MetaLeak: Uncovering Side Channels in Secure Processor Architectures Exploiting Metadata

Md Hafizul Islam Chowdhuryy, Hao Zheng and Fan Yao

Computer Architecture and Systems Research Lab (CASRL)

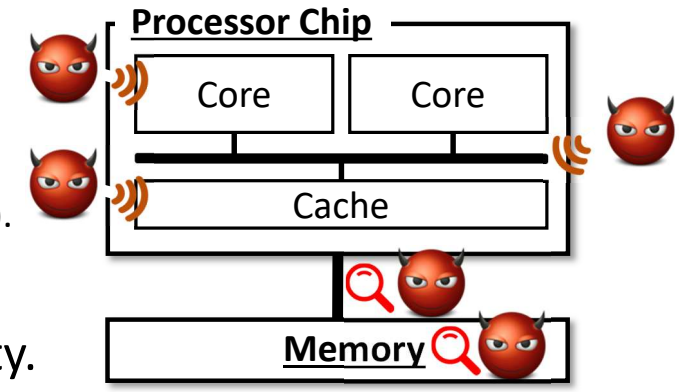
University of Central Florida

Orlando, USA

51st International Symposium on Computer Architecture (**ISCA 2024**)
June 29th – July 3rd, 2024

Microarchitecture Security Problems

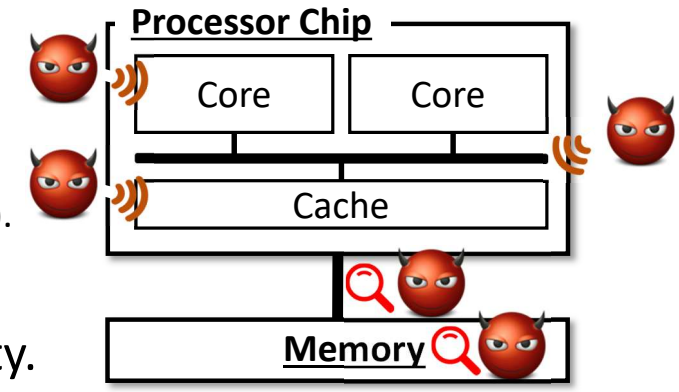
- Two different attack domains are considered:
  - **On-chip:** uArch side channel attacks (e.g., cache attacks).
  - **Off-chip:** Physical attacks (e.g., bus snooping, data tampering attacks).
- Increasing demand for both on-chip and off-chip security.



UCF

Microarchitecture Security Problems

- Two different attack domains are considered:
  - **On-chip:** uArch side channel attacks (e.g., cache attacks).
  - **Off-chip:** Physical attacks (e.g., bus snooping, data tampering attacks).
- Increasing demand for both on-chip and off-chip security.



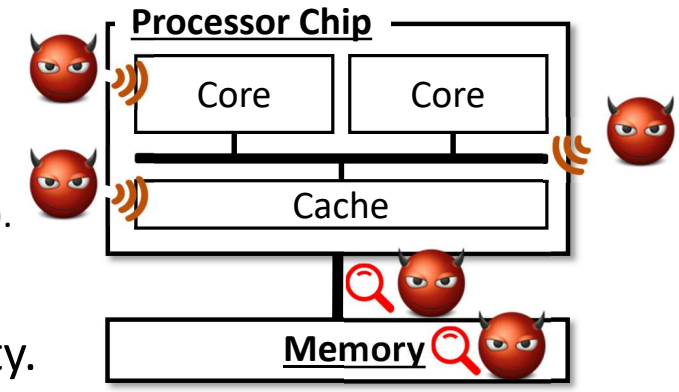
Understanding the interplay between **microarchitecture security** and **secure processor designs** is crucial.

Prior works do not investigate underlying architectural mechanisms in secure processors designs.

UCF

Microarchitecture Security Problems

- Two different attack domains are considered:
  - **On-chip:** uArch side channel attacks (e.g., cache attacks).
  - **Off-chip:** Physical attacks (e.g., bus snooping, data tampering attacks).
- Increasing demand for both on-chip and off-chip security.



Understanding the interplay between **microarchitecture security** and **secure processor designs** is crucial.

Prior works do not investigate underlying architectural mechanisms in secure processors designs.

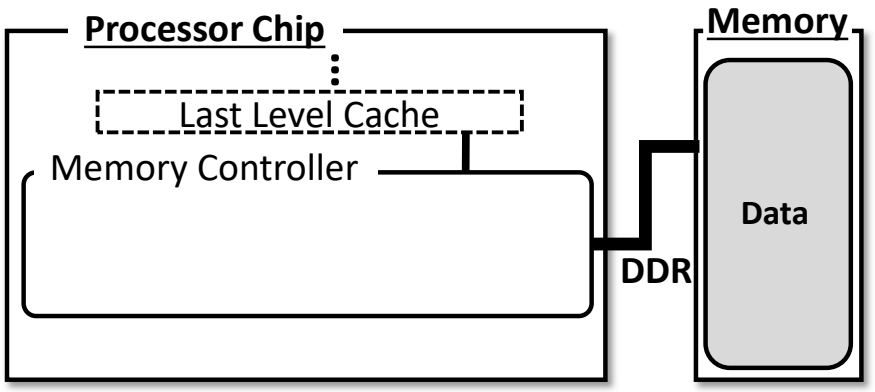
How does the introduction of security mechanism for one threat (i.e., secure processor) reshape the attack surface of another (i.e., uArch side channel)?

UCF

This work

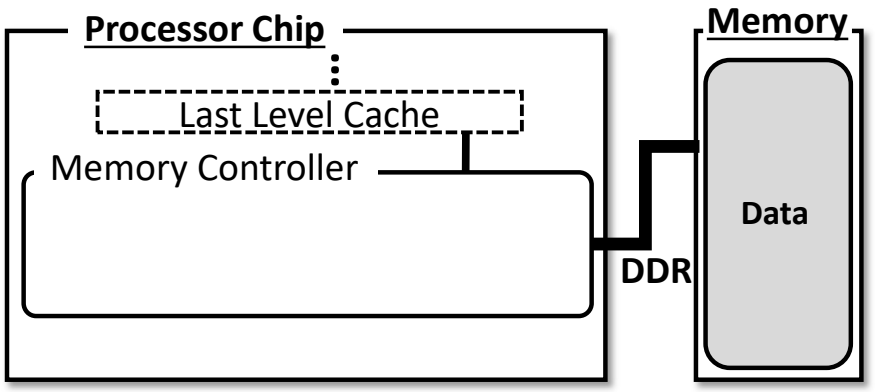
Exploration of *microarchitecture security vulnerabilities* in the ***design space*** *of secure processor architectures*.

Secure Processor Architecture



UCF

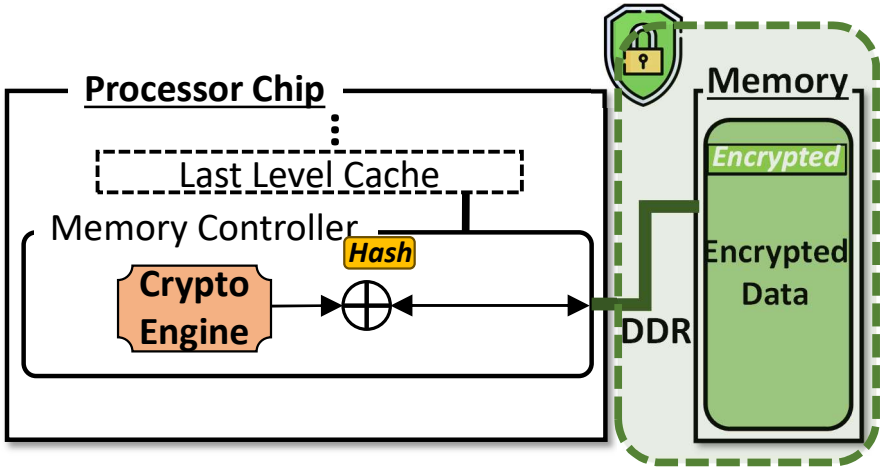
Secure Processor Architecture



Limit trust boundary to the processor chip
Protect confidentiality and integrity of **off-chip data**.

UCF

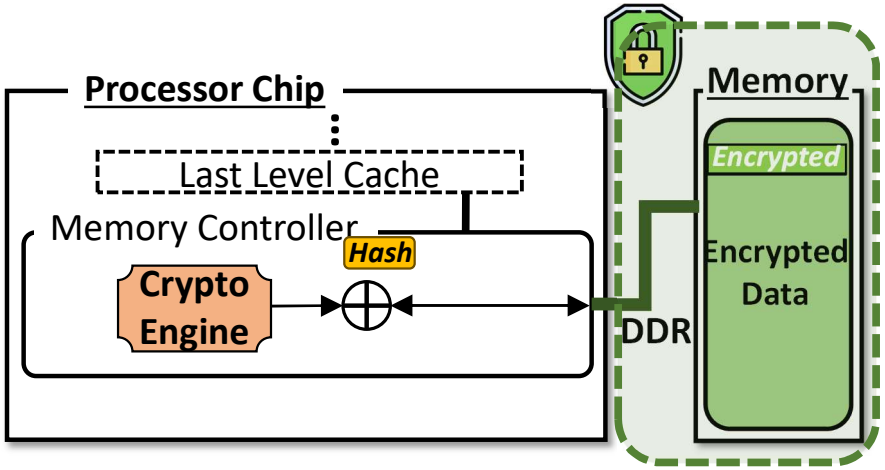
Secure Processor Architecture



Limit trust boundary to the processor chip
Protect confidentiality and integrity of **off-chip data**.

UCF

Secure Processor Architecture

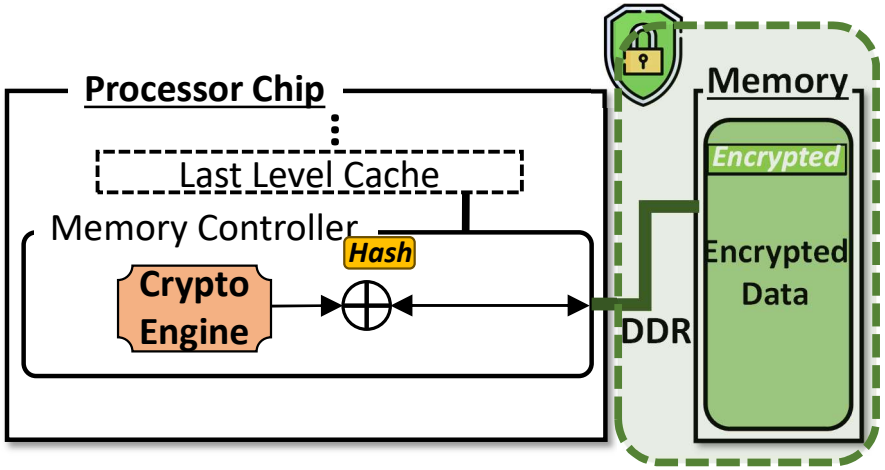


Limit trust boundary to the processor chip
Protect confidentiality and integrity of **off-chip data**.

Best practice → Perform encryption and integrity verification using processor-maintained metadata.

UCF

Secure Processor Architecture



Limit trust boundary to the processor chip
Protect confidentiality and integrity of **off-chip data**.

Best practice → Perform encryption and integrity verification using processor-maintained metadata.

Security impact of supporting secure processor architectures on **uArch side channel leakage** is not well understood.

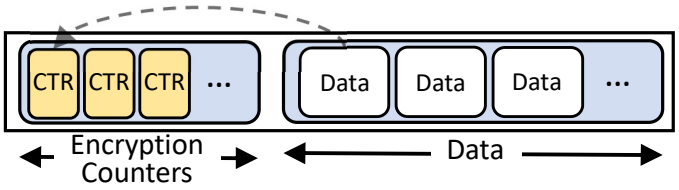
UCF

Design Space of Memory Encryption

- **Memory Encryption:** Typically performed using block ciphers.
  - **AES Counter-mode encryption** is the predominant choice.
  - Requires **per-block counter** (typically stored in memory).

Design Space of Memory Encryption

- **Memory Encryption:** Typically performed using block ciphers.
  - **AES Counter-mode encryption** is the predominant choice.
  - Requires **per-block counter** (typically stored in memory).

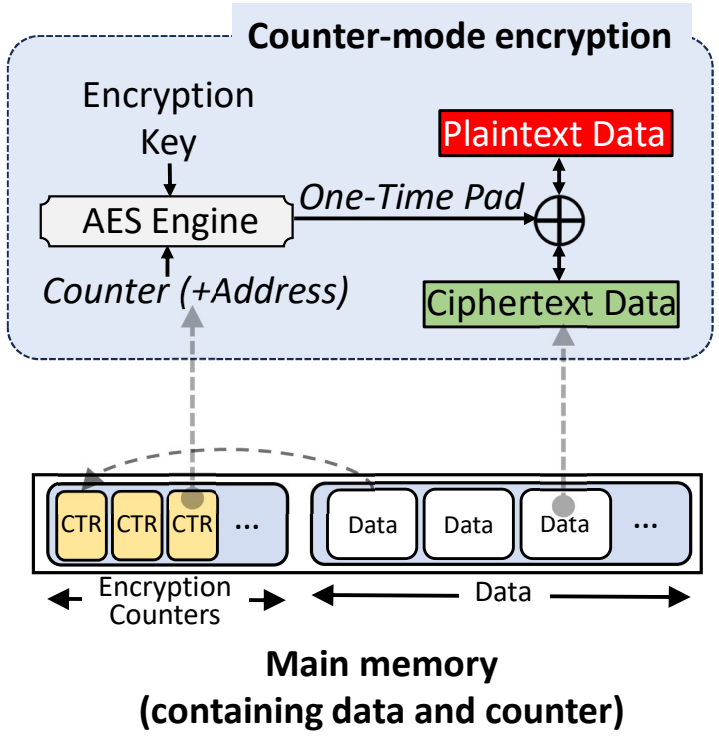


Main memory (containing data and counter)

UCF

Design Space of Memory Encryption

- **Memory Encryption:** Typically performed using block ciphers.
  - **AES Counter-mode encryption** is the predominant choice.
  - Requires **per-block counter** (typically stored in memory).

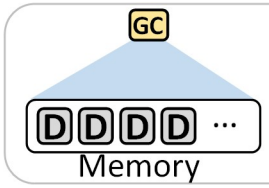


UCF

Design Space of Memory Encryption

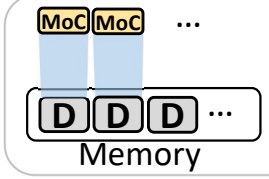
- **Memory Encryption:** Typically performed using block ciphers.
  - **AES Counter-mode encryption** is the predominant choice.
  - Requires **per-block counter** (typically stored in memory).

Different counter-mode encryption schemes:



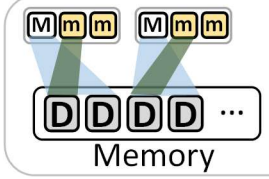
a) Global Counter (GC):

One counter *shared among all memory blocks*.



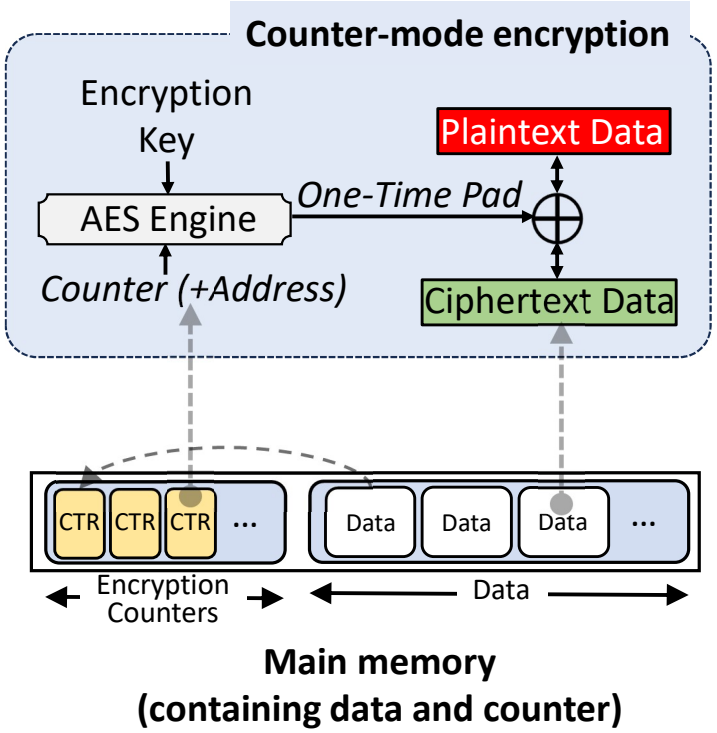
b) Monolithic Counter (MoC):

One counter *per memory block*.



c) Split Counter: Major (M) + *per-block* Minor (m)

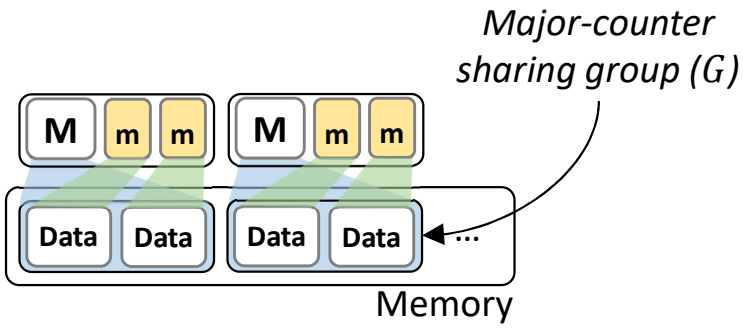
M is per-group (group of blocks), m is per-block.



UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



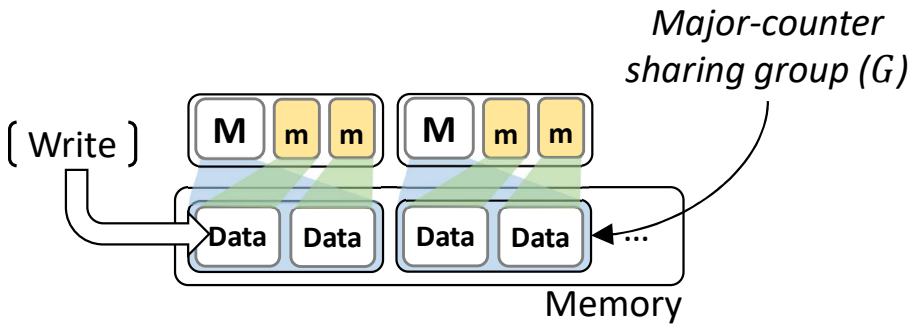
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



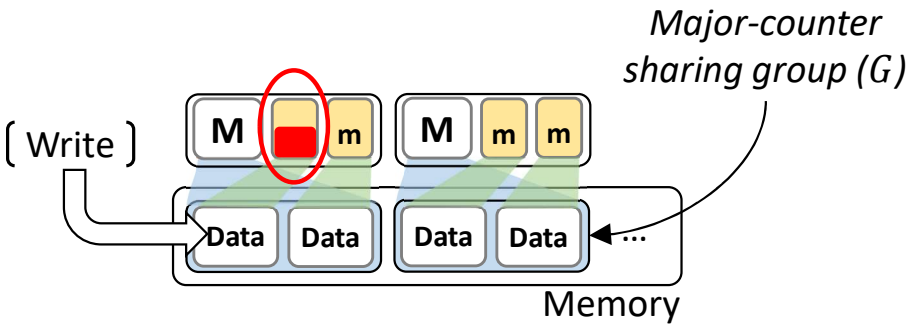
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



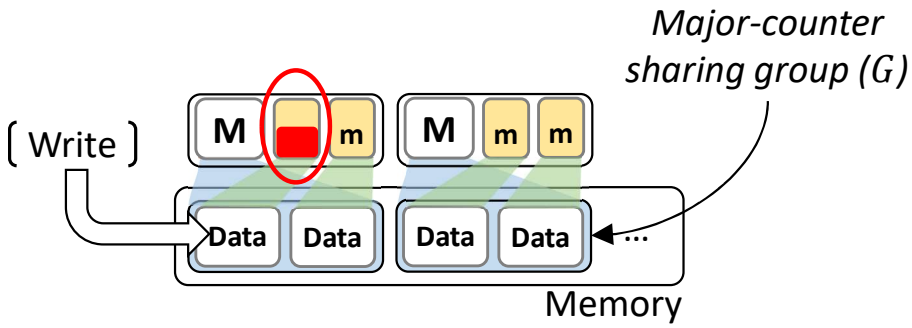
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



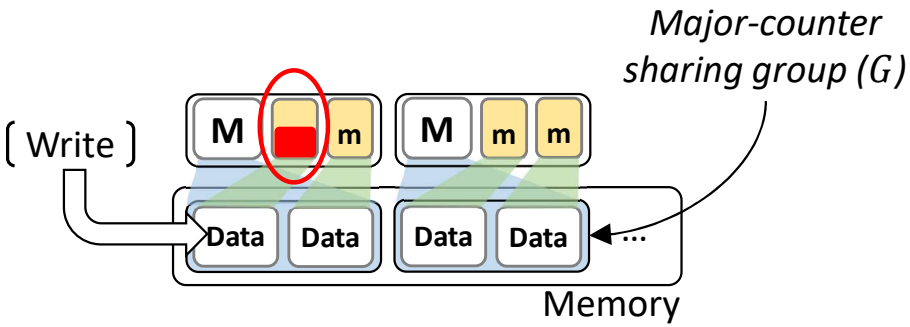
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



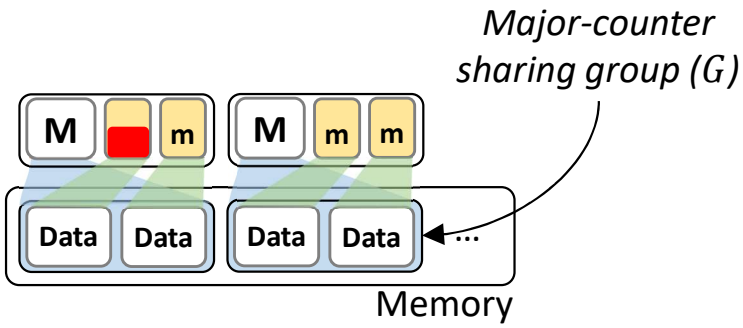
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



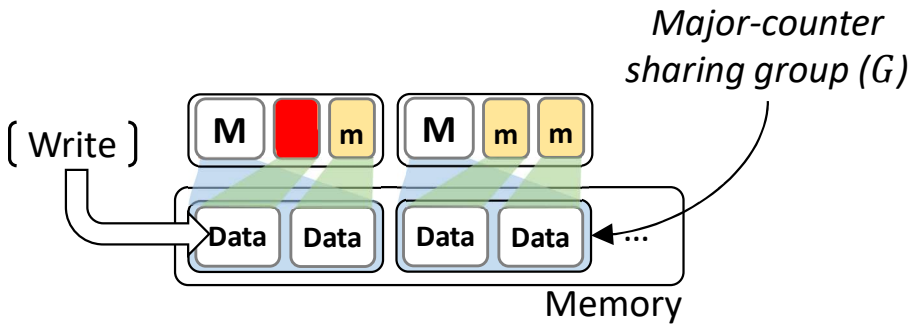
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



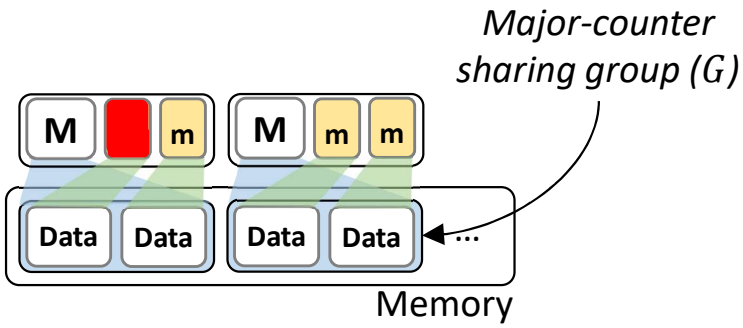
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



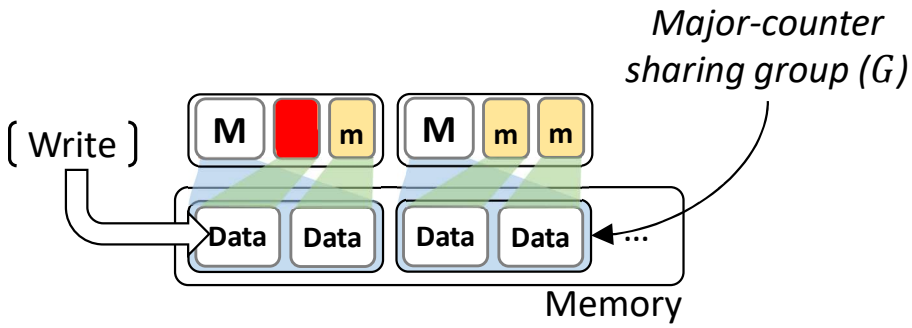
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



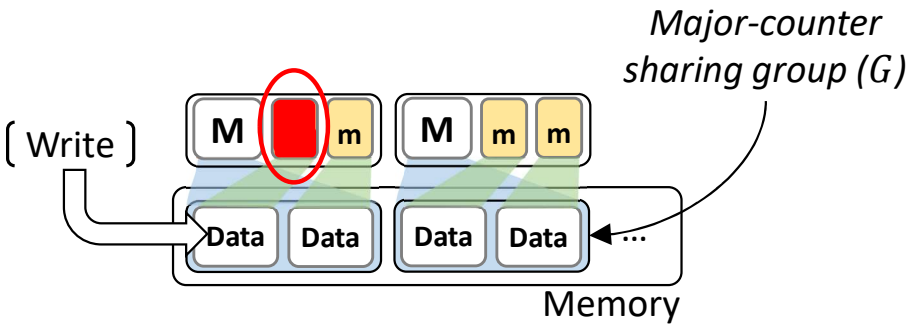
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



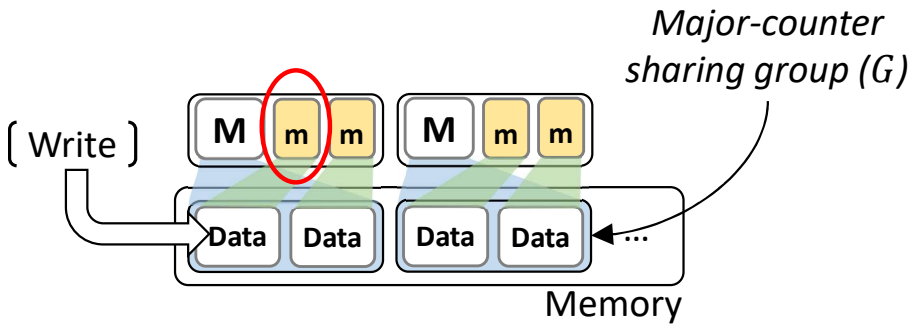
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



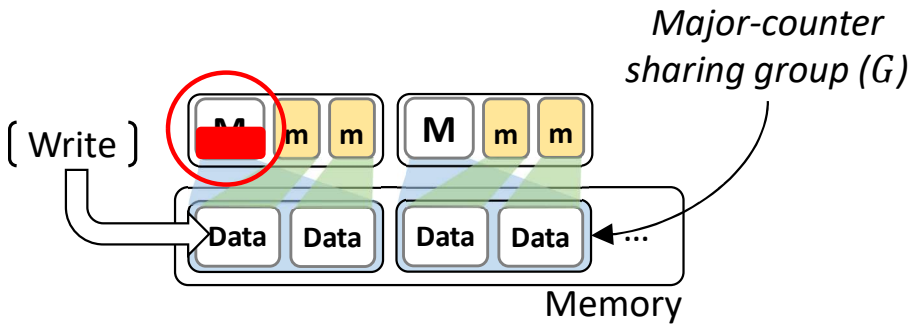
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



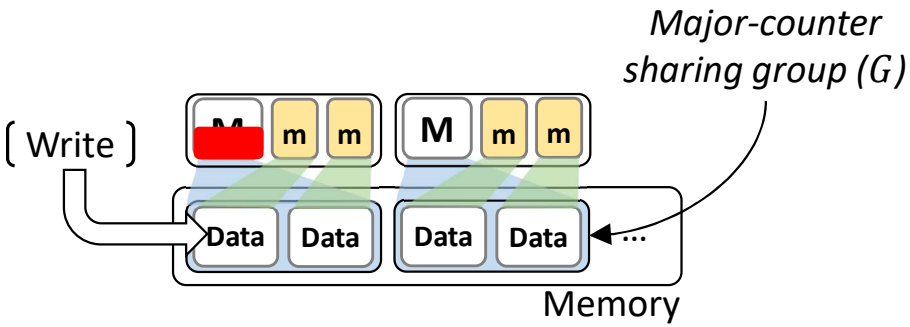
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



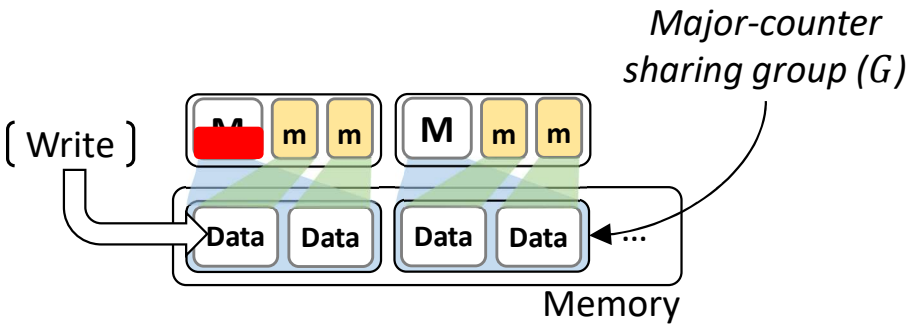
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



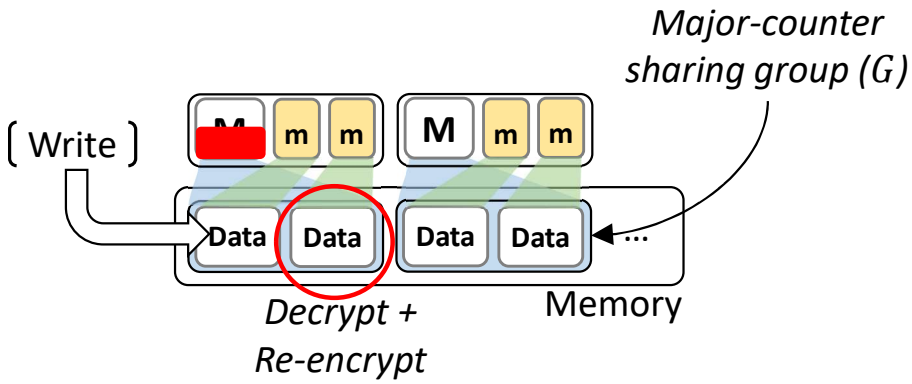
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



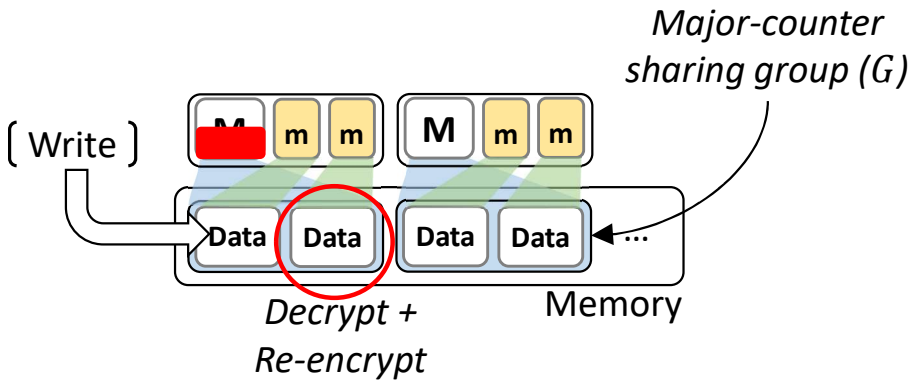
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



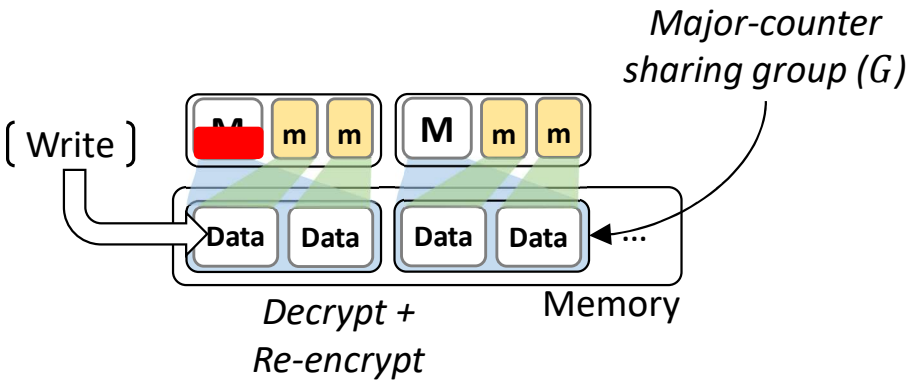
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



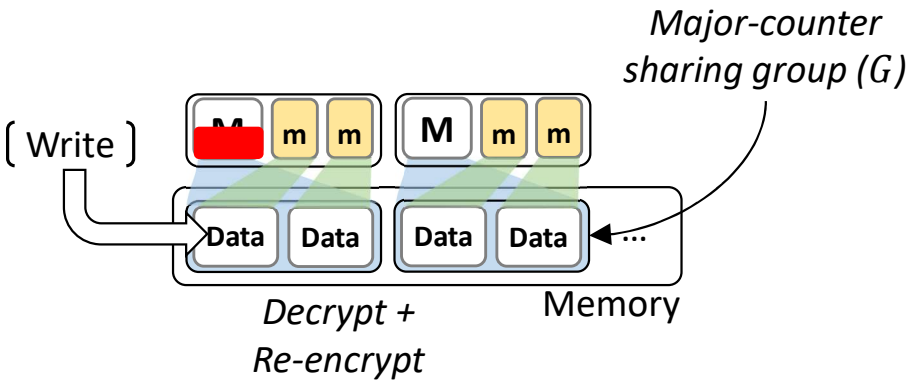
Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```


UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



Algorithm: Counter-mode Encryption Mechanism

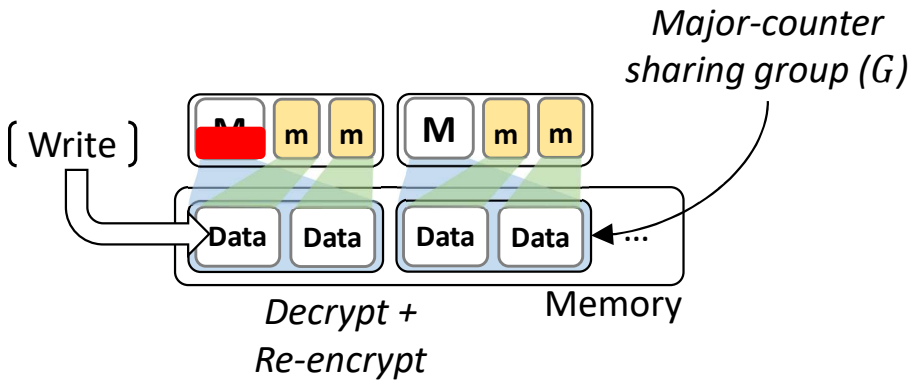
```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```

Counter overflow → requires *re-encryption of the entire counter-sharing group*.

UCF

Timing Vulnerabilities in Memory Encryption

- **Split Counter:** Major (M) + *per-block* Minor (m).



Algorithm: Counter-mode Encryption Mechanism

```
Input: P_t: current block to encrypt
Function Encrypt(P_t):
    ctr_old = ctr
    Increment (ctr)                    // Increment the counter
    if ctr_old = ctr^max then          // Overflow detected
        // Re-encrypt memory blocks in group
        for P_i in {G − P_t} do
            Decrypt(P_i) with old counter
            Encrypt(P_i) with new counter
        Encrypt(P_t) using ctr
    else
        Encrypt(P_t) using ctr
```

Counter overflow → requires *re-encryption of the entire counter-sharing group*.

Vulnerability Class-1

Encryption counter creates *metadata state dependent* execution paths:

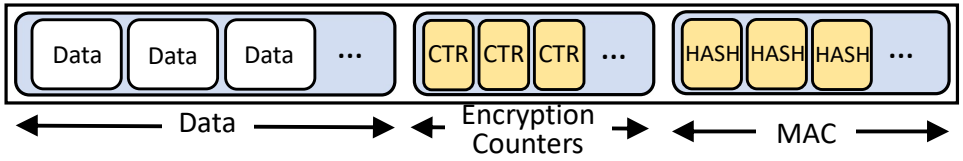
1. **Slower:** Program data write leading to **counter overflow**.
2. **Faster:** Regular program data write (not triggering counter overflow).

UCF

Design Space of Memory Data Authentication

- **Memory Data Authentication:** Typically performed through keyed hash.
- **Different MAC authentication schemes:**
  - Computed over data-only: $\mathcal{M} = MAC_k(C, addr)$; C = ciphertext block, $addr$ = block address.
  - Computed by pairing data and counter: $\mathcal{M} = MAC_k(C, ctr, addr)$; ctr = counter.

- MAC cannot prevent **data replay**!

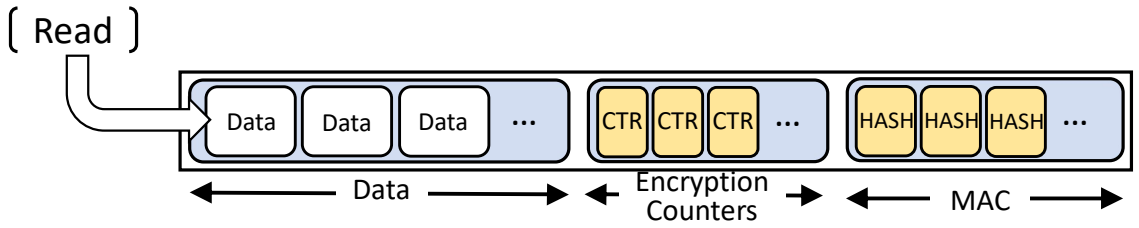


UCF

Design Space of Memory Data Authentication

- **Memory Data Authentication:** Typically performed through keyed hash.
- **Different MAC authentication schemes:**
  - Computed over data-only: $\mathcal{M} = MAC_k(C, addr)$; C = ciphertext block, $addr$ = block address.
  - Computed by pairing data and counter: $\mathcal{M} = MAC_k(C, ctr, addr)$; ctr = counter.

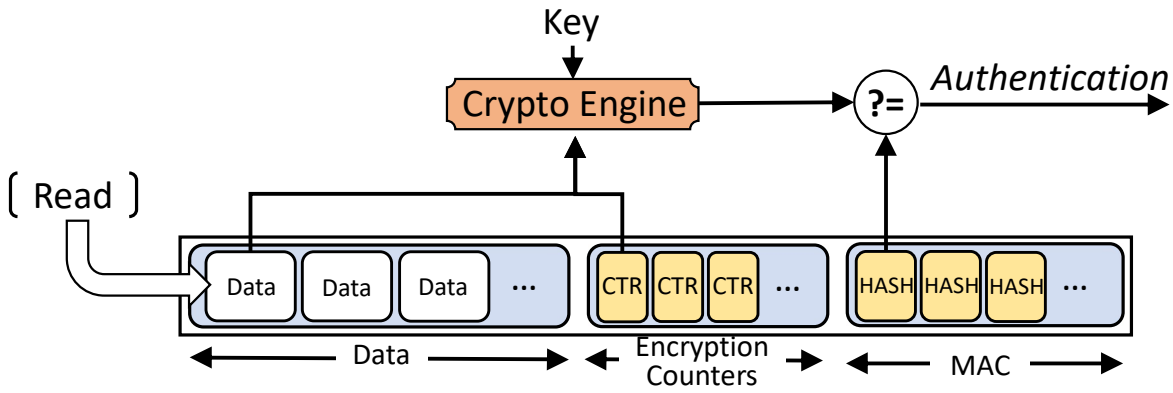
- MAC cannot prevent **data replay**!



UCF

Design Space of Memory Data Authentication

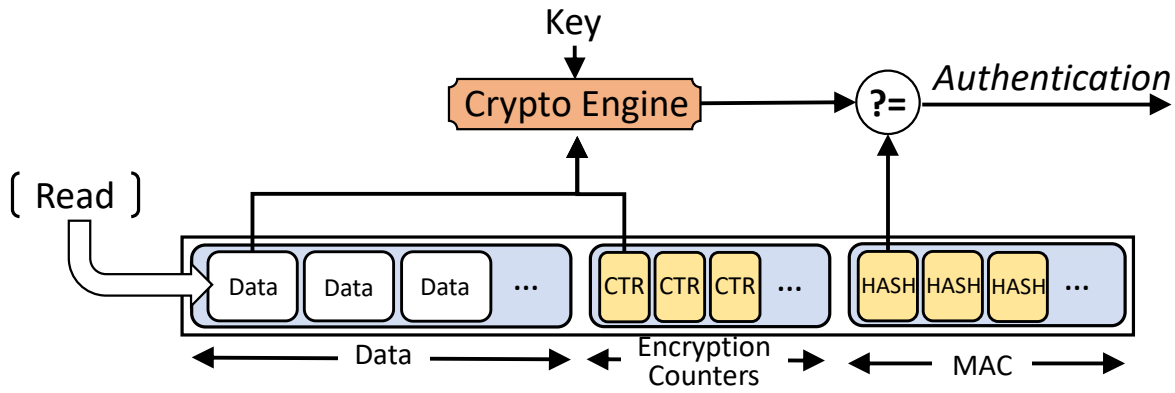
- **Memory Data Authentication:** Typically performed through keyed hash.
- **Different MAC authentication schemes:**
  - Computed over data-only: $\mathcal{M} = MAC_k(C, addr)$; C = ciphertext block, $addr$ = block address.
  - Computed by pairing data and counter: $\mathcal{M} = MAC_k(C, ctr, addr)$; ctr = counter.
- MAC cannot prevent **data replay**!



UCF

Design Space of Memory Data Authentication

- **Memory Data Authentication:** Typically performed through keyed hash.
- **Different MAC authentication schemes:**
  - Computed over data-only: $\mathcal{M} = MAC_k(C, addr)$; C = ciphertext block, $addr$ = block address.
  - Computed by pairing data and counter: $\mathcal{M} = MAC_k(C, ctr, addr)$; ctr = counter.
- MAC cannot prevent **data replay**!

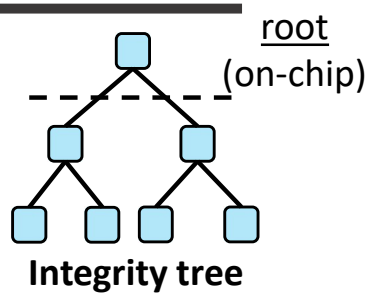


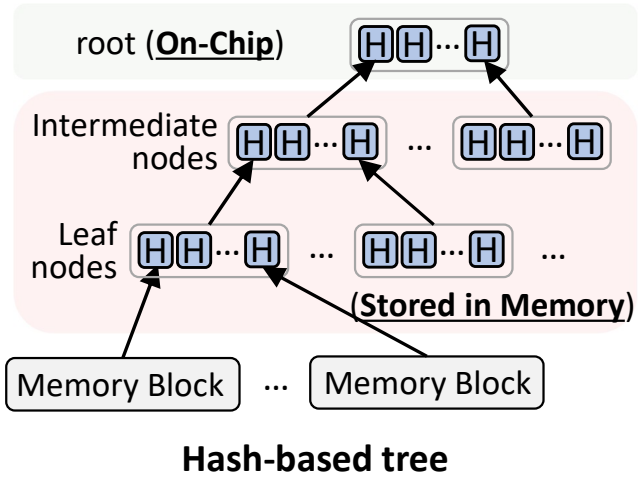
Memory authentication using MAC **does not result in variable execution latency differences**. Agnostic of program or secure memory metadata access patterns.

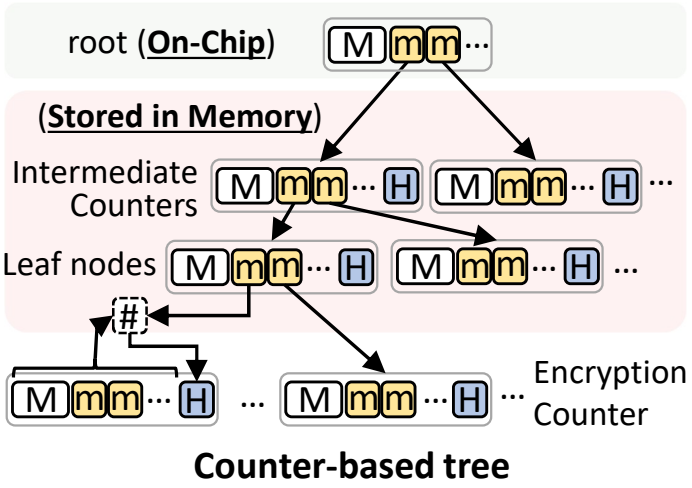
UCF

Design Space of Integrity Verification

- **Memory Data Integrity Protection:** Typically performed using **Integrity Tree**.
  - **Hash-based tree:** Each node in tree is a *hash* of its child nodes.
  - **Counter-based tree:** Each node contains *write counters* for its child nodes.



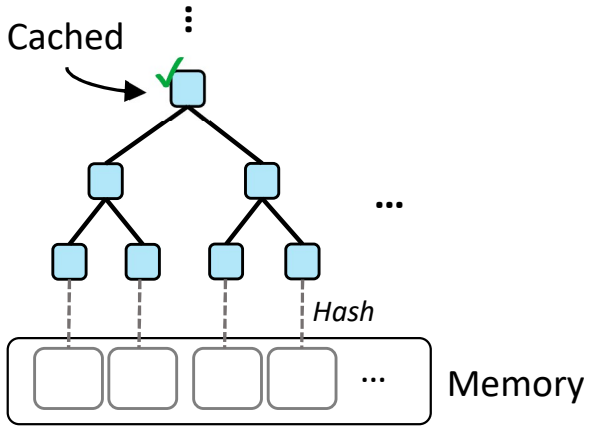




UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



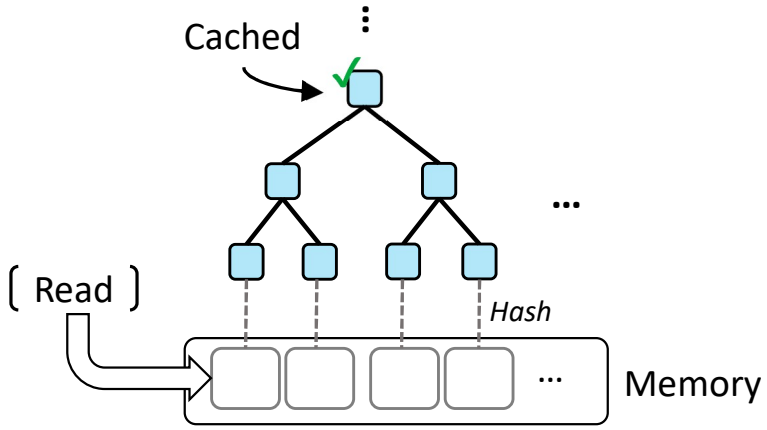
Algorithm: Memory Integrity Verification Mechanism

Input: $\mathcal{B}$: // Memory block to verify
Leaf node: N_A^i // The i^{th} −level ancestor tree node for an attached memory block $\mathcal{B}$
Function Verify($\mathcal{B}$):
  // Assume block N_A^L is cached
  for i *from* $\{1 - L\}$ **do**
    Load Block(N_A^i)
    Verify (Block(N_A^{i-1})) with N_A^i
  Verify($\mathcal{B}$) with N_A^o

UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



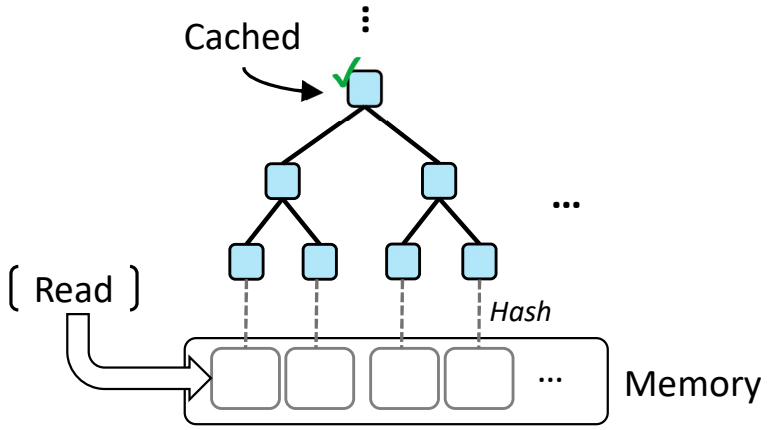
Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
       node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 − L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i−1))) with N_A^i
    Verify(B) with N_A^o
```


UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



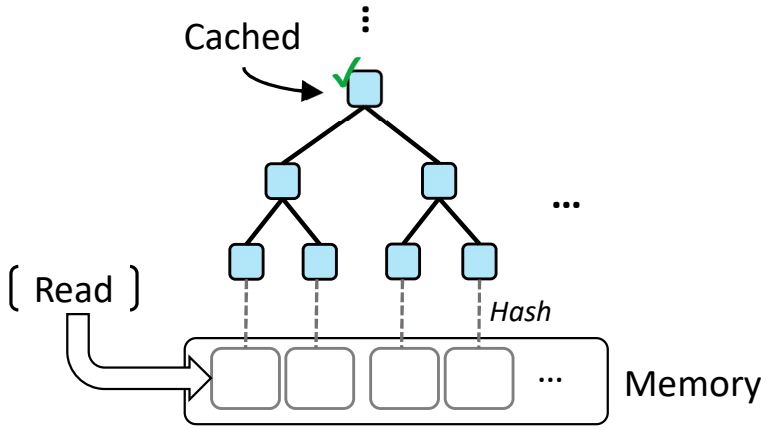
Algorithm: Memory Integrity Verification Mechanism

Input: $\mathcal{B}$: // Memory block to verify
Leaf node: N_A^i // The i^{th} −level ancestor tree node for an attached memory block $\mathcal{B}$
Function Verify($\mathcal{B}$):
  // Assume block N_A^L is cached
  for i *from* $\{1 - L\}$ **do**
    Load Block(N_A^i)
    Verify (Block(N_A^{i-1})) with N_A^i
  Verify($\mathcal{B}$) with N_A^o

UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



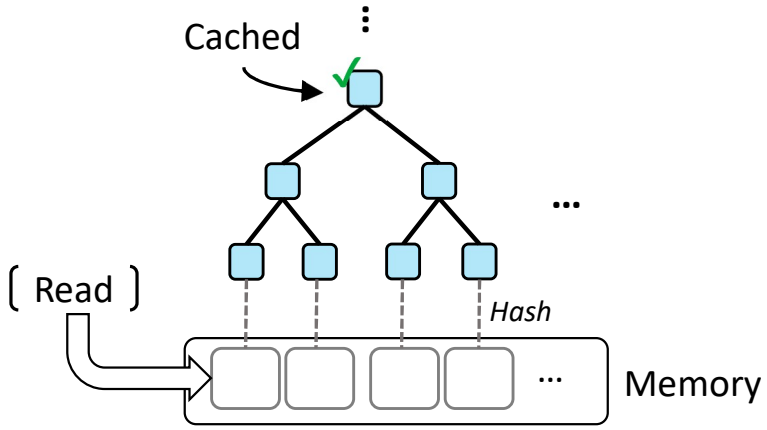
Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
        node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 – L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i-1))) with N_A^i
    Verify(B) with N_A^o
```


UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



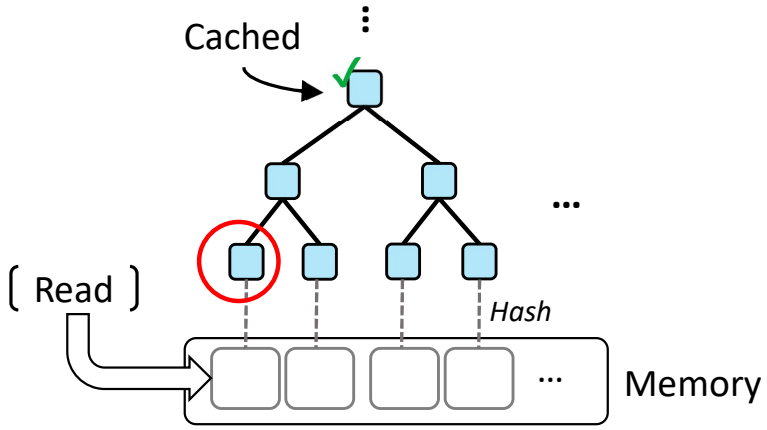
Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
        node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 − L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i−1))) with N_A^i
    Verify(B) with N_A^o
```


UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



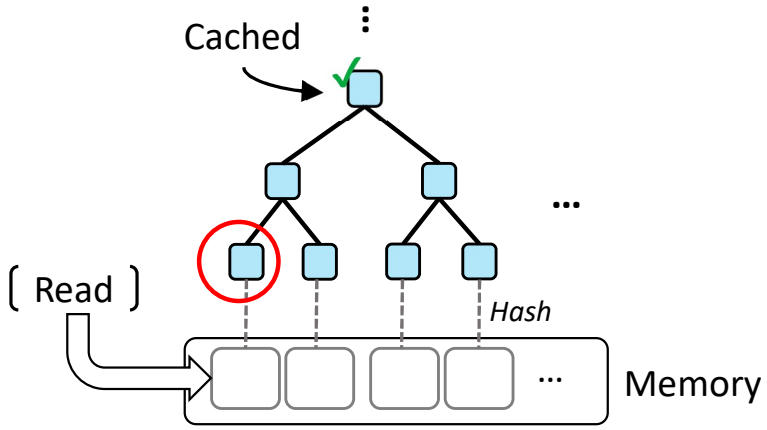
Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
        node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 − L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i−1))) with N_A^i
    Verify(B) with N_A^o
```


UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



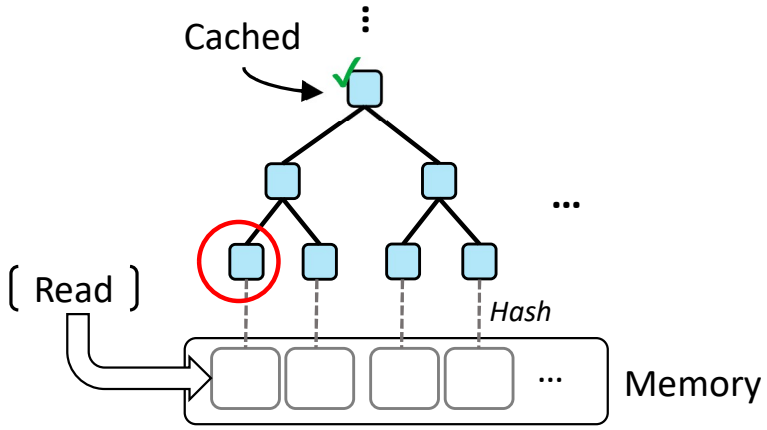
Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
       node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 − L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i−1))) with N_A^i
    Verify(B) with N_A^o
```


UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



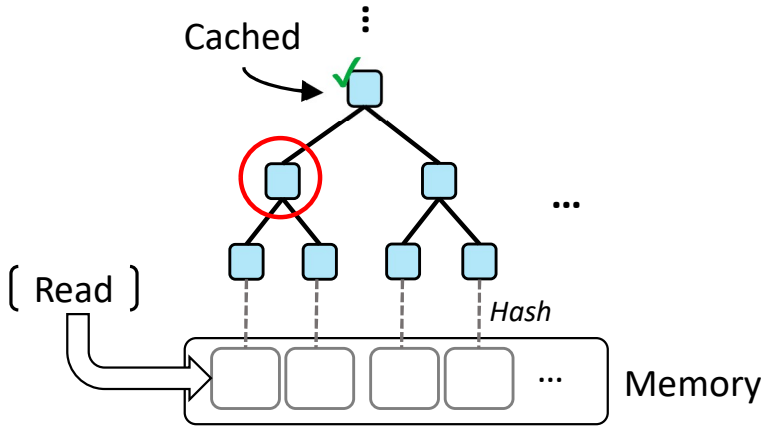
Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th −level ancestor tree
        node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 − L} do
        Load Block(N_A^i)
        Verify (Block(N_A^{i−1})) with N_A^i
    Verify(B) with N_A^o
```


UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



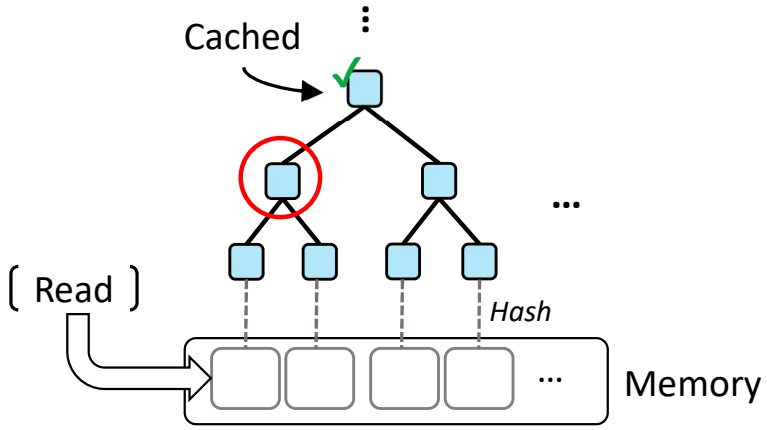
Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
        node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 – L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i-1))) with N_A^i
    Verify(B) with N_A^o
```


UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



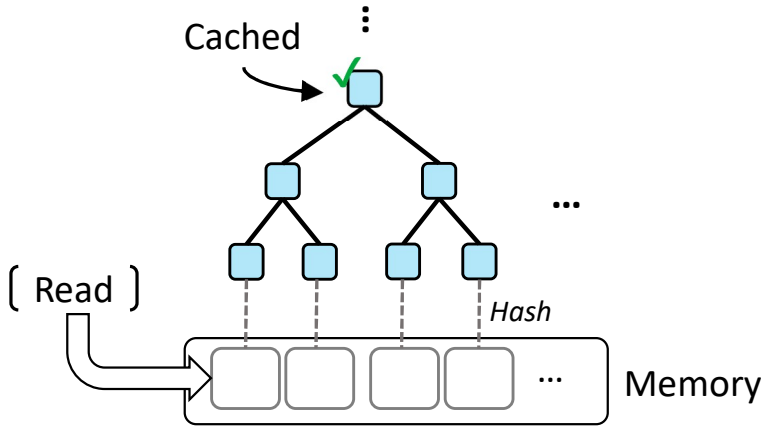
Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
        node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 − L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i−1))) with N_A^i
    Verify(B) with N_A^o
```


UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



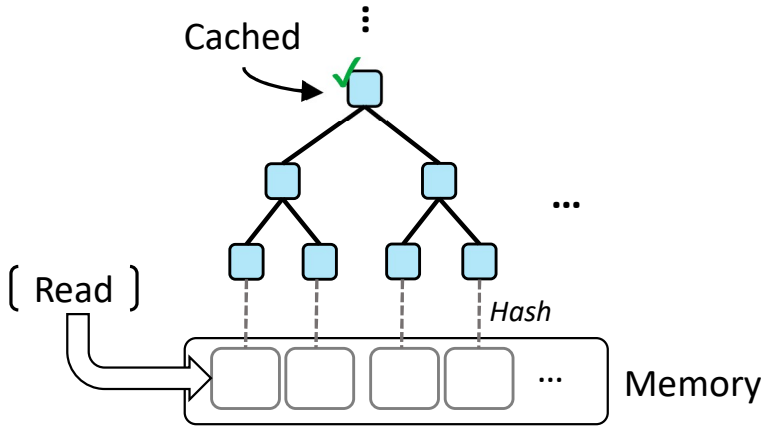
Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
        node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 − L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i−1))) with N_A^i
    Verify(B) with N_A^o
```


UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



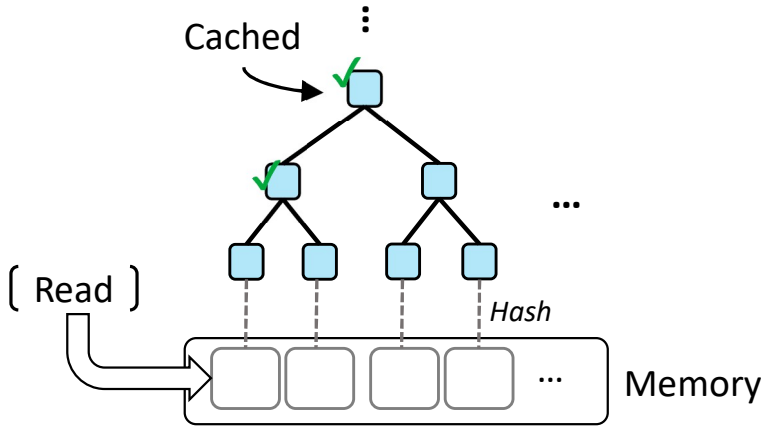
Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
        node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 − L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i−1))) with N_A^i
    Verify(B) with N_A^o
```


UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



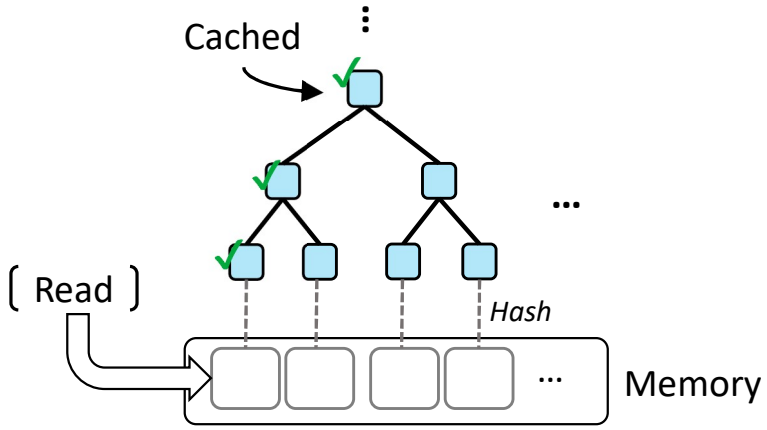
Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
       node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 − L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i−1))) with N_A^i
    Verify(B) with N_A^o
```


UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



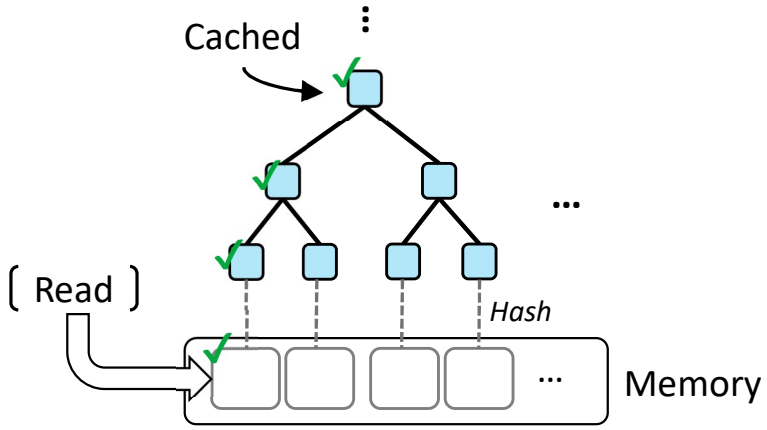
Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
        node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 − L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i−1))) with N_A^i
    Verify(B) with N_A^o
```


UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



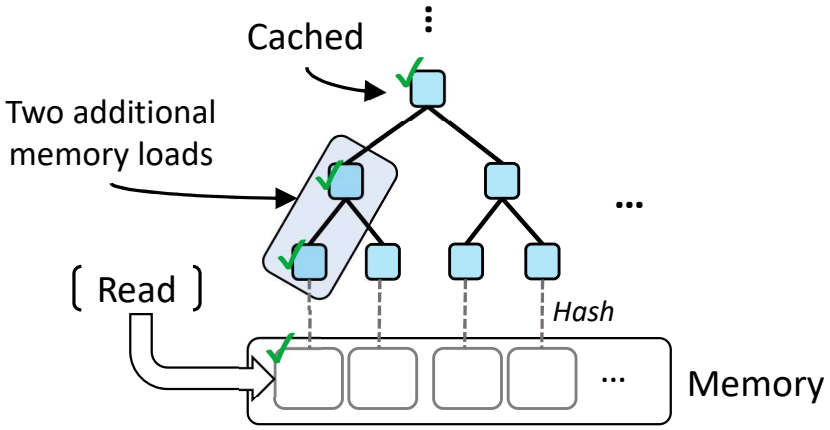
Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
        node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 − L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i−1))) with N_A^i
    Verify(B) with N_A^o
```


UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**

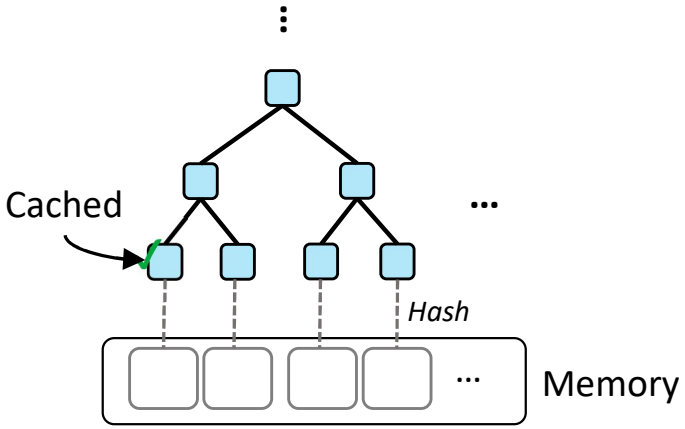


Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
        node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 − L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i−1))) with N_A^i
    Verify(B) with N_A^o
```

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



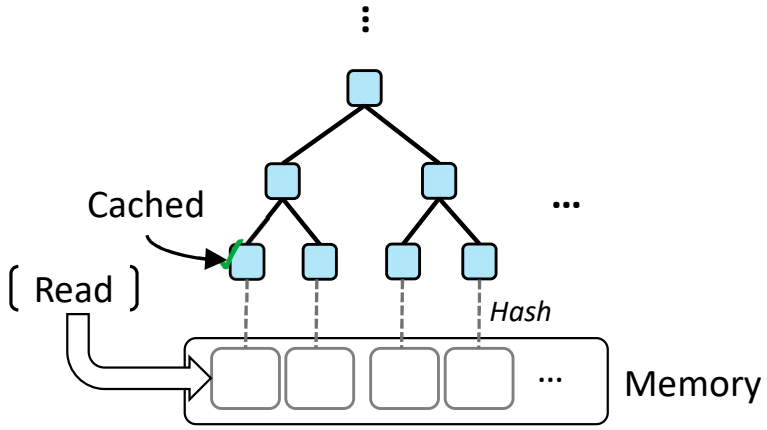
Algorithm: Memory Integrity Verification Mechanism

Input: $\mathcal{B}$: // Memory block to verify
Leaf node: N_A^i // The i^{th} −level ancestor tree node for an attached memory block $\mathcal{B}$
Function Verify($\mathcal{B}$):
  // Assume block N_A^L is cached
  for i *from* $\{1 - L\}$ **do**
    Load Block(N_A^i)
    Verify (Block(N_A^{i-1})) with N_A^i
  Verify($\mathcal{B}$) with N_A^o

UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**

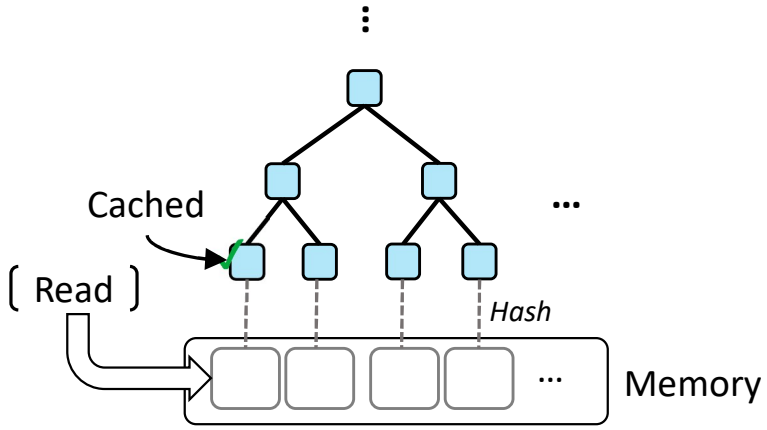


Algorithm: Memory Integrity Verification Mechanism

Input: $\mathcal{B}$: // Memory block to verify
Leaf node: N_A^i // The i^{th} −level ancestor tree node for an attached memory block $\mathcal{B}$
Function Verify($\mathcal{B}$):
  // Assume block N_A^L is cached
  for i *from* $\{1 - L\}$ **do**
    Load Block(N_A^i)
    Verify (Block(N_A^{i-1})) with N_A^i
  Verify($\mathcal{B}$) with N_A^o

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



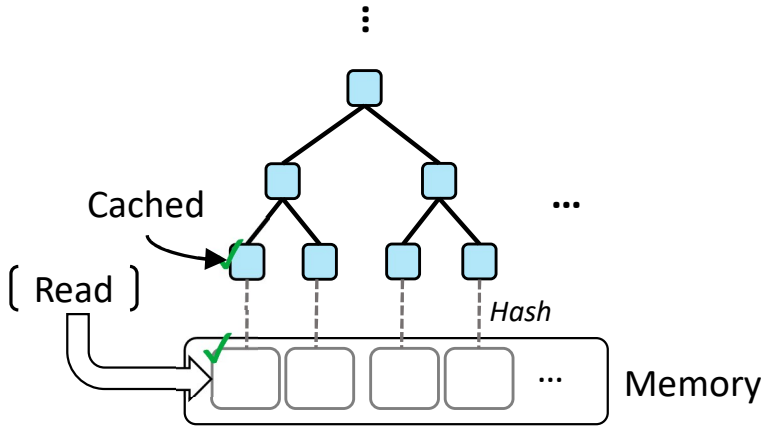
Algorithm: Memory Integrity Verification Mechanism

Input: $\mathcal{B}$: // Memory block to verify
Leaf node: N_A^i // The i^{th} −level ancestor tree node for an attached memory block $\mathcal{B}$
Function Verify($\mathcal{B}$):
  // Assume block N_A^L is cached
  for i *from* $\{1 - L\}$ **do**
    Load Block(N_A^i)
    Verify (Block(N_A^{i-1})) with N_A^i
  Verify($\mathcal{B}$) with N_A^o

UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



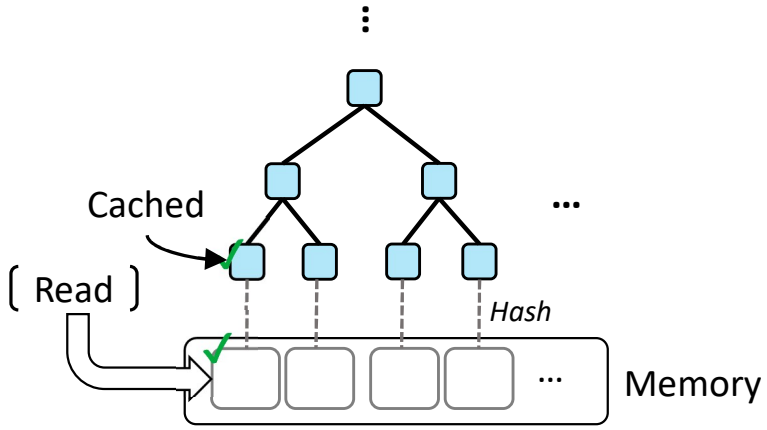
Algorithm: Memory Integrity Verification Mechanism

Input: $\mathcal{B}$: // Memory block to verify
Leaf node: N_A^i // The i^{th} −level ancestor tree node for an attached memory block $\mathcal{B}$
Function Verify($\mathcal{B}$):
  // Assume block N_A^L is cached
  for i *from* $\{1 - L\}$ **do**
    Load Block(N_A^i)
    Verify (Block(N_A^{i-1})) with N_A^i
  Verify($\mathcal{B}$) with N_A^o

UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



Algorithm: Memory Integrity Verification Mechanism

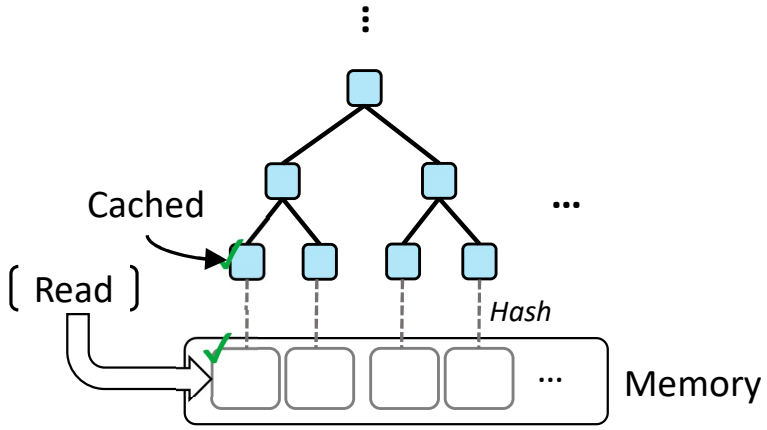
```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
        node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 − L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i−1))) with N_A^i
    Verify(B) with N_A^o
```

- Integrity verification latency path varies according to ***tree node caching state***.

UCF

Timing Vulnerabilities in Integrity Verification

- **Integrity tree verification operation:**



Algorithm: Memory Integrity Verification Mechanism

```
Input: B: // Memory block to verify
Leaf node: N_A^i // The i^th –level ancestor tree
        node for an attached memory block B
Function Verify(B):
    // Assume block N_A^L is cached
    for i from {1 − L} do
        Load Block(N_A^i)
        Verify (Block(N_A^(i−1))) with N_A^i
    Verify(B) with N_A^o
```

- Integrity verification latency path varies according to ***tree node caching state***.

Vulnerability Class-2

Integrity tree is global structure → Any block shares ***at least one node*** with any other block. Adversary can exploit the ***timing of secret-dependent integrity metadata access***.

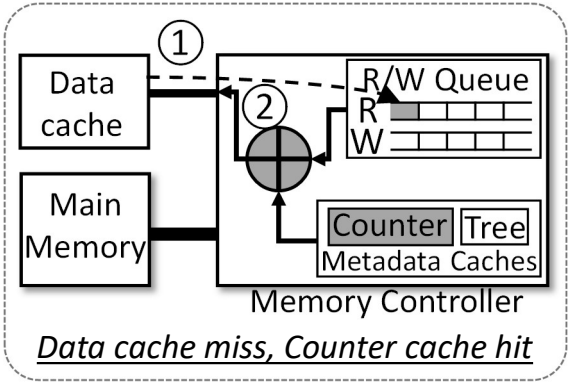
UCF

Timing Characterization: Reads

Processor data read requests follow *different access paths* under *different metadata caching states*.

Timing Characterization: Reads

Processor data read requests follow *different access paths* under *different metadata caching states*.

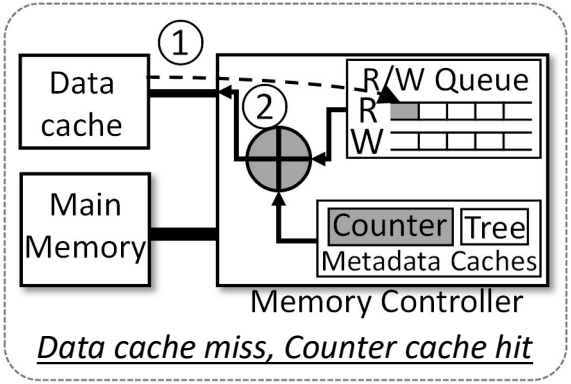


Data cache miss, Counter cache hit

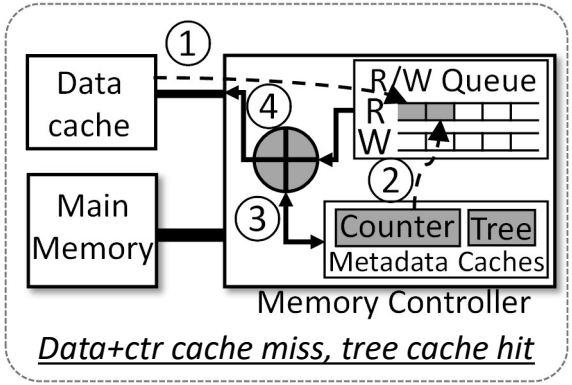
UCF

Timing Characterization: Reads

Processor data read requests follow *different access paths* under *different metadata caching states*.



Data cache miss, Counter cache hit

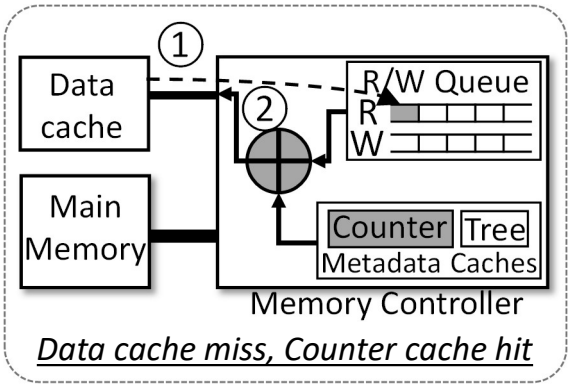


Data+ctr cache miss, tree cache hit

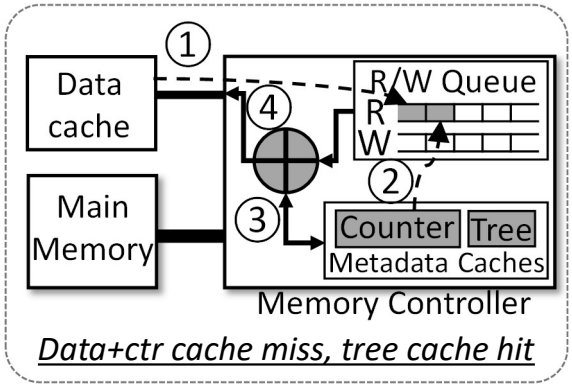
UCF

Timing Characterization: Reads

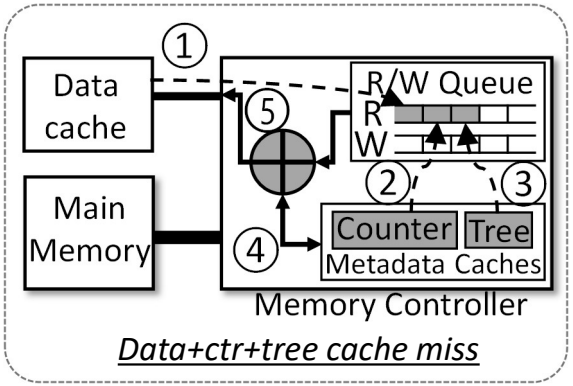
Processor data read requests follow *different access paths* under *different metadata caching states*.



Data cache miss, Counter cache hit



Data+ctr cache miss, tree cache hit

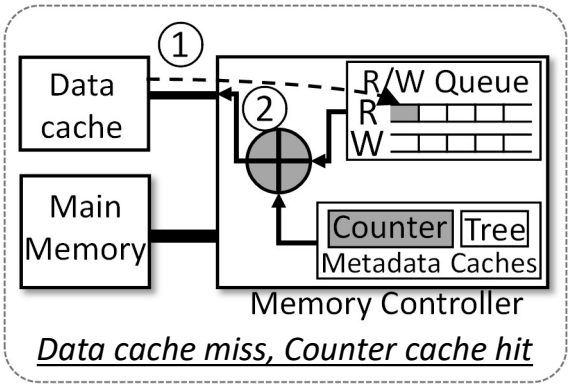


Data+ctr+tree cache miss

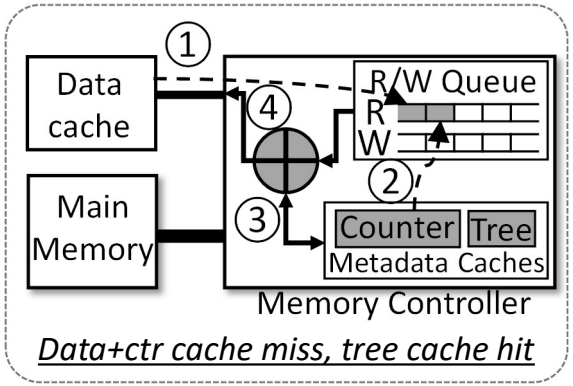
UCF

Timing Characterization: Reads

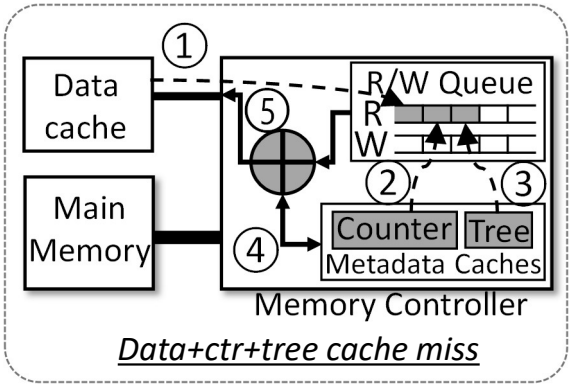
Processor data read requests follow *different access paths* under *different metadata caching states*.



Data cache miss, Counter cache hit

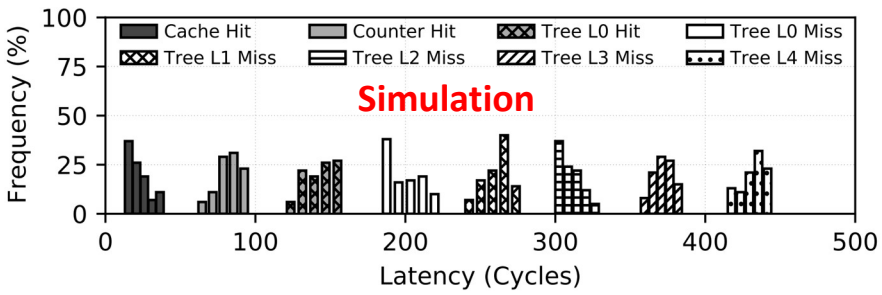


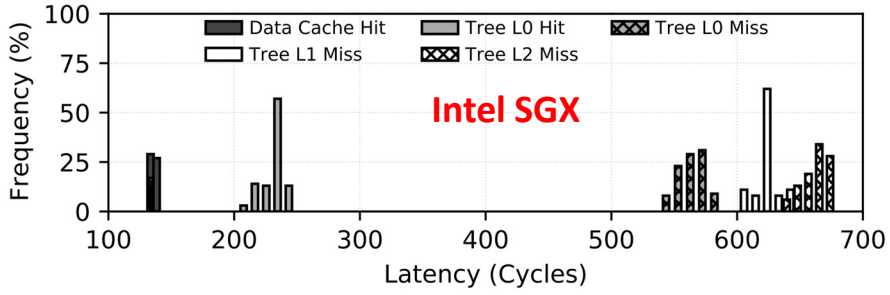
Data+ctr cache miss, tree cache hit



Data+ctr+tree cache miss

Latency distribution across access paths



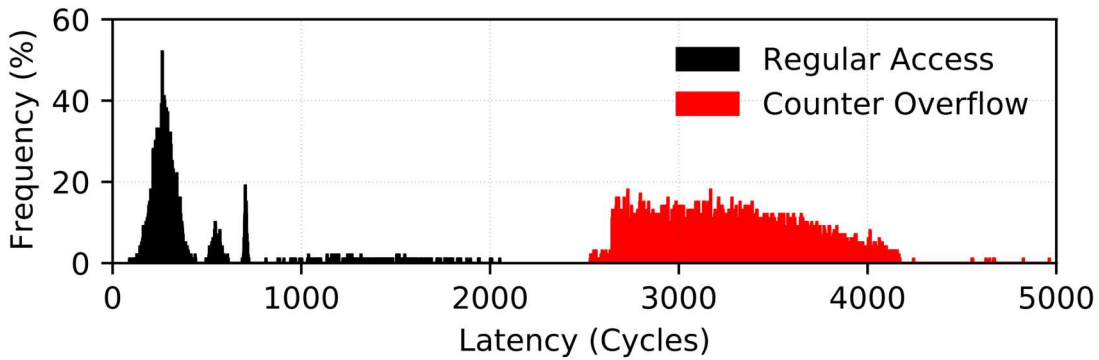


Different paths exhibit **highly distinguishable access latencies**.

UCF

Timing Characterizations: Writes

- Processor data writes can exhibit the same set of variations due to caching status of security metadata.
- Different from reads, write operations increment encryption counter (+ Counters in integrity tree).

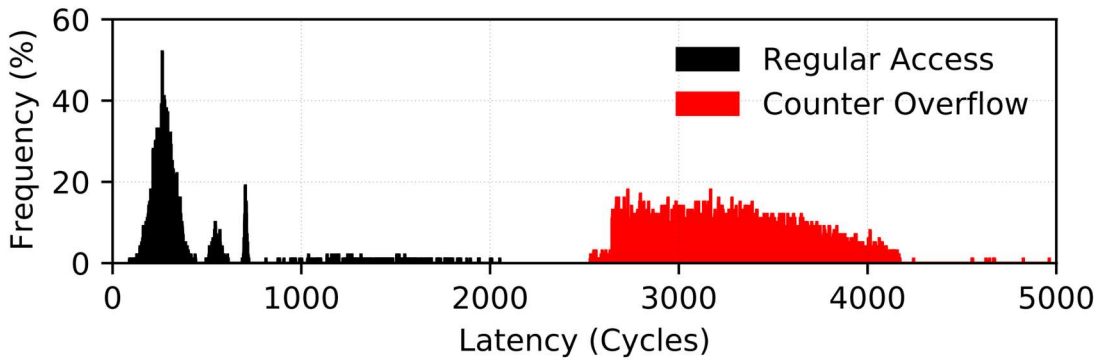


Memory access latency impacted by counter tree overflow (Simulation)

UCF

Timing Characterizations: Writes

- Processor data writes can exhibit the same set of variations due to caching status of security metadata.
- Different from reads, write operations increment encryption counter (+ Counters in integrity tree).



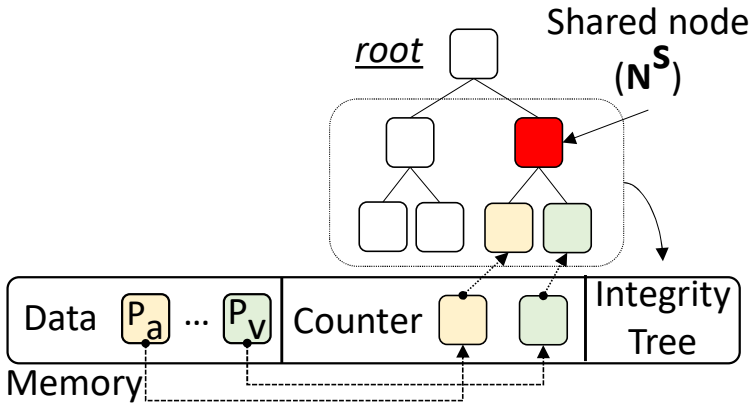
Memory access latency impacted by counter tree overflow (Simulation)

Counter overflow leads to **highly distinguishable slow/fast path**.

UCF

MetaLeak-T: Side Channel Exploiting Tree Sharing

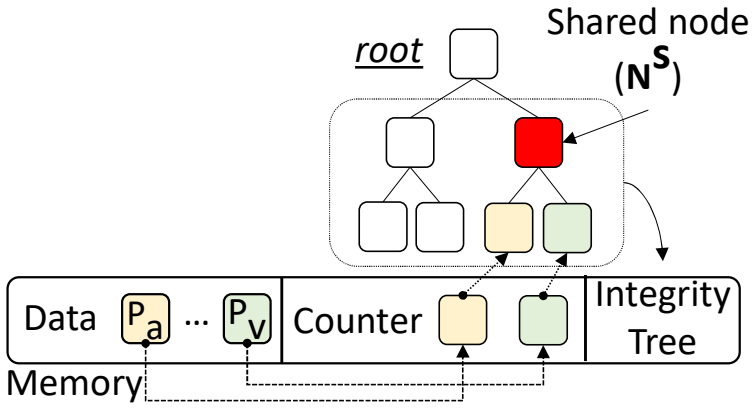
- **Implicit memory sharing:**
  - Integrity tree creates ***shared tree blocks*** *among pages* (across processes).
  - Creates practical **shared-memory** side channel **without explicit data sharing**.



UCF

MetaLeak-T: Side Channel Exploiting Tree Sharing

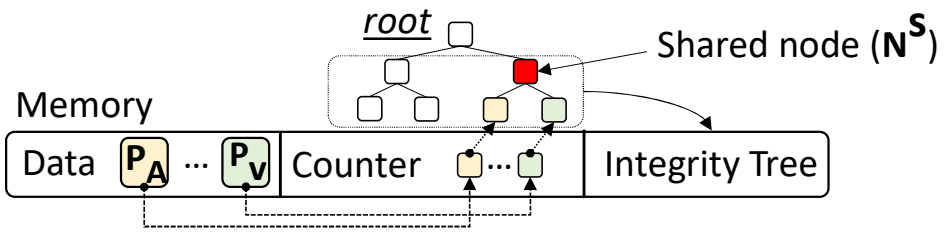
- **Implicit memory sharing:**
  - Integrity tree creates ***shared tree blocks*** *among pages* (across processes).
  - Creates practical **shared-memory** side channel **without explicit data sharing**.



Vulnerability stemming from **a new source of sharing - Metadata**

UCF

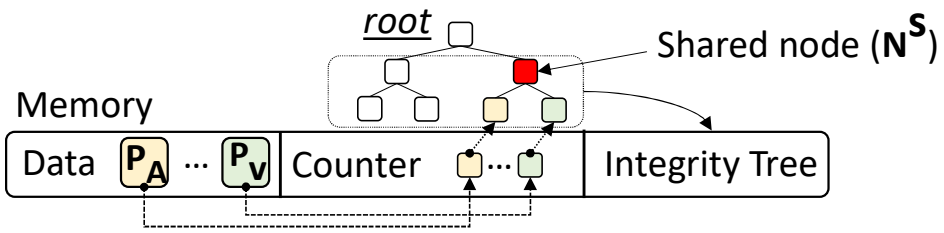
MetaLeak-T: Side Channel Exploiting Tree Sharing



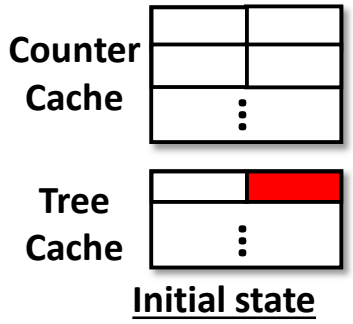
Implicit sharing of integrity tree metadata

UCF

MetaLeak-T: Side Channel Exploiting Tree Sharing



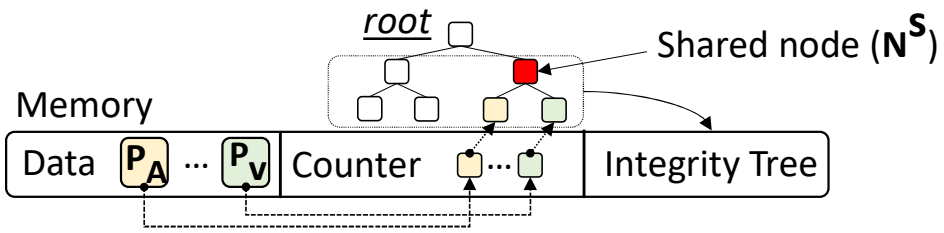
Implicit sharing of integrity tree metadata



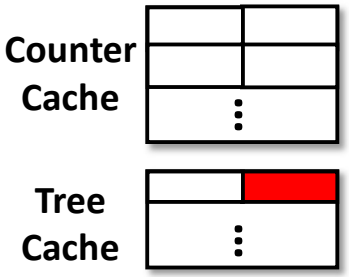
Initial state

UCF

MetaLeak-T: Side Channel Exploiting Tree Sharing

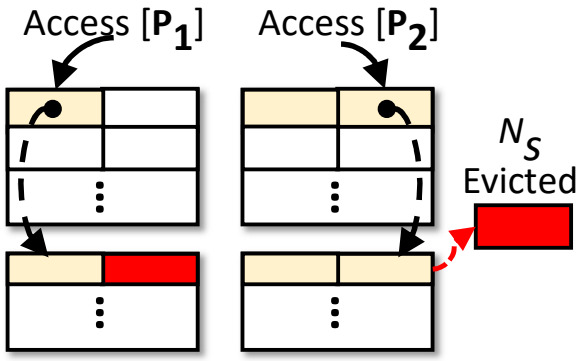


Implicit sharing of integrity tree metadata



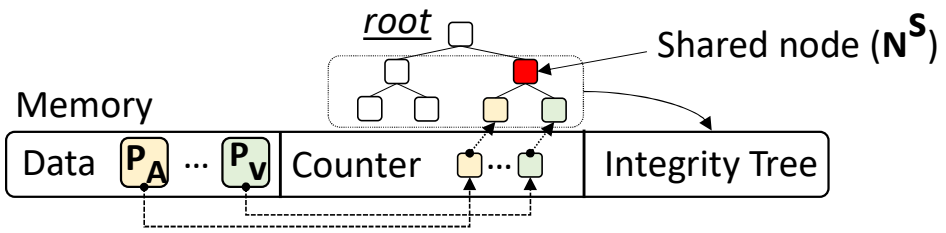
Initial state

Step 1: Evict shared tree blocks

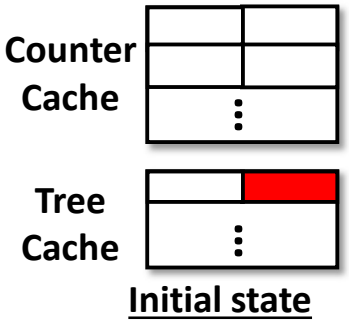


UCF

MetaLeak-T: Side Channel Exploiting Tree Sharing

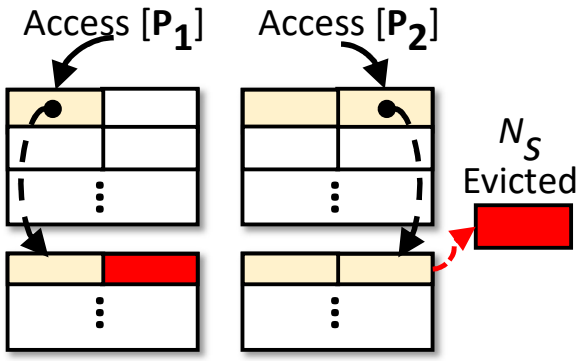


Implicit sharing of integrity tree metadata

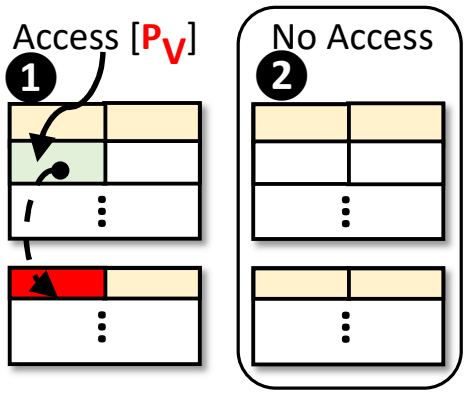


Initial state

Step 1: Evict shared tree blocks

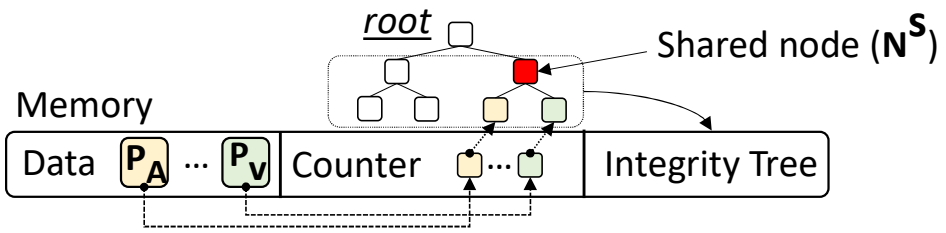


Step 2: Victim execution

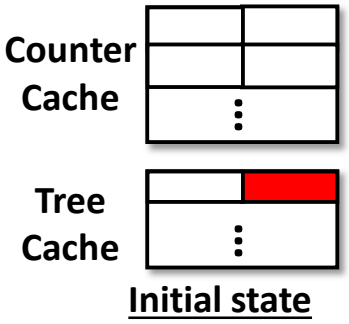


UCF

MetaLeak-T: Side Channel Exploiting Tree Sharing

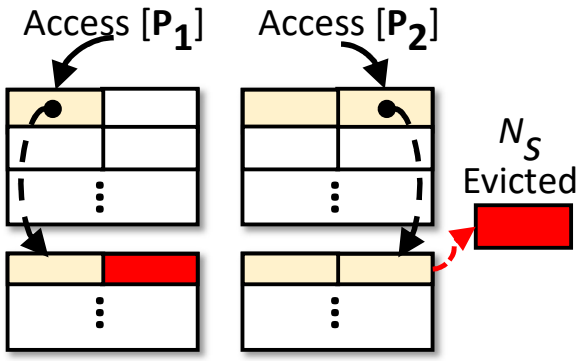


Implicit sharing of integrity tree metadata

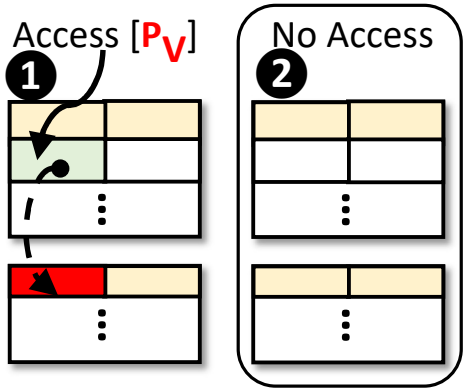


Initial state

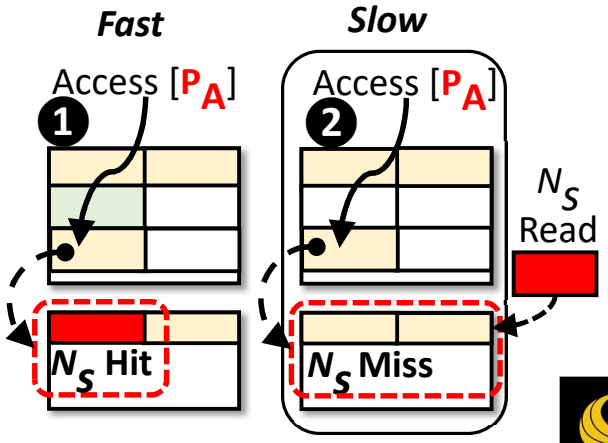
Step 1: Evict shared tree blocks



Step 2: Victim execution



Step 3: Timed reload shared tree blocks



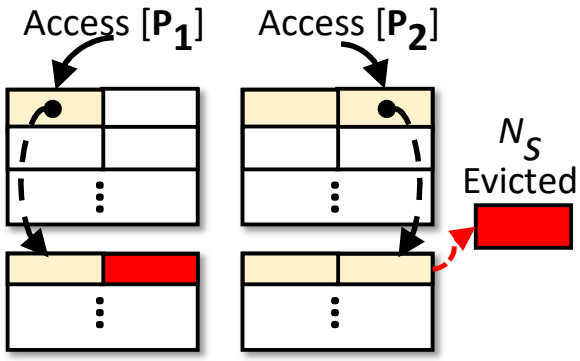
UCF

MetaLeak-T: Side Channel Exploiting Tree Sharing

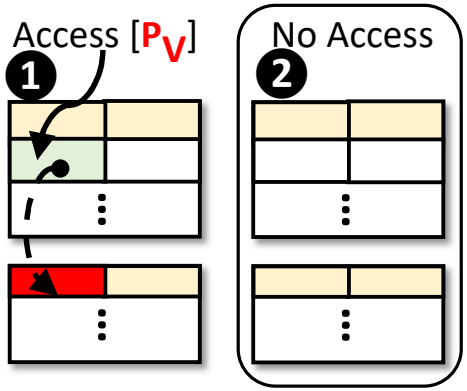
❶ **Fast integrity verification: Victim accessed shared tree block**

❷ **Slow integrity verification: Victim did not access shared tree block**

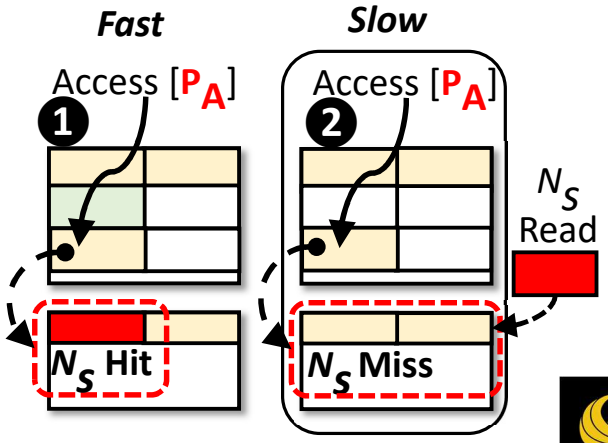
Step 1: Evict shared tree blocks



Step 2: Victim execution



Step 3: Timed reload shared tree blocks



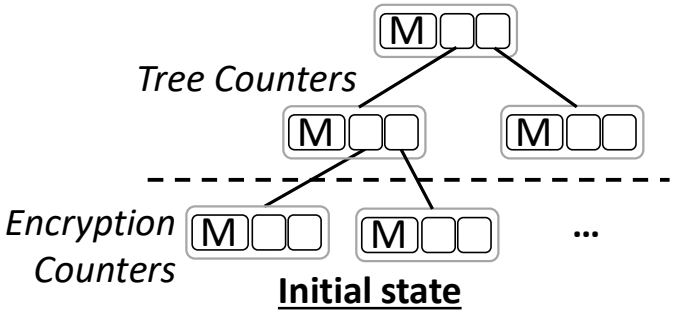
UCF

MetaLeak-C: Exploiting Counters for Write Monitoring

- **Recall:** Counter overflow leads to high latency variations.
- Encryption counters are typically per-page → not shared across processes.
- Counters in *Counter-based integrity tree* are shared across processes.

MetaLeak-C: Exploiting Counters for Write Monitoring

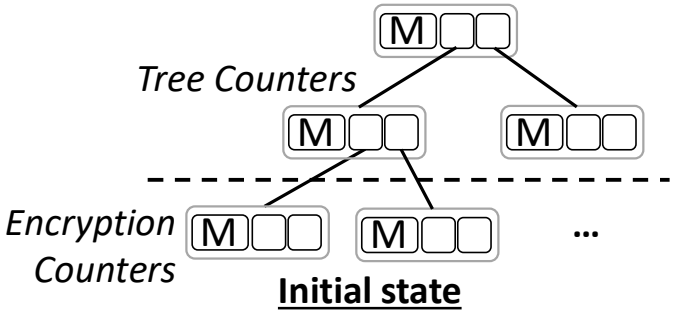
- **Recall:** Counter overflow leads to high latency variations.
- Encryption counters are typically per-page → not shared across processes.
- Counters in *Counter-based integrity tree* are shared across processes.



UCF

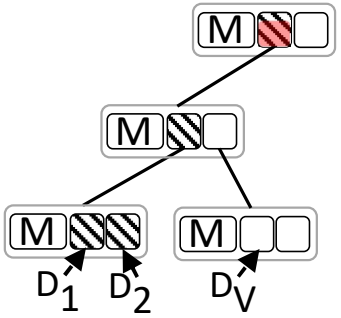
MetaLeak-C: Exploiting Counters for Write Monitoring

- **Recall:** Counter overflow leads to high latency variations.
- Encryption counters are typically per-page → not shared across processes.
- Counters in *Counter-based integrity tree* are shared across processes.



Step 1: Preset shared counter

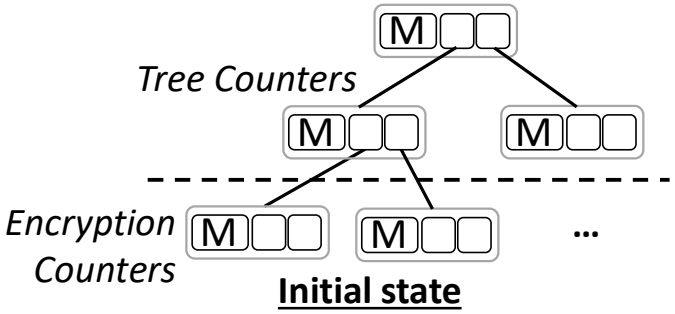
Write [P_1, P_2]

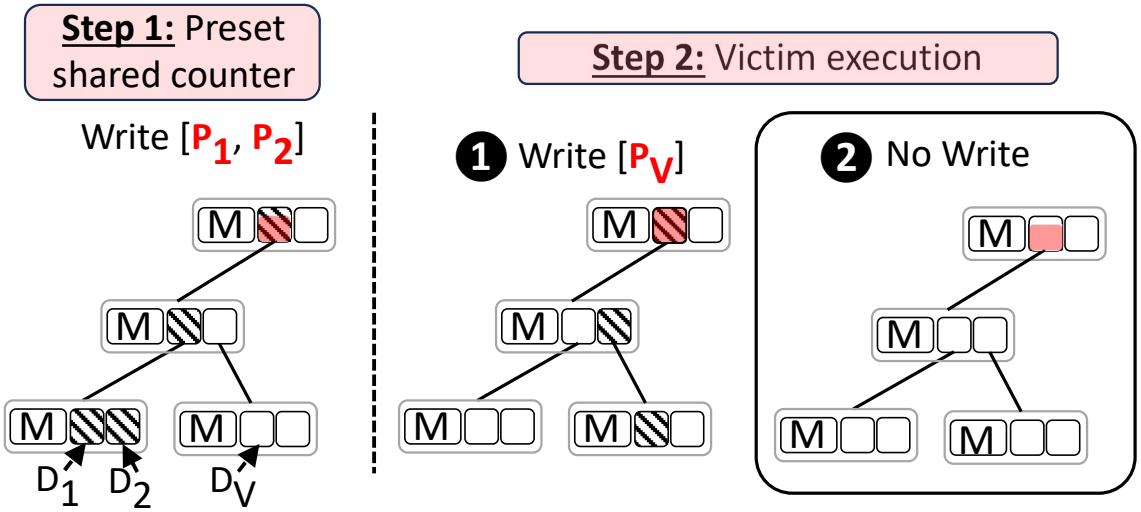


UCF

MetaLeak-C: Exploiting Counters for Write Monitoring

- **Recall:** Counter overflow leads to high latency variations.
- Encryption counters are typically per-page → not shared across processes.
- Counters in *Counter-based integrity tree* are shared across processes.

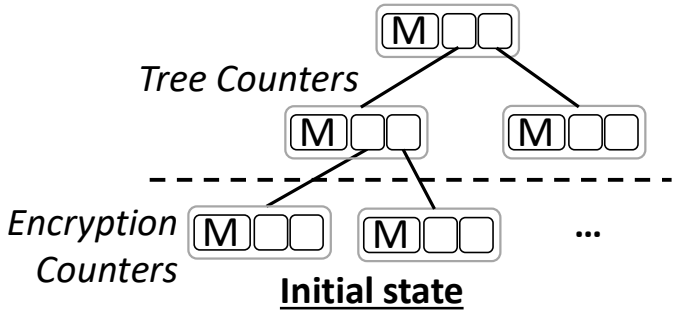




UCF

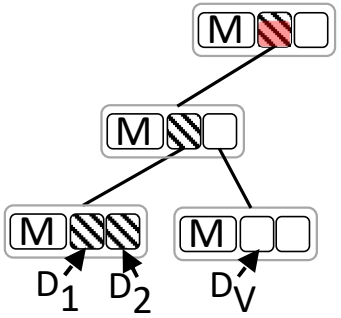
MetaLeak-C: Exploiting Counters for Write Monitoring

- **Recall:** Counter overflow leads to high latency variations.
- Encryption counters are typically per-page → not shared across processes.
- Counters in *Counter-based integrity tree* are shared across processes.



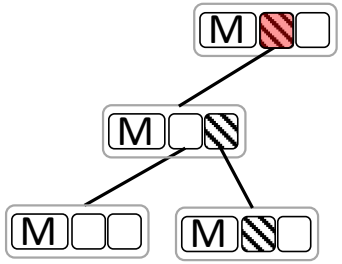
Step 1: Preset shared counter

Write [P_1, P_2]

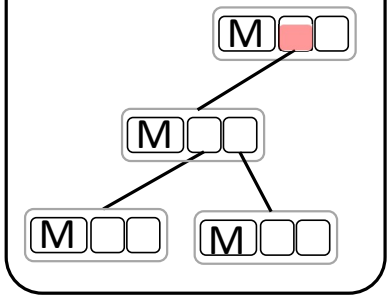


Step 2: Victim execution

1 Write [P_V]

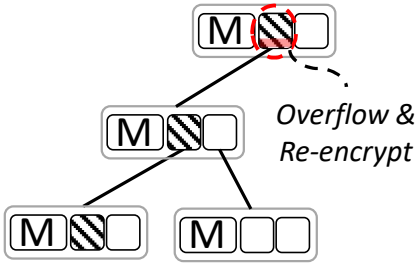


2 No Write



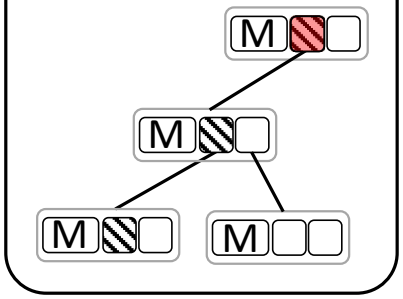
Step 3: Infer counter overflow

1 Write [P_1] → ***Read*** [P_A]



Slow

2 Write [P_1] → ***Read*** [P_A]



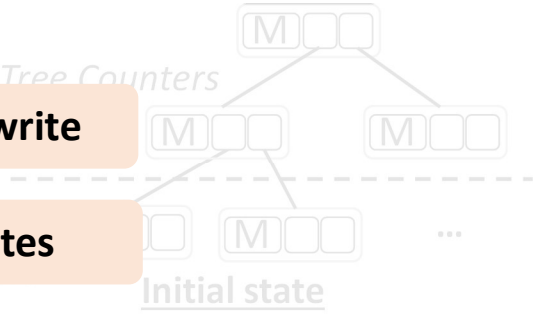
Fast

UCF

MetaLeak-C: Exploiting Counters for Write Monitoring

Recall: Counter overflow leads to high latency variations.

- Encryption cou
- Counters in Co



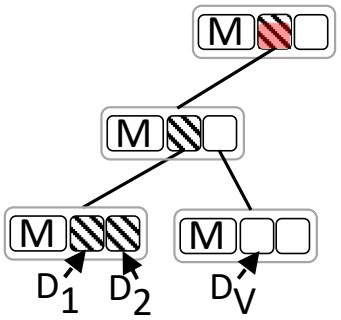
❶ **Fast memory access latency: Victim did not perform write**

❷ **Slow memory access latency: Victim performed writes**

mPreset + mProbe

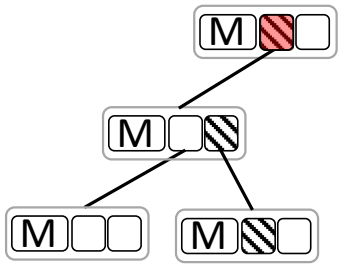
Step 1: Preset shared counter

Write [P_1, P_2]

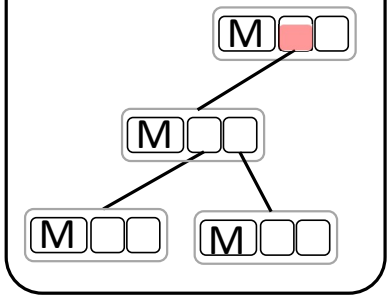


Step 2: Victim execution

❶ Write [P_V]

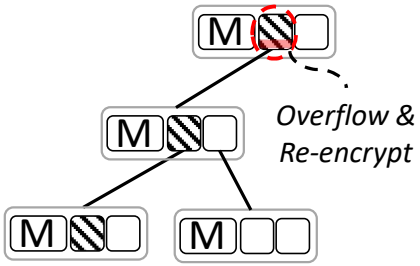


❷ No Write



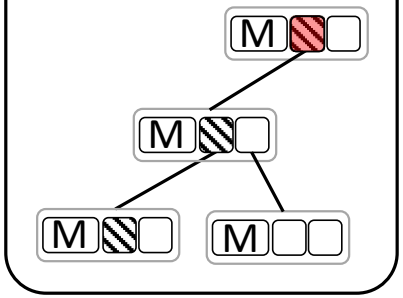
Step 3: Infer counter overflow

❶ Write [P_1] → ***Read*** [P_A]



Slow

❷ Write [P_1] → ***Read*** [P_A]



Fast

UCF

Experimental Setup

- We investigate both *state-of-the-art* **academic designs** and **commercial implementations**.
- Due to lack of commercial prototype, we extensively model academic design in **Gem5**.

	Simulation	**Intel SGX**
Architecture Configuration	**Gem5** (Quad-core, OoO, Full system)	**Intel Core i7-9700K**
Encryption	Counter-mode (Split-counter)	Counter-mode (Monolithic counter)
Integrity tree	1. Hash-based tree (Bonsai merkle tree) 2. Counter-based tree (Split-counter tree)	Counter-based tree (Monolithic counter)
Demonstrated attacks	1. `libjpeg`: Image processing library 2. `libgcrypt, mbedTLS`: Crypto library	1. `libgcrypt, mbedTLS`: Crypto library

Case-study in Simulation: MetaLeak-T in `libjpeg`

Target application: `libjpeg` (Open-source image processing library).

Exploited gadget:

```
1   encode_one_block (...) {
2    ...
3    /* Encode the coefficients */
4    for (k = 1; k < DCTSIZE2; k++) {
5     if (block[jpeg_natural_order[k]] == 0) {
6      r++;
7     } else {
8     ...
9      /* Check for out-of-range coefficient */
10    if (nbits > MAX_COEF_BITS) { ... }
11   }
12  }
```

Gadget description:

- JPEG compression algorithm using Huffman coding.
- **`block[][]:`** Per-block entropy (i.e., *changes in image patterns across blocks*).
- These are used to derive compressed coefficient, used to generate JPEG.
- Observing accesses in lines 6 & 10 can leak sensitive information:
  - i.e., *sharp color gradient changes* across blocks.

Attack setup:

Case-study in Simulation: MetaLeak-T in `libjpeg`

Target application: `libjpeg` (Open-source image processing library).

Exploited gadget:

```
1   encode_one_block (...) {
2    ...
3    /* Encode the coefficients */
4    for (k = 1; k < DCTSIZE2; k++) {
5     if (block[jpeg_natural_order[k]] == 0) {
6      r++;
7     } else {
8     ...
9      /* Check for out-of-range coefficient */
10    if (nbits > MAX_COEF_BITS) { ... }
11   }
12  }
```

Gadget description:

- JPEG compression algorithm using Huffman coding.
- **`block[][]:`** Per-block entropy (i.e., *changes in image patterns across blocks*).
- These are used to derive compressed coefficient, used to generate JPEG.
- Observing accesses in lines 6 & 10 can leak sensitive information:
  - i.e., *sharp color gradient changes* across blocks.

Attack setup:

Case-study in Simulation: MetaLeak-T in `libjpeg`

Target application: `libjpeg` (Open-source image processing library).

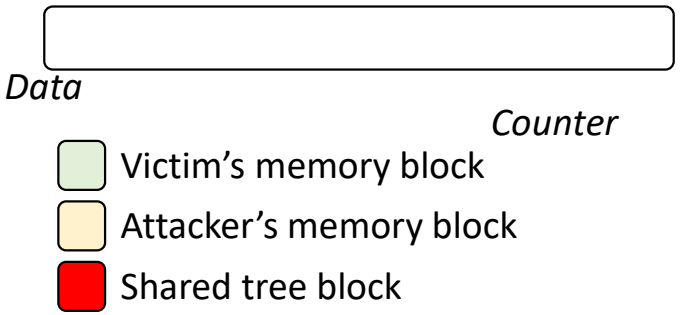
Exploited gadget:

```
1   encode_one_block (...) {
2    ...
3    /* Encode the coefficients */
4    for (k = 1; k < DCTSIZE2; k++) {
5     if (block[jpeg_natural_order[k]] == 0) {
6      r++;
7     } else {
8     ...
9      /* Check for out-of-range coefficient */
10    if (nbits > MAX_COEF_BITS) { ... }
11   }
12  }
```

Gadget description:

- JPEG compression algorithm using Huffman coding.
- **`block[][]:`** Per-block entropy (i.e., *changes in image patterns across blocks*).
- These are used to derive compressed coefficient, used to generate JPEG.
- Observing accesses in lines 6 & 10 can leak sensitive information:
  - i.e., *sharp color gradient changes* across blocks.

Attack setup:



UCF

Case-study in Simulation: MetaLeak-T in `libjpeg`

Target application: `libjpeg` (Open-source image processing library).

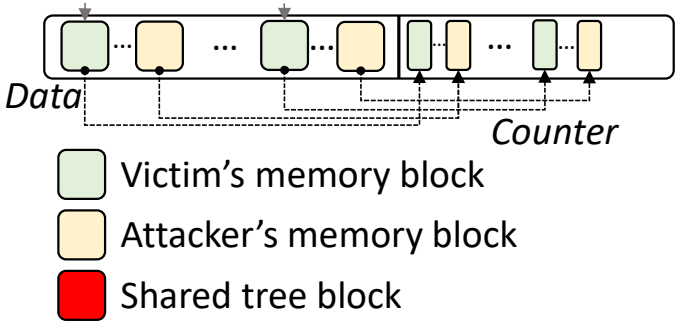
Exploited gadget:

```
1   encode_one_block (...) {
2    ...
3    /* Encode the coefficients */
4    for (k = 1; k < DCTSIZE2; k++) {
5     if (block[jpeg_natural_order[k]] == 0) {
6      r++;
7     } else {
8     ...
9      /* Check for out-of-range coefficient */
10    if (nbits > MAX_COEF_BITS) { ... }
11   }
12  }
```

Gadget description:

- JPEG compression algorithm using Huffman coding.
- **`block[][]:`** Per-block entropy (i.e., *changes in image patterns across blocks*).
- These are used to derive compressed coefficient, used to generate JPEG.
- Observing accesses in lines 6 & 10 can leak sensitive information:
  - i.e., *sharp color gradient changes* across blocks.

Attack setup:



Victim's memory block

Attacker's memory block

Shared tree block

UCF

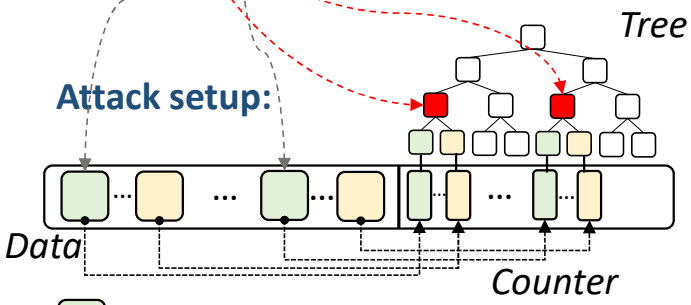
Case-study in Simulation: MetaLeak-T in `libjpeg`

Target application: `libjpeg` (Open-source image processing library).

Exploited gadget:

```
1   encode_one_block (...) {
2    ...
3    /* Encode the coefficients */
4    for (k = 1; k < DCTSIZE2; k++) {
5     if (block[jpeg_natural_order[k]] == 0) {
6      r++;
7     } else {
8     ...
9      /* Check for out-of-range coefficient */
10    if (nbits > MAX_COEF_BITS) { ... }
11   }
12  }
```

Attack setup:



Victim's memory block

Attacker's memory block

Shared tree block

Gadget description:

- JPEG compression algorithm using Huffman coding.
- **`block[][]:`** Per-block entropy (i.e., *changes in image patterns across blocks*).
- These are used to derive compressed coefficient, used to generate JPEG.
- Observing accesses in lines 6 & 10 can leak sensitive information:
  - i.e., *sharp color gradient changes* across blocks.

UCF

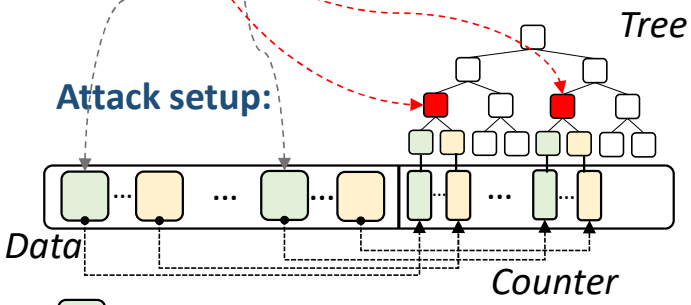
Case-study in Simulation: MetaLeak-T in `libjpeg`

Target application: `libjpeg` (Open-source image processing library).

Exploited gadget:

```
1  encode_one_block (...) {
2    ...
3    /* Encode the coefficients */
4    for (k = 1; k < DCTSIZE2; k++) {
5      if (block[jpeg_natural_order[k]] == 0) {
6        r++;
7      } else {
8        ...
9        /* Check for out-of-range coefficient */
10       if (nbits > MAX_COEF_BITS) { ... }
11     }
12   }
```

Attack setup:



Victim's memory block

Attacker's memory block

Shared tree block

Gadget description:

- JPEG compression algorithm using Huffman coding.
- **`block[][]:`** Per-block entropy (i.e., *changes in image patterns across blocks*).
- These are used to derive compressed coefficient, used to generate JPEG.
- Observing accesses in lines 6 & 10 can leak sensitive information:
  - i.e., *sharp color gradient changes* across blocks.

Results:

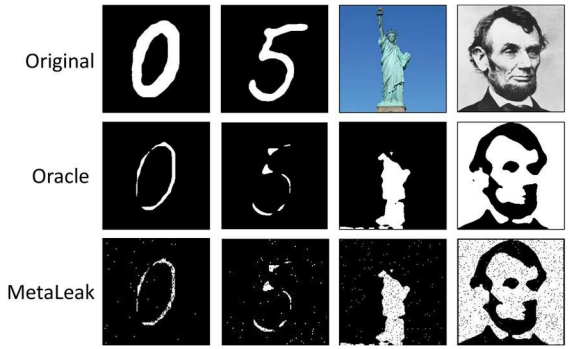


Image reconstruction using MetaLeak-C

UCF

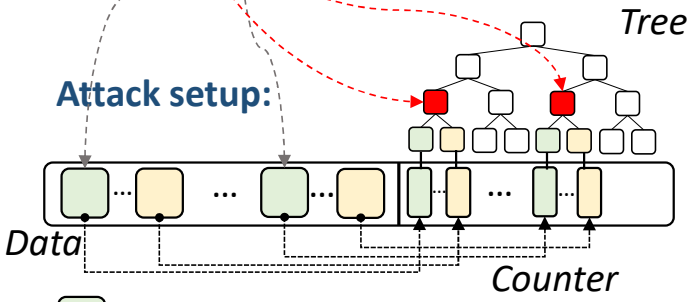
Case-study in Simulation: MetaLeak-T in `libjpeg`

Target application: `libjpeg` (Open-source image processing library).

Exploited gadget:

```
1   encode_one_block (...) {
2    ...
3    /* Encode the coefficients */
4    for (k = 1; k < DCTSIZE2; k++) {
5     if (block[jpeg_natural_order[k]] == 0) {
6      r++;
7     } else {
8     ...
9      /* Check for out-of-range coefficient */
10    if (nbits > MAX_COEF_BITS) { ... }
11   }
12  }
```

Attack setup:



Victim's memory block

Attacker's memory block

Shared tree block

Gadget description:

- JPEG compression algorithm using Huffman coding.
- **`block[][]`:** Per-block entropy (i.e., *changes in image patterns across blocks*).
- These are used to derive compressed coefficient, used to generate JPEG.
- Observing accesses in lines 6 & 10 can leak sensitive information:
  - i.e., *sharp color gradient changes* across blocks.

Results:

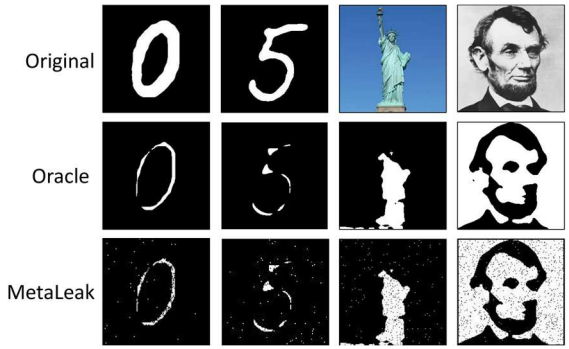


Image reconstruction using MetaLeak-C

Victim access detection accuracy: **94.3%**

UCF

Case-study in SGX: MetaLeak-T in **`libgcrypt`**

Case-study in SGX: MetaLeak-T in `libgcrypt`

Target application: `libgcrypt` (Open-source cryptographic library).

Exploited gadget:

```
1   gcry_mpi_powm (...) {
2    ...
3    /* Main loop */
4    for (;;) {
5     /* Square operation */
6     _gcry_mpih_sqr_n_basecase(...);
7     /* Check if exponent bit is 1 */
8     if ( (mpi_limb_signed_t)e < 0 ) {
9      /* Multiplication operation */
10     _gcry_mpih_mul_karatsuba_case(...);
11    }
12   }
```

Gadget description:

- RSA implementation using square-and-multiply arithmetic.
- Square and multiply (line 6 and 10) are executed *if secret exponent is 1*.
- Only square (line 6) is executed *if secret exponent is 0*.
- Secret exponent bits can be leaked by observing accesses to lines 6 & 10.

Case-study in SGX: MetaLeak-T in `libgcrypt`

Target application: `libgcrypt` (Open-source cryptographic library).

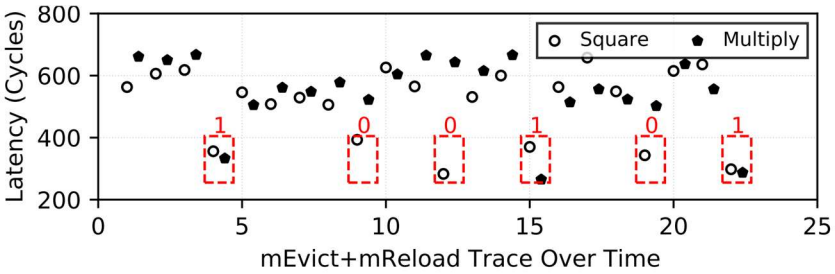
Exploited gadget:

```
1  gcry_mpi_powm (...) {
2   ...
3   /* Main loop */
4   for (;;) {
5    /* Square operation */
6    _gcry_mpih_sqr_n_basecase(...);
7    /* Check if exponent bit is 1 */
8    if ( (mpi_limb_signed_t)e < 0 ) {
9     /* Multiplication operation */
10    _gcry_mpih_mul_karatsuba_case(...);
11   }
12  }
```

Gadget description:

- RSA implementation using square-and-multiply arithmetic.
- Square and multiply (line 6 and 10) are executed *if secret exponent is 1*.
- Only square (line 6) is executed *if secret exponent is 0*.
- Secret exponent bits can be leaked by observing accesses to lines 6 & 10.

Results:



Accuracy of exponent bit stealing:

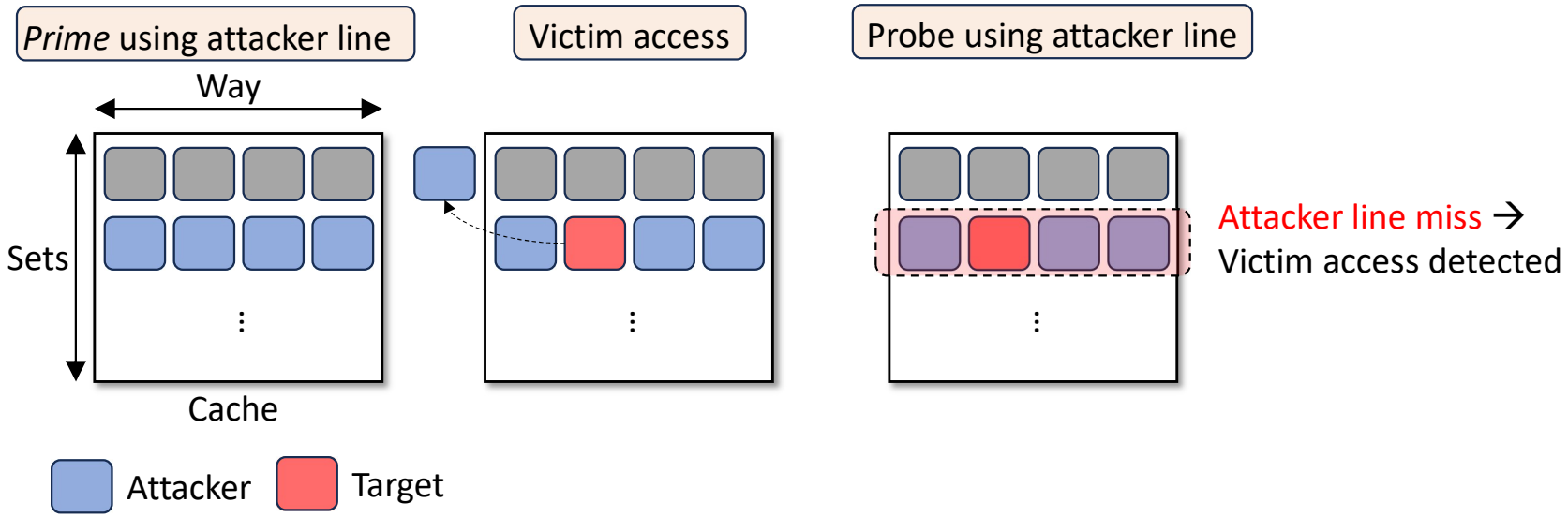
- **SGX:** 91.2%
- **Simulation:** 95.1%

mEvict+mReload latency traces for secret exponent bit '100101'

UCF

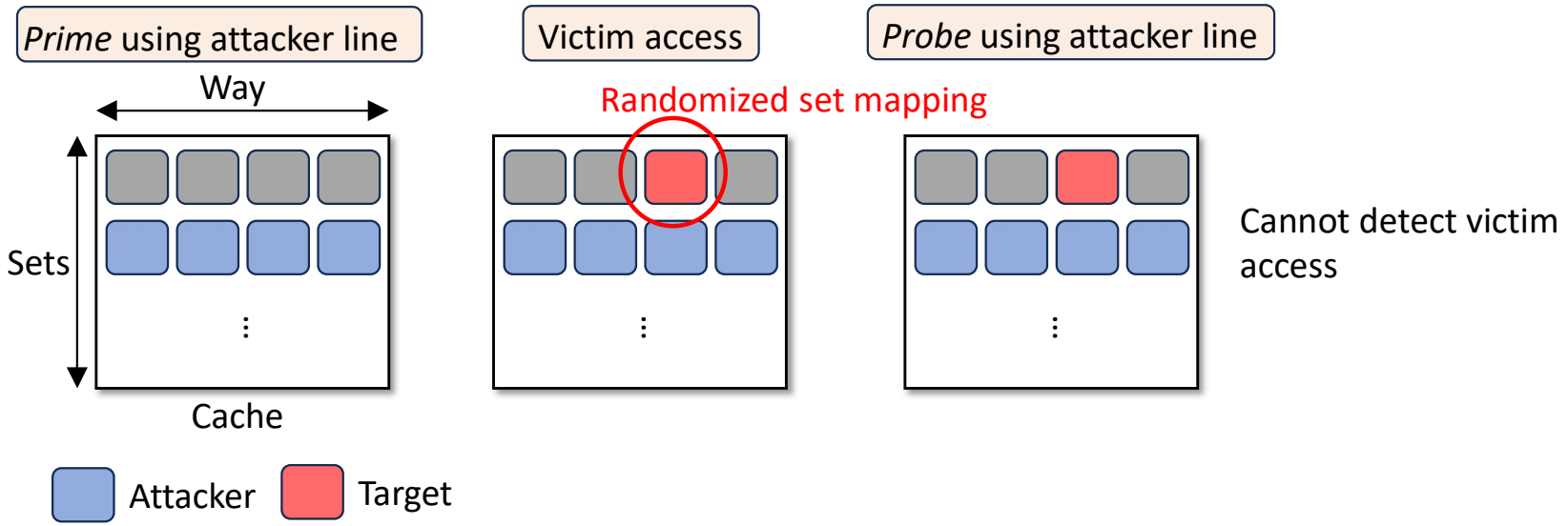
Uniqueness of MetaLeak Attack

- MetaLeak introduces *new source of leakage.*
  - Prior TEE works utilize *known microarchitectural vulnerabilities*.
  - MetaLeak exposes a **new vulnerability** *uniquely tied to the design of secure processors*.
- MetaLeak is resilient against prior defenses (i.e., partitioning and randomization/obfuscation).
  - Mainstream protection schemes assume *there is no sharing of data*.
  - Secure processor metadata is *shared and updated during runtime*.



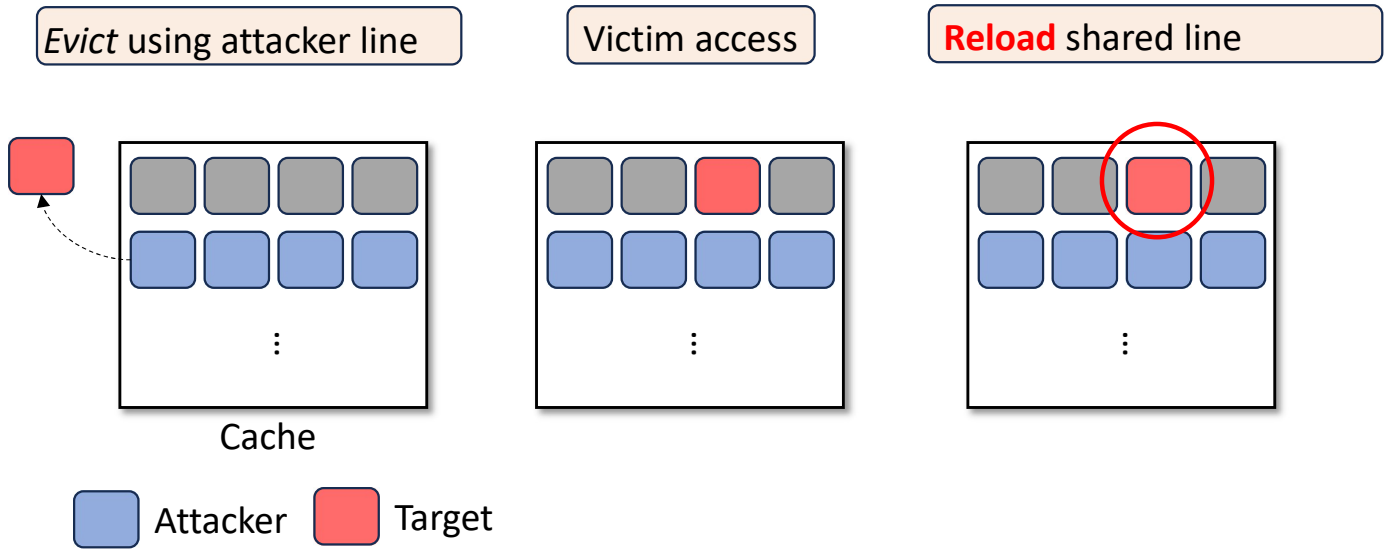
Uniqueness of MetaLeak Attack

- MetaLeak introduces *new source of leakage.*
  - Prior TEE works utilize *known microarchitectural vulnerabilities*.
  - MetaLeak exposes a **new vulnerability** *uniquely tied to the design of secure processors*.
- MetaLeak is resilient against prior defenses (i.e., partitioning and randomization/obfuscation).
  - Mainstream protection schemes assume *there is no sharing of data*.
  - Secure processor metadata is *shared and updated during runtime*.



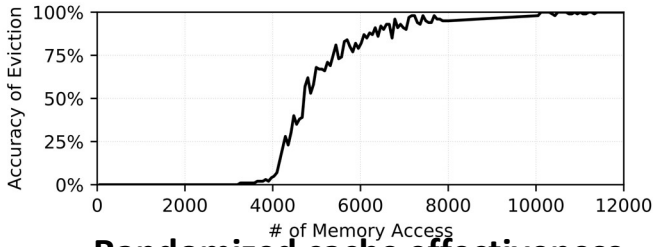
Uniqueness of MetaLeak Attack

- MetaLeak introduces *new source of leakage.*
  - Prior TEE works utilize *known microarchitectural vulnerabilities*.
  - MetaLeak exposes a **new vulnerability** *uniquely tied to the design of secure processors*.
- MetaLeak is resilient against prior defenses (i.e., partitioning and randomization/obfuscation).
  - Mainstream protection schemes assume *there is no sharing of data*.
  - Secure processor metadata is *shared and updated during runtime*.

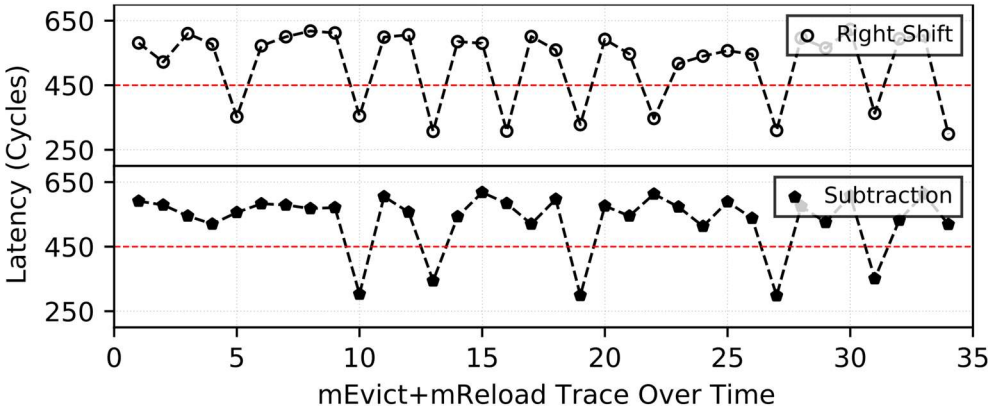


More On Paper…

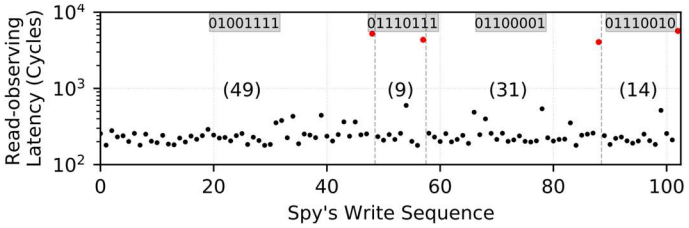
- Covert channel using MetaLeak-T and MetaLeak-C.
- SGX case-study on mbedTLS cryptographic library.
- Impact of cache protection scheme.
- Discussion about scope and effectiveness of MetaLeak.
- Pitfalls and potential direction of MetaLeak mitigation.



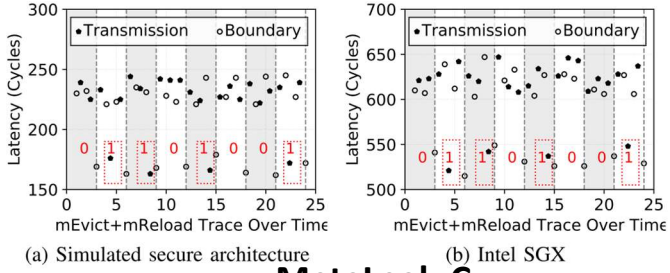
Randomized cache effectiveness



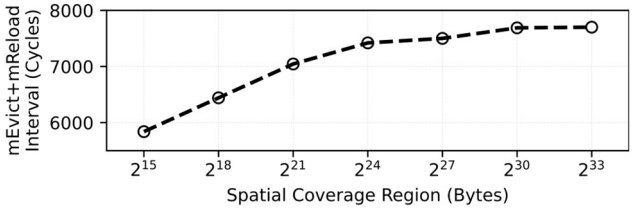
Attack on mBedTLS



Covert channel: MetaLeak-T



MetaLeak-C



Spatial/temporal resolution

UCF

Conclusion

- We investigate uArch security in secure processor architecture.
- Our work identifies unique attack vectors in secure processor design.
- We present MetaLeak, side channel framework for security metadata.
- We demonstrate extensive case-study on both academic designs and commercial secure processor implementation (SGX).

Thanks! Questions?

Md Hafizul Islam Chowdhuryy
CASR Lab (https://casr.ece.ucf.edu)
Email: hafizul.islam@ucf.edu