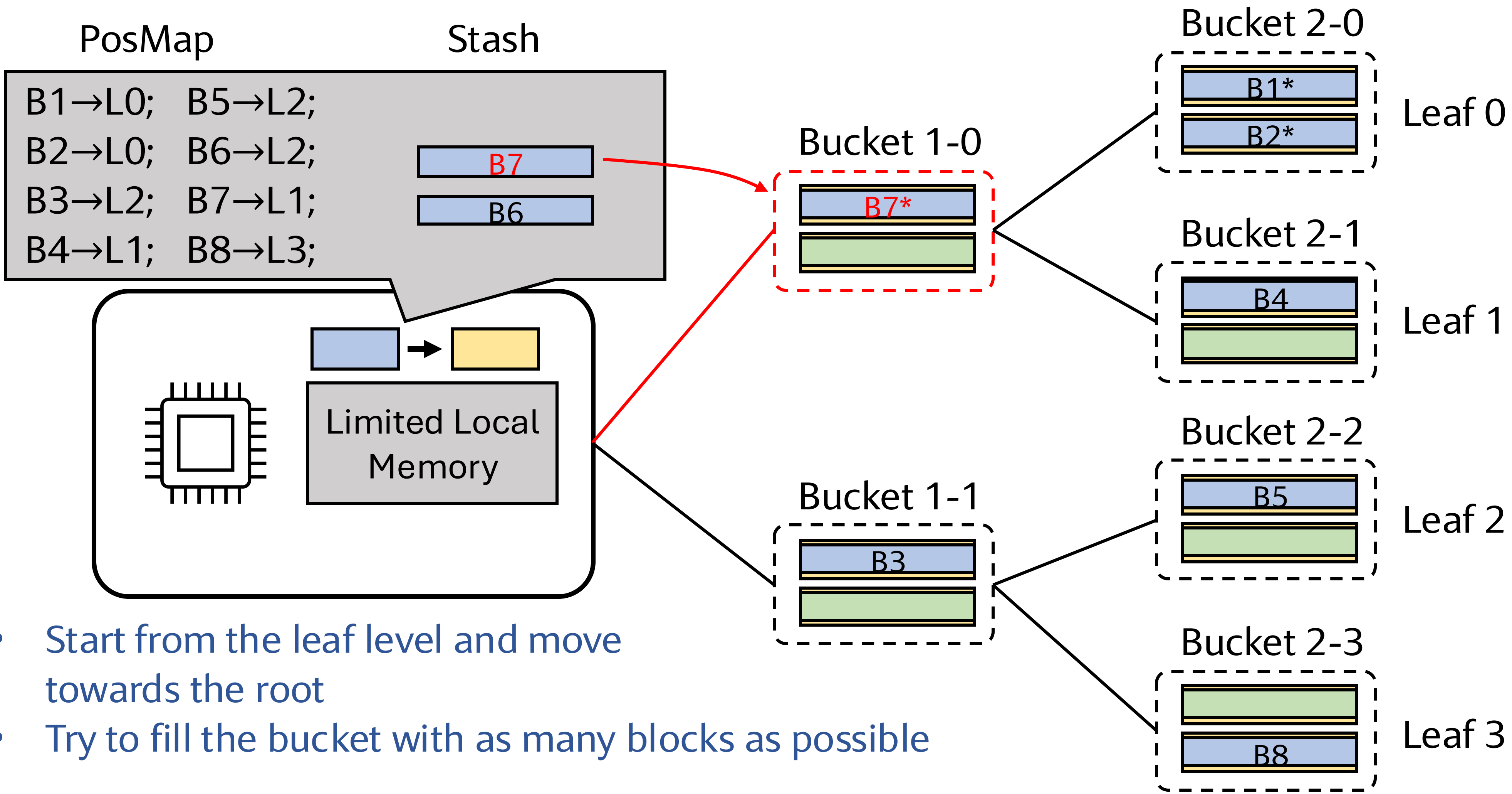
- Read all the buckets along the path
- Select the data block
- Update its mapping

PathORAM: Write back



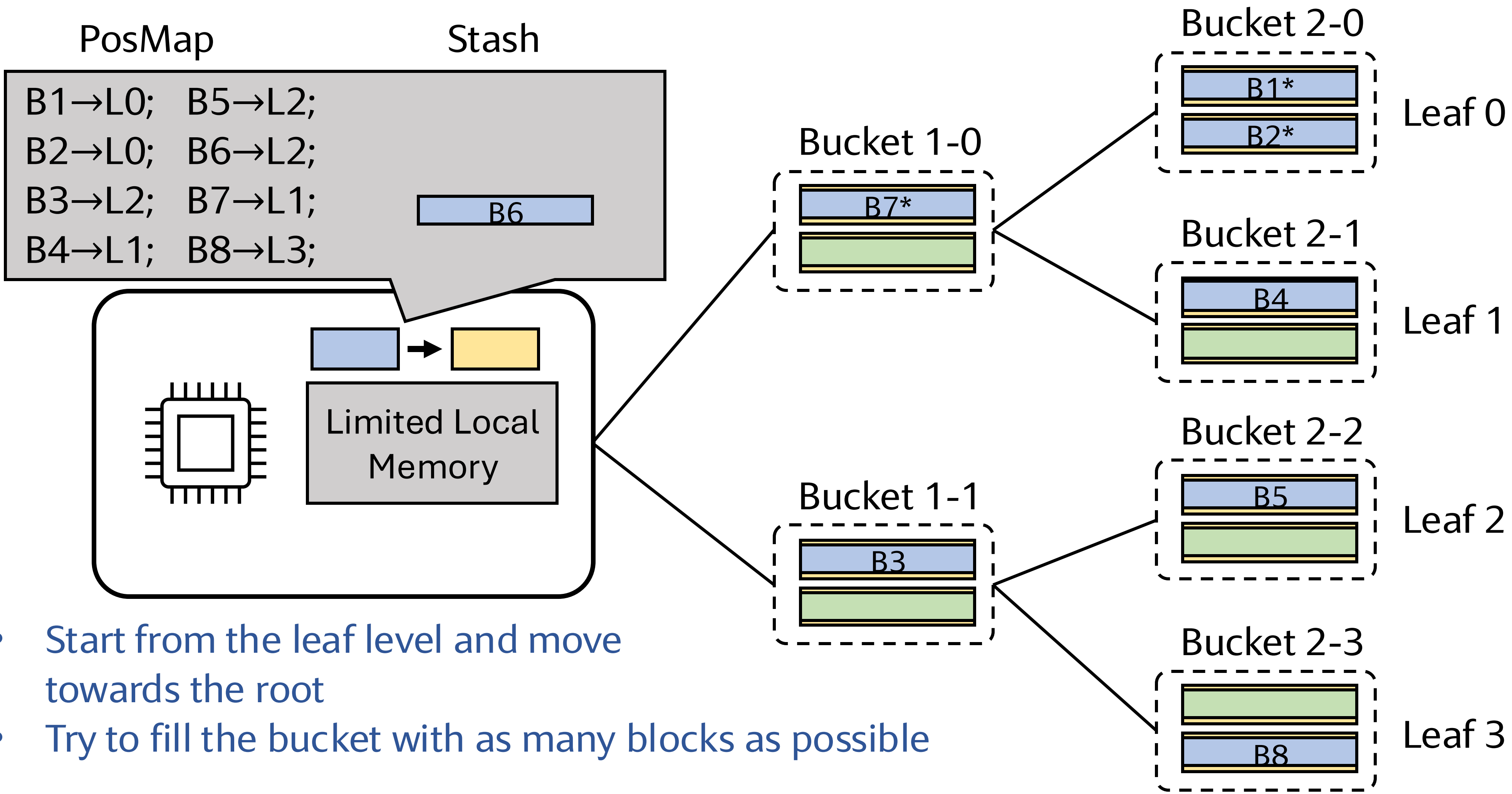
- Start from the leaf level and move towards the root
- Try to fill the bucket with as many blocks as possible

PathORAM: Write back



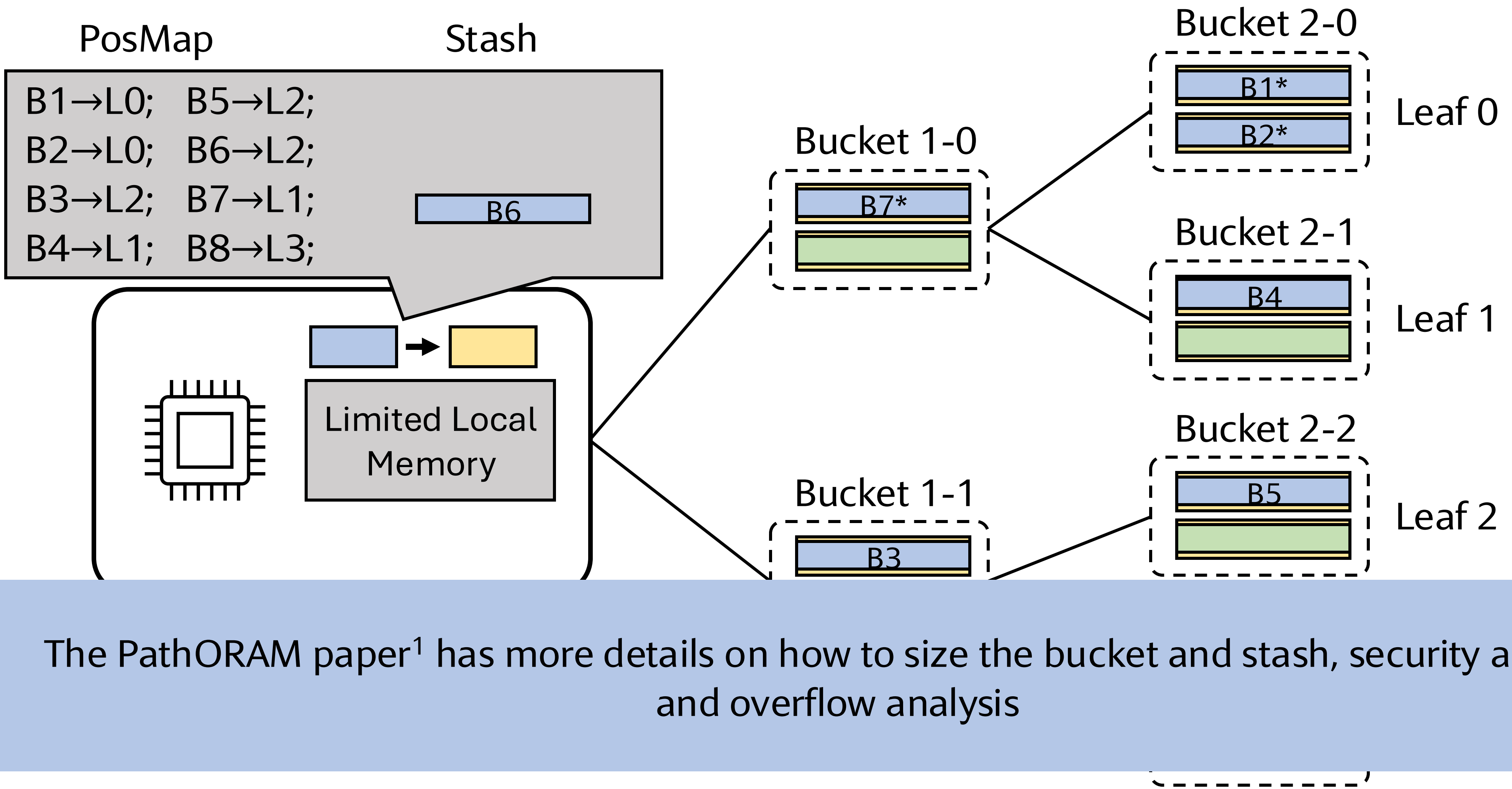
- Start from the leaf level and move towards the root
- Try to fill the bucket with as many blocks as possible

PathORAM: Write back



- Start from the leaf level and move towards the root
- Try to fill the bucket with as many blocks as possible

PathORAM: Write back



The PathORAM paper[1] has more details on how to size the bucket and stash, security analysis, and overflow analysis

[1] Stefanov et al. "Path ORAM: An Extremely Simple Oblivious RAM Protocol"

End of Module 1: Microarchitectural Side Channels

Meltdown

Spectre

Announcements and Reminders

- Proposal due this Wednesday
- Our first paper discussion next week
- Trying something new with the pre-lecture reading.
  - On average, you spent
    - 1.5 hours on each pre-lecture reading
    - 4 hours on each post-lecture reading
  - Will try:
    - Videos
    - Short blogs
    - Suggestions?