

System Security Overview with an Emphasis on Security Issues for Storage and Emerging NVM (Part 2)

Byoungyoung Lee (이병영)
byoungyoung@snu.ac.kr
Seoul National University

Outline

Part1. Bugs in File Systems

Semantic inconsistency inference

Fuzzing

Part2. Attacks and Defenses

Ransomware

Cold boot attacks

Side-channels

The CIA Principle in Security

- Confidentiality
 - Ability to hide information from unauthorized access
- Integrity
 - Maintaining consistency, accuracy, and trustworthiness of data
- Availability
 - Information requested is readily available to authorized entity

Defenses

- Many defense schemes
 - Access control
 - Encryption
 - Authentication
 - Authorization
 - Firewall
 - Intrusion detection system
 - etc.
- Which defense schemes should you need?
 - It depends on an attack model.

Attack Vectors

- Privilege escalation attacks
 - Exploiting software bugs
 - Exploiting hardware bugs
- Ransomware
- Cold boot attacks
- Side-channels

Outline

Part1. Bugs in File Systems

Semantic inconsistency inference

Fuzzing

Part2. Attacks and Defenses

Ransomware

Cold boot attacks

Side-channels

Ransomware



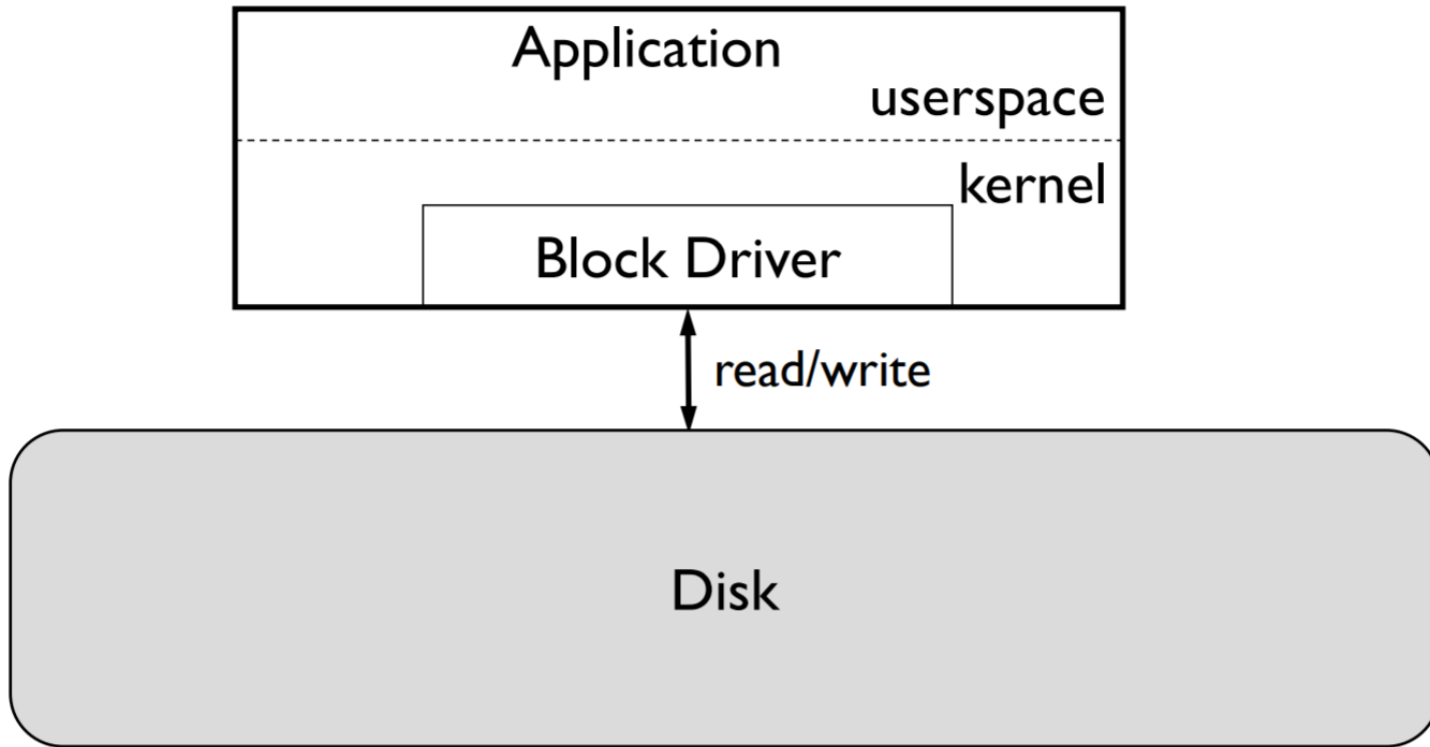
A random notification:
users files have been
encrypted

Pay ransom to recover
user files

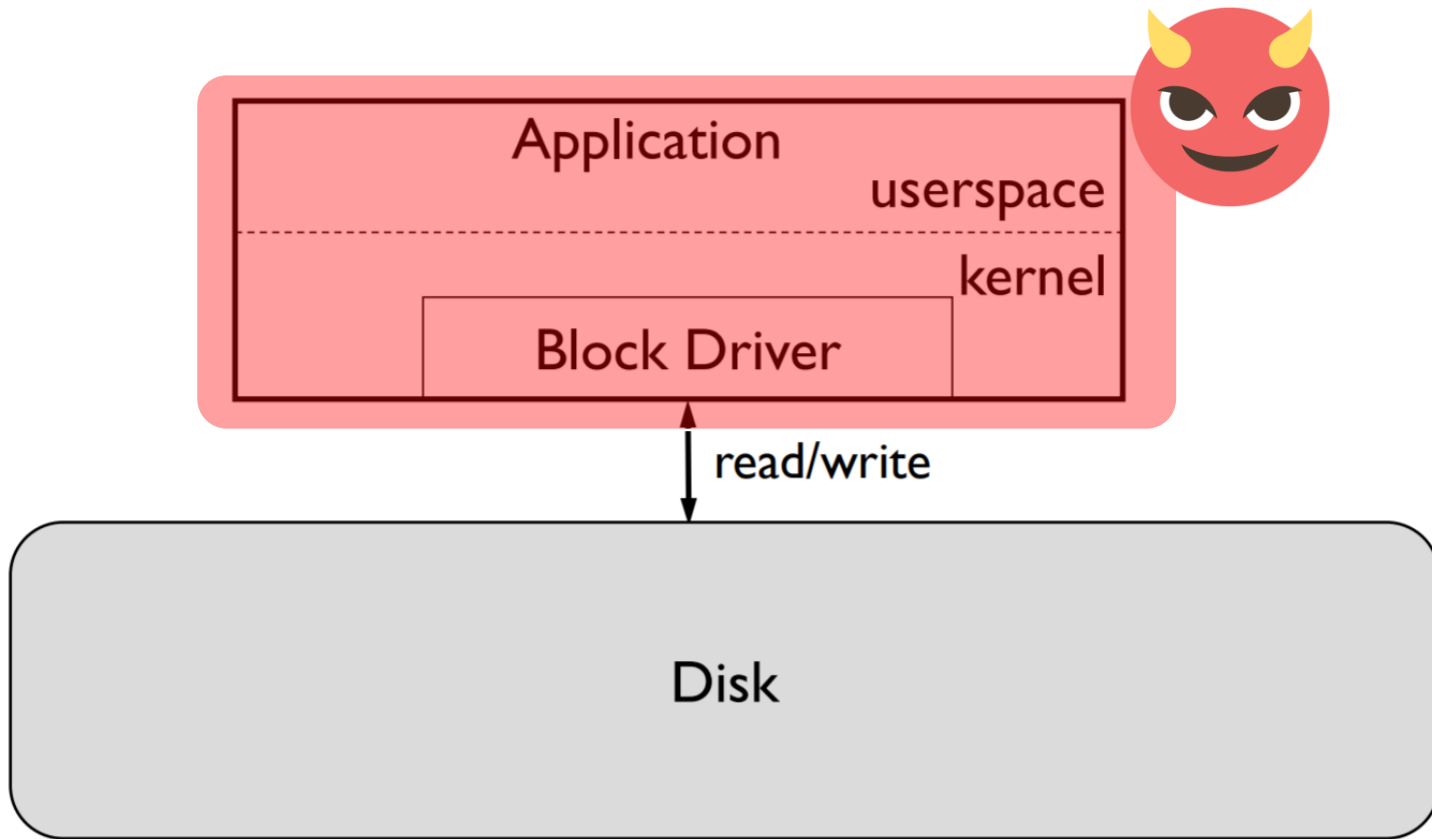
Candidate Defenses against Ransomware

- Malware detection
 - Damage has already happened when ransomware is detected
- Journaling & log-structured filesystem
 - Ransomware with kernel privilege can destroy data backups
- Networked & cloud storage
 - Increased storage cost
 - Can be stopped by ransomware

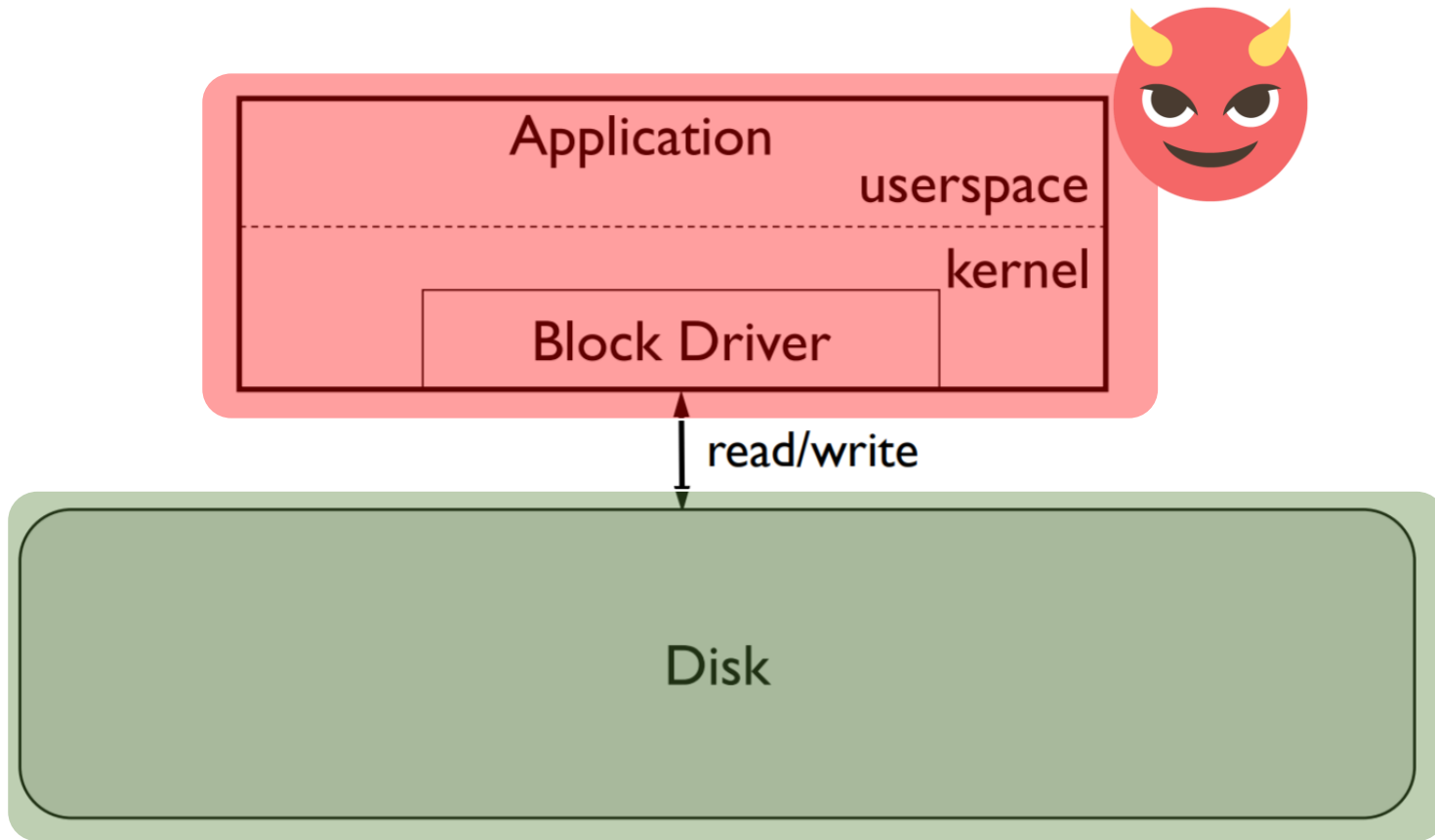
Threat Model of Ransomware



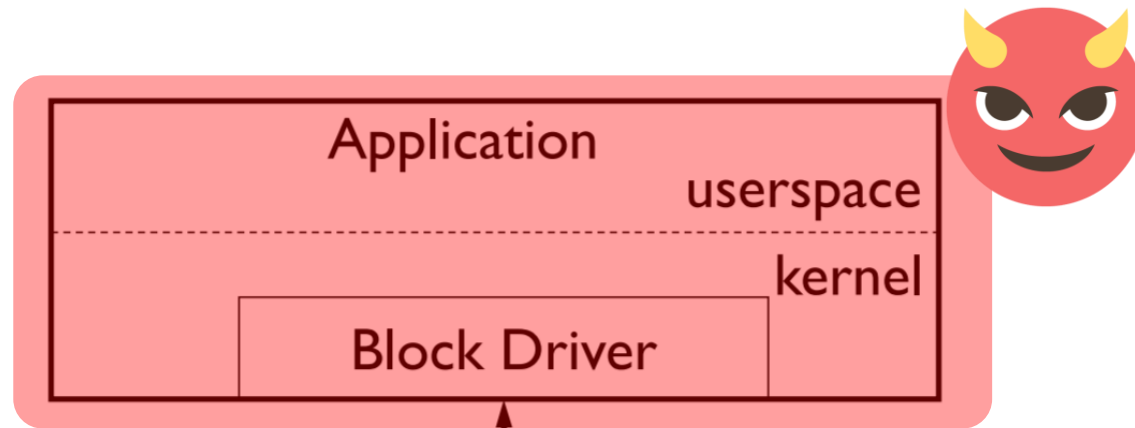
Threat Model of Ransomware



Threat Model of Ransomware

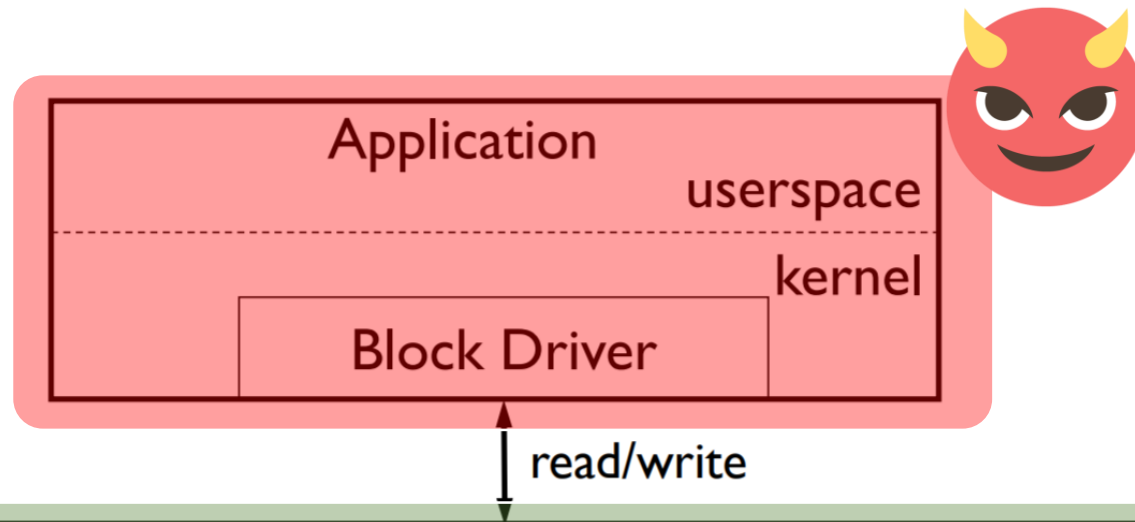


Threat Model of Ransomware



**Traditional security solutions
does not work**

Threat Model of Ransomware



**Traditional security solutions
does not work**

**Need low-level solutions
at the disk layer**

Defense Solutions in SSD

- Defenses implemented in flash-based SSD
 - Without relying on software-based solutions
 - **FlashGuard: Data recovery [CCS 17]**
 - Leveraging existing features in SSD: out-of-place update and garbage collection
 - Retain pages caused by ransomware encryptions
 - **SSD-Insider: Attack detection [ICDCS 18]**
 - Detection based on behavior characteristics
 - Recovery of infected files using intrinsic delayed deletion features of NAND flash

Outline

Part1. Bugs in File Systems

Semantic inconsistency inference

Fuzzing

Part2. Attacks and Defenses

Ransomware

Cold boot attacks

Side-channels

Disk Encryption (Encrypted File Systems)



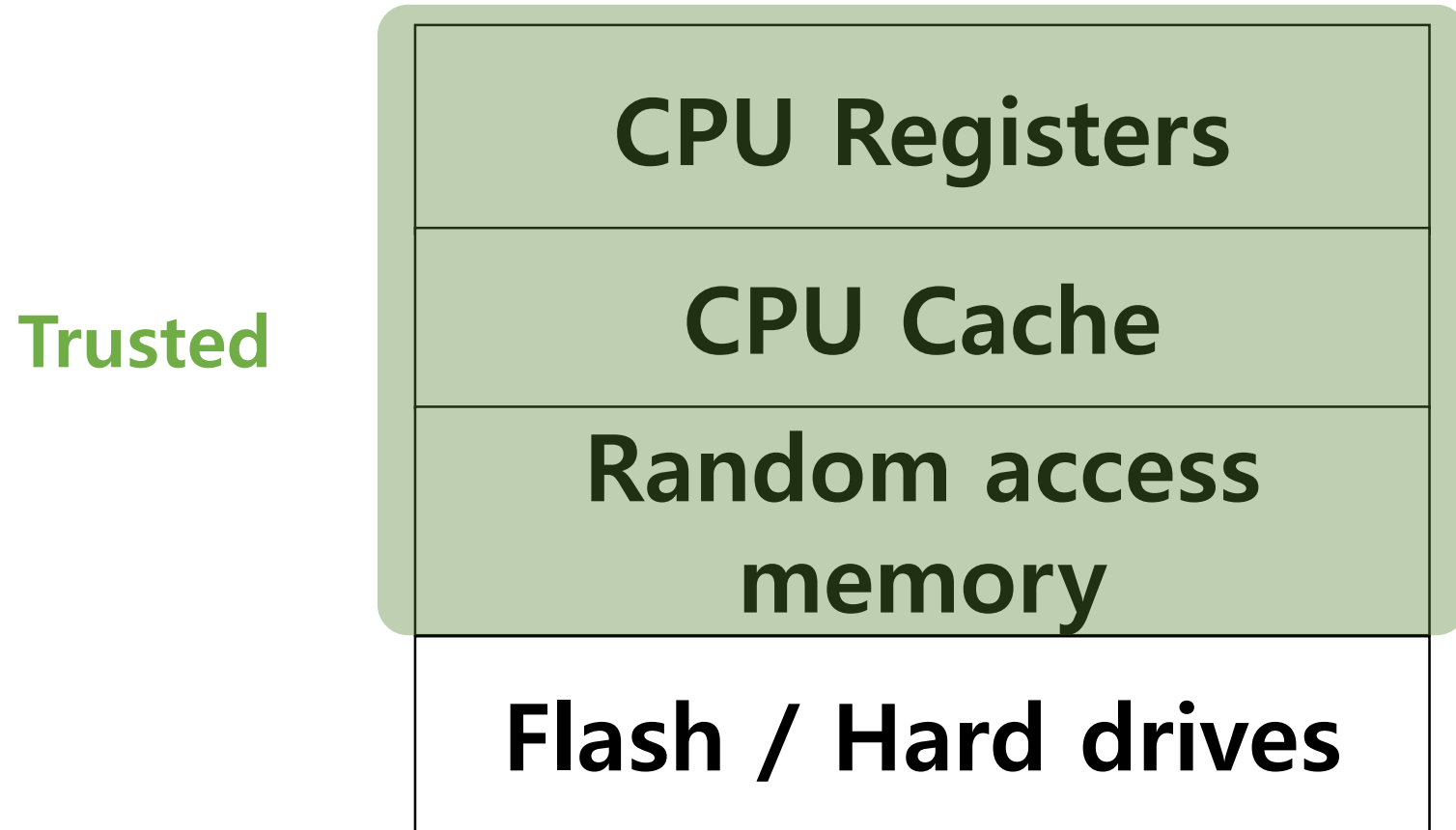
Trust Models in Encrypted FS

- Memory hierarchy with trust models (when OS is trusted)

CPU Registers
CPU Cache
Random access memory
Flash / Hard drives

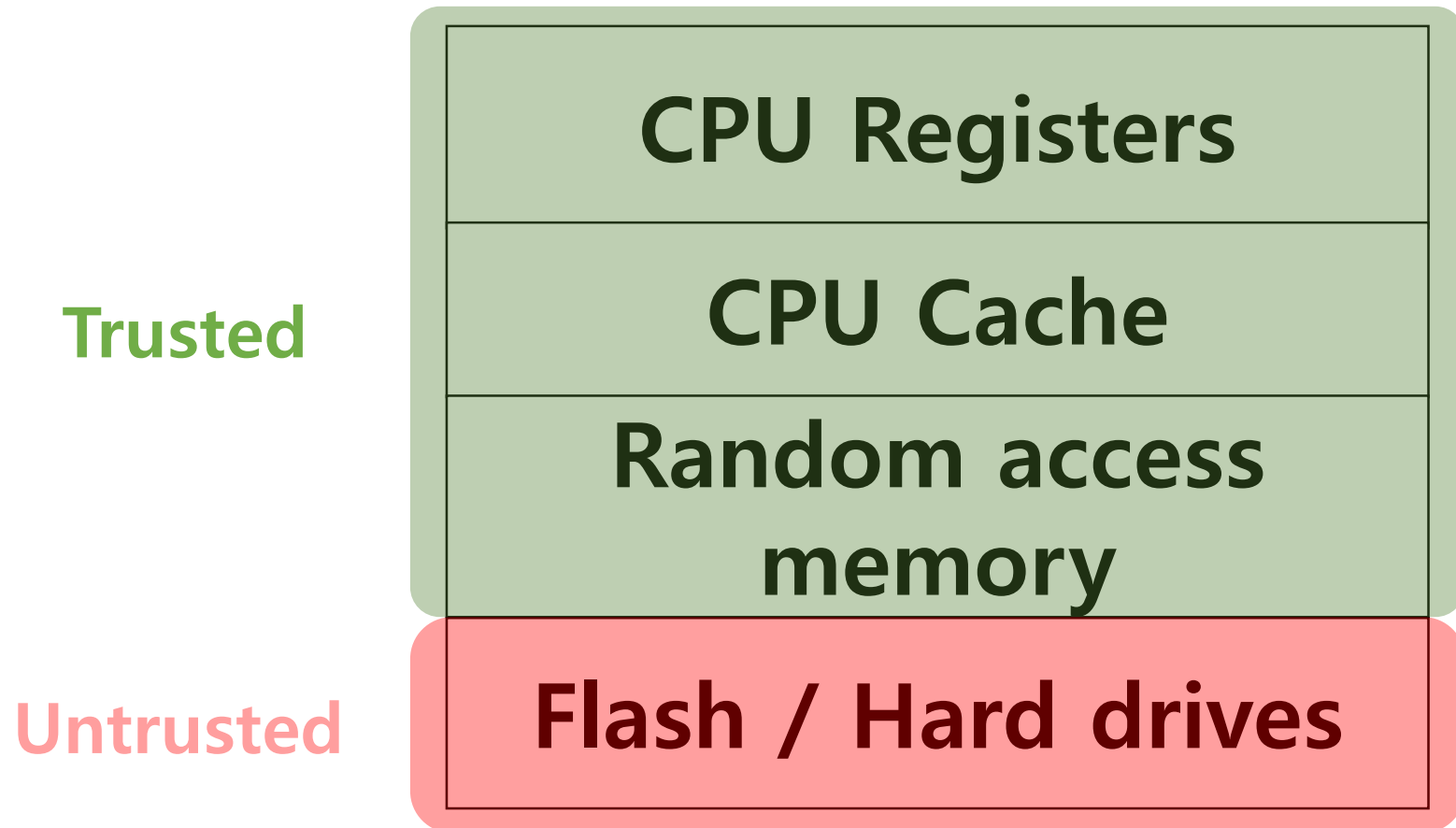
Trust Models in Encrypted FS

- Memory hierarchy with trust models (when OS is trusted)



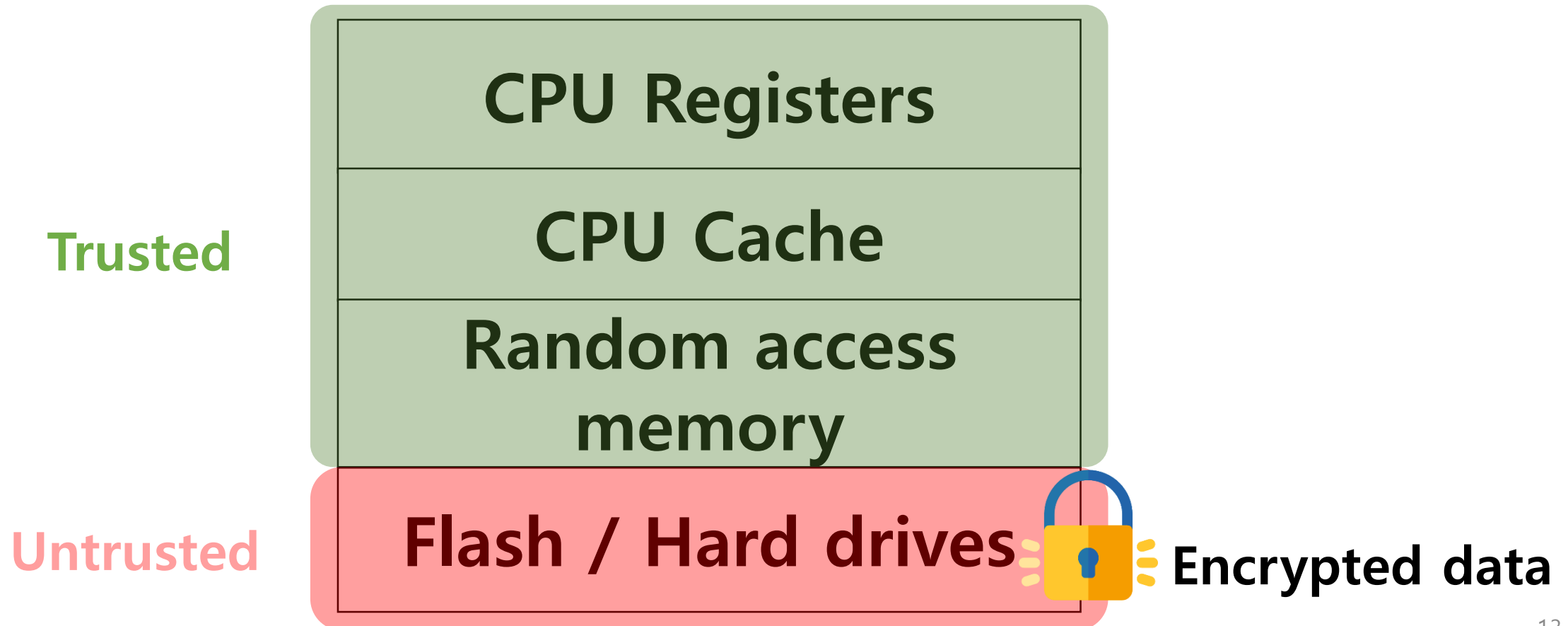
Trust Models in Encrypted FS

- Memory hierarchy with trust models (when OS is trusted)



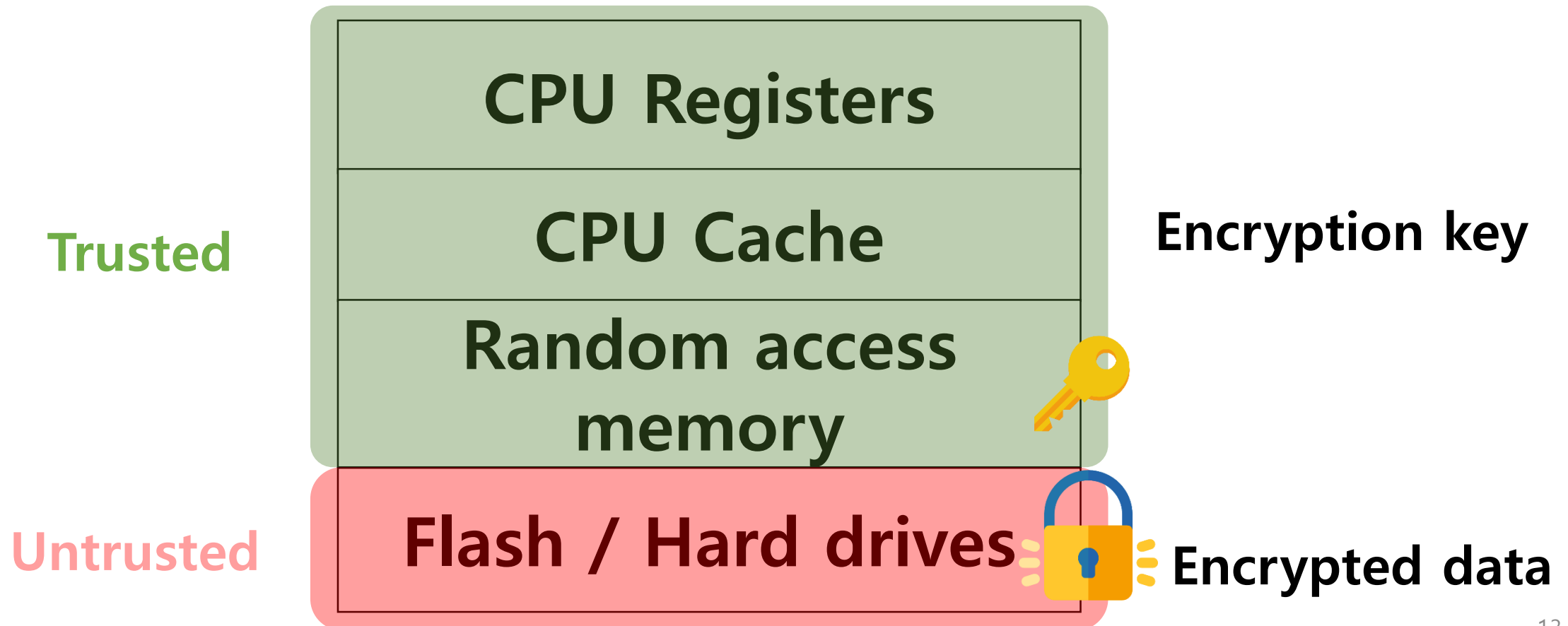
Trust Models in Encrypted FS

- Memory hierarchy with trust models (when OS is trusted)



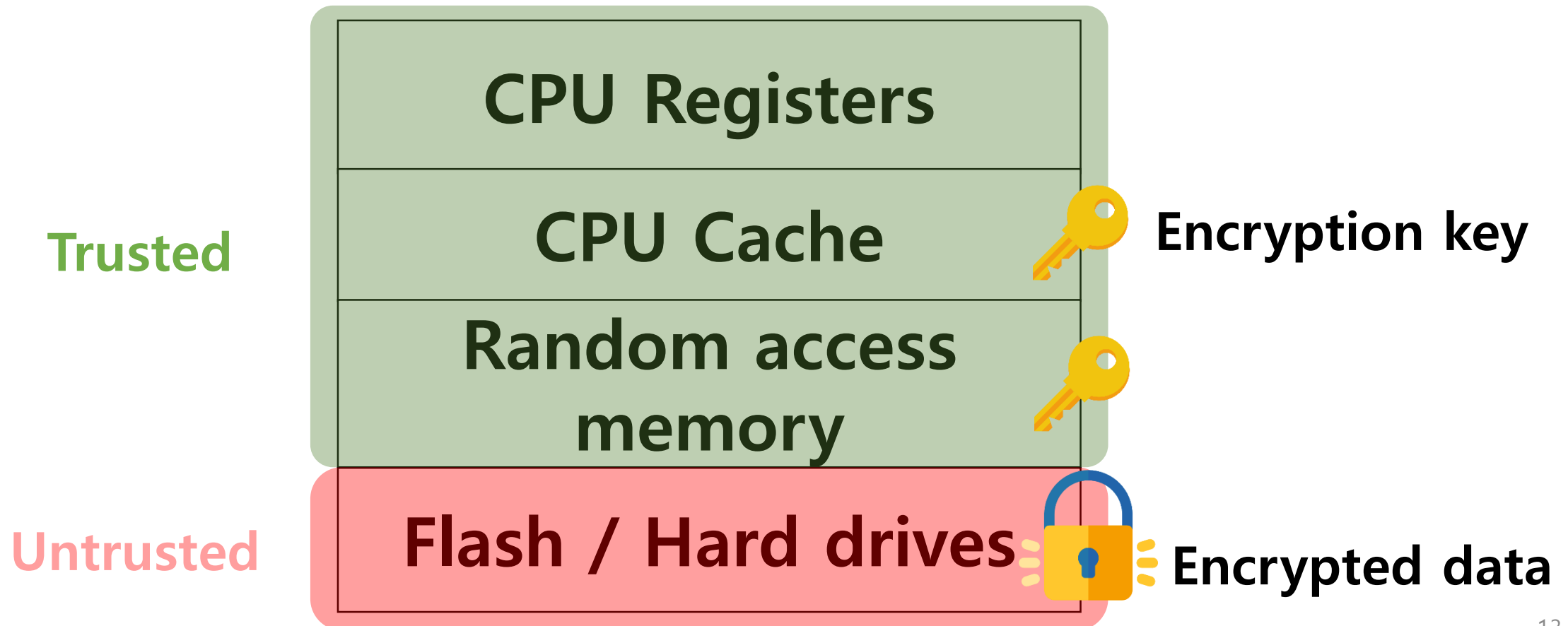
Trust Models in Encrypted FS

- Memory hierarchy with trust models (when OS is trusted)



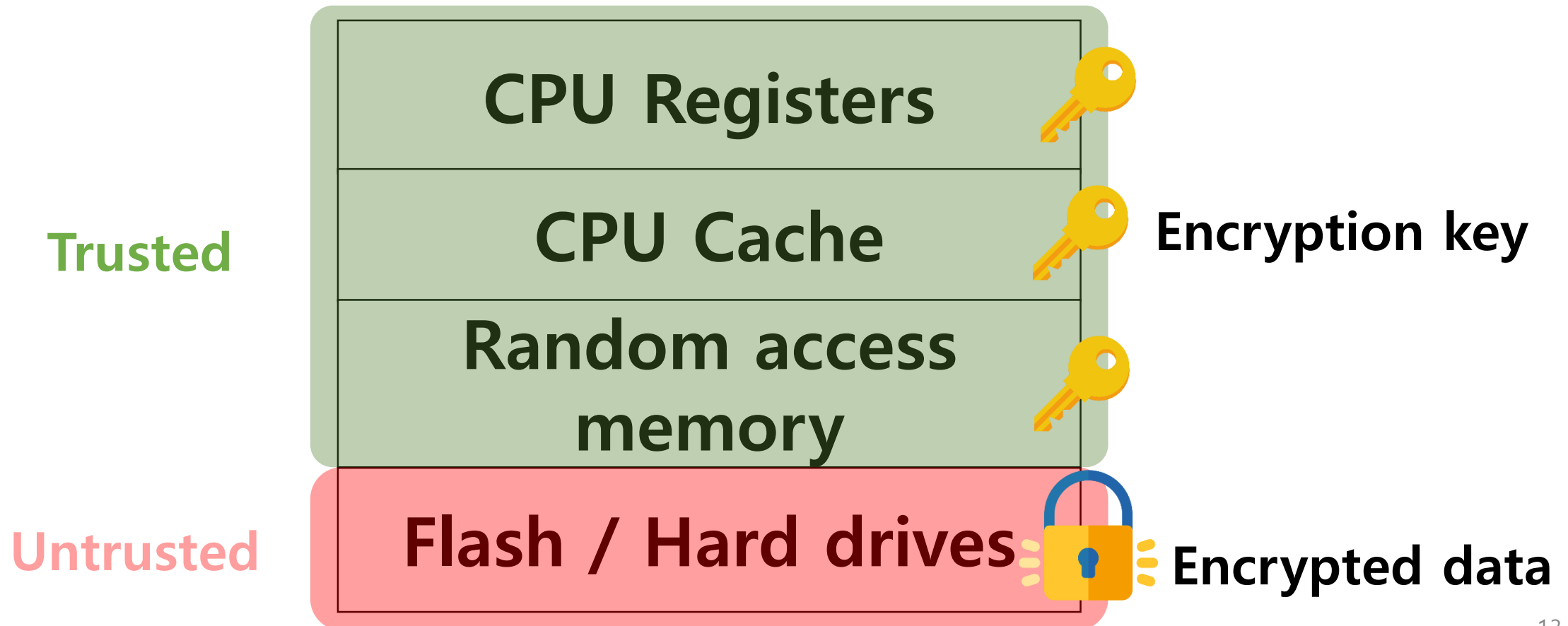
Trust Models in Encrypted FS

- Memory hierarchy with trust models (when OS is trusted)



Trust Models in Encrypted FS

- Memory hierarchy with trust models (when OS is trusted)



Attacking DRAM

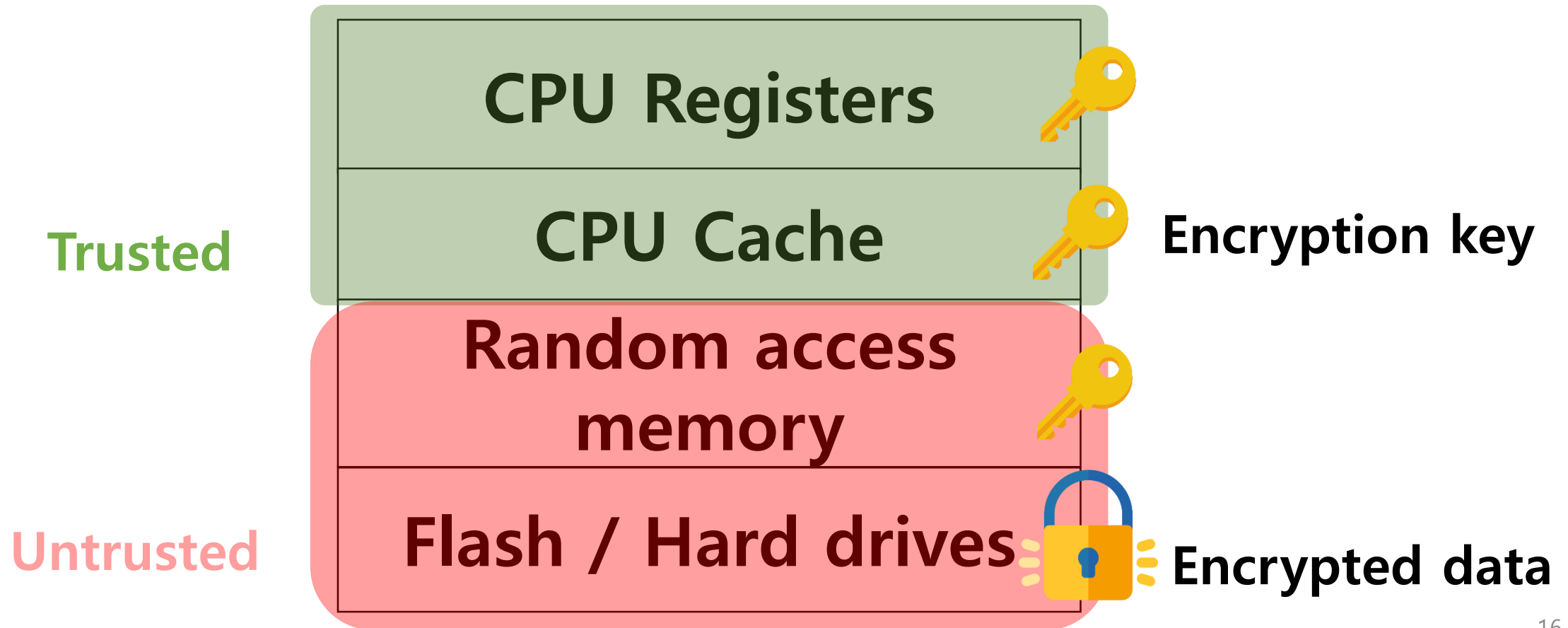
- Physical attacks against DRAM
 - What happen if DRAMs are detached from DIMM slots?
 - Should data be retained? Probably not.
 - DRAM cell has to be refreshed
 - If detached, a data value in a capacitor decays over time
- Can we slowdown decay?

Cold Boot Attack: Slowing Decay by Cooling

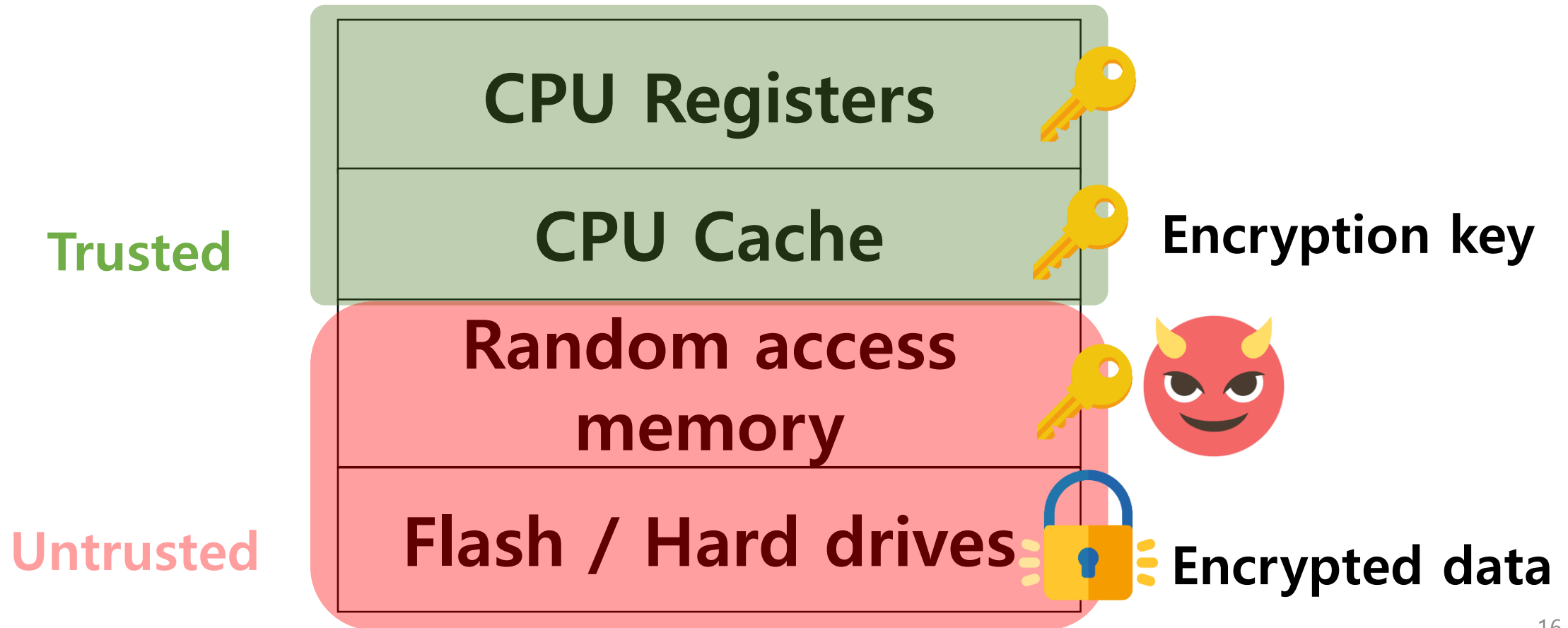


-50°C: less than 0.2% decay after 1 minute

Security Implications of Cold Boot Attack



Security Implications of Cold Boot Attack



Security Implications of Cold Boot Attack

- Encryption keys stored in DRAM can be leaked
- Demonstrated attacks in [USENIX Security 08]
 - Windows BitLocker
 - MacOS FileVault
 - Linux dm-crypt
 - Linux LoopAES
 - TrueCrypt

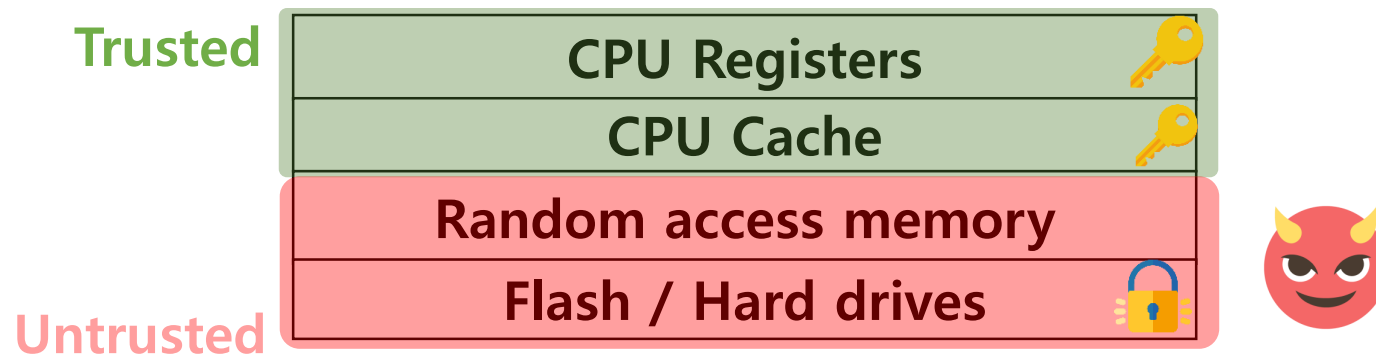
Security Implications of Cold Boot Attack

- Encryption keys stored in DRAM can be leaked
- Demonstrated attacks in [USENIX Security 08]
 - Windows BitLocker
 - MacOS FileVault
 - Linux dm-crypt
 - Linux LoopAES
 - TrueCrypt

“the emergence of **non-volatile DIMMs** that fit into DDR4 buses is going to **exacerbate the risk of cold boot attacks.**” [USENIX Security 08]

Countermeasures against Cold Boot Attack

- Encryption key and states only present in registers/cache
 - TRESOR [USENIX Security 11]
 - Linux kernel patch
 - The AES encryption algorithm and its key management solely on CPU



Countermeasures against Cold Boot Attack

- Sensitive data always leaves CPU as encrypted
 - Software-only solution
 - Leverage iRAM or locked L2 cache [ASPLOS 15]
 - Hardware solution ➔ Fully encrypted memory
 - Hardware-assisted Trusted Execution Environments
 - Intel SGX, AMD Secure Execution Environment, RISC-V Keystone [USENIX Security 16]
 - Encrypted channel
 - InvisiMem [ISCA 17]
 - ORAM-based memory controllers
 - ObfusMem [ISCA 17], SDIMM [HCPA 18]

Outline

Part1. Bugs in File Systems

Semantic inconsistency inference

Fuzzing

Part2. Attacks and Defenses

Ransomware

Cold boot attacks

Side-channels

Side-Channels

- Definition from Wikipedia

“Any attack based on information **gained from the implementation** of a computer system, rather than weaknesses in the implemented algorithm itself (e.g. cryptanalysis and software bugs)”

“Timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information, which can be exploited.”

Timing attacks

- This may happen for website login
 - Your (plain) password is compared at the server side

```
// php
$secret = "thisismykey";
if ($_GET['secret'] !== $secret) {
    die("Not Allowed!");
}
```

Timing attacks

- This may happen for website login
 - Your (plain) password is compared at the server side

```
// php
$secret = "thisismykey";
if ($_GET['secret'] !== $secret) {
    die("Not Allowed!");
}
```

```
case IS_STRING:
    if (Z_STR_P(op1) == Z_STR_P(op2)) {
        ZVAL_BOOL(result, 1);
    } else {
        ZVAL_BOOL(result, (Z_STRLEN_P(op1) == Z_STRLEN_P(op2))
            && (!memcmp(Z_STRVAL_P(op1), Z_STRVAL_P(op2), Z_STRLEN_P(op1))));
    }
    break;
```

Timing attacks

- This may happen for website login
 - Your (plain) password is compared at the server side

```
// php
$secret = "thisismykey";
if ($_GET['secret'] !== $secret) {
    die("Not Allowed!");
}
```

```
case IS_STRING:
    if (Z_STR_P(op1) == Z_STR_P(op2)) {
        ZVAL_BOOL(result, 1);
    } else {
        ZVAL_BOOL(result, (Z_STRLEN_P(op1)
        && (!memcmp(Z_STRVAL_P(op1), Z_ST
        }
    break;
```

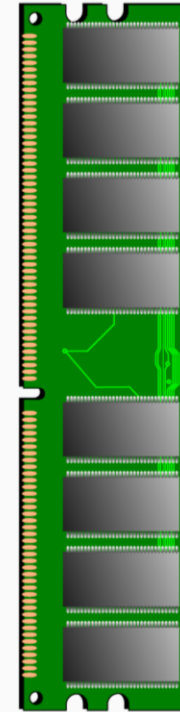
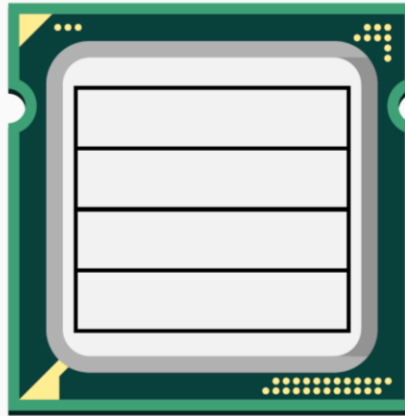
```
int memcmp(const void *s1, const void *s2, size_t n)
{
    unsigned char u1, u2;

    for ( ; n-- ; s1++, s2++) {
        u1 = * (unsigned char *) s1;
        u2 = * (unsigned char *) s2;
        if ( u1 != u2)
            return (u1-u2);
    }
    return 0;
}
```

Cache Side-Channel

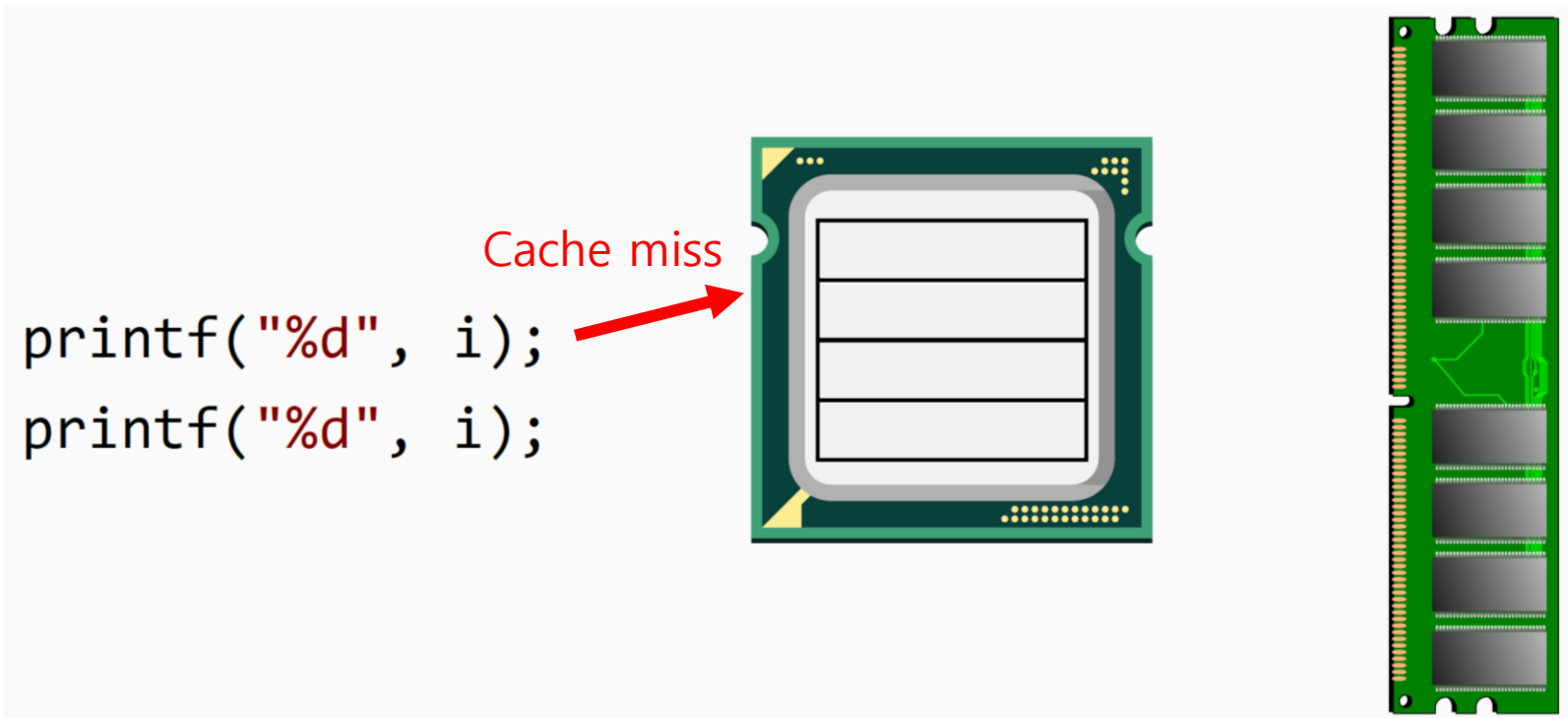
- Timing channels in CPU Cache

```
printf("%d", i);  
printf("%d", i);
```



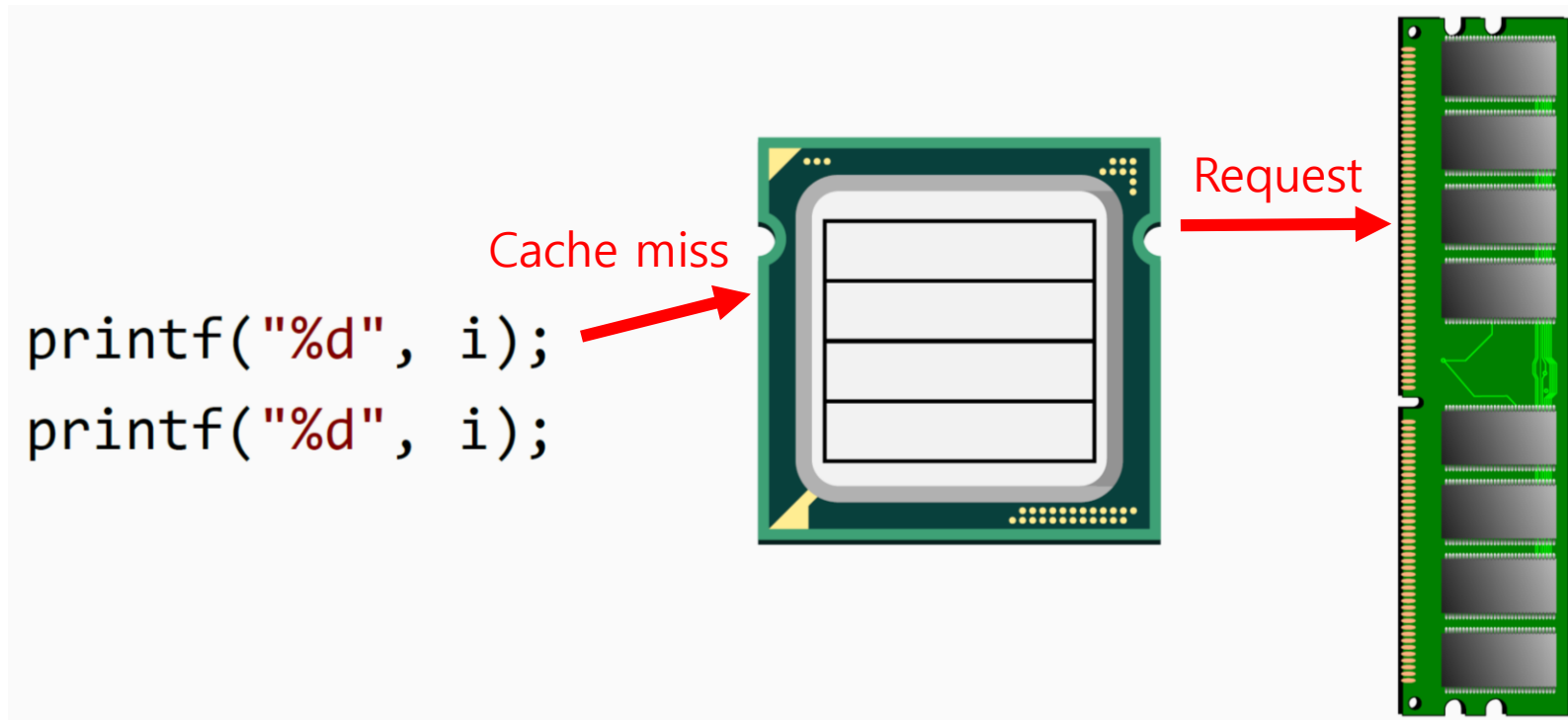
Cache Side-Channel

- Timing channels in CPU Cache



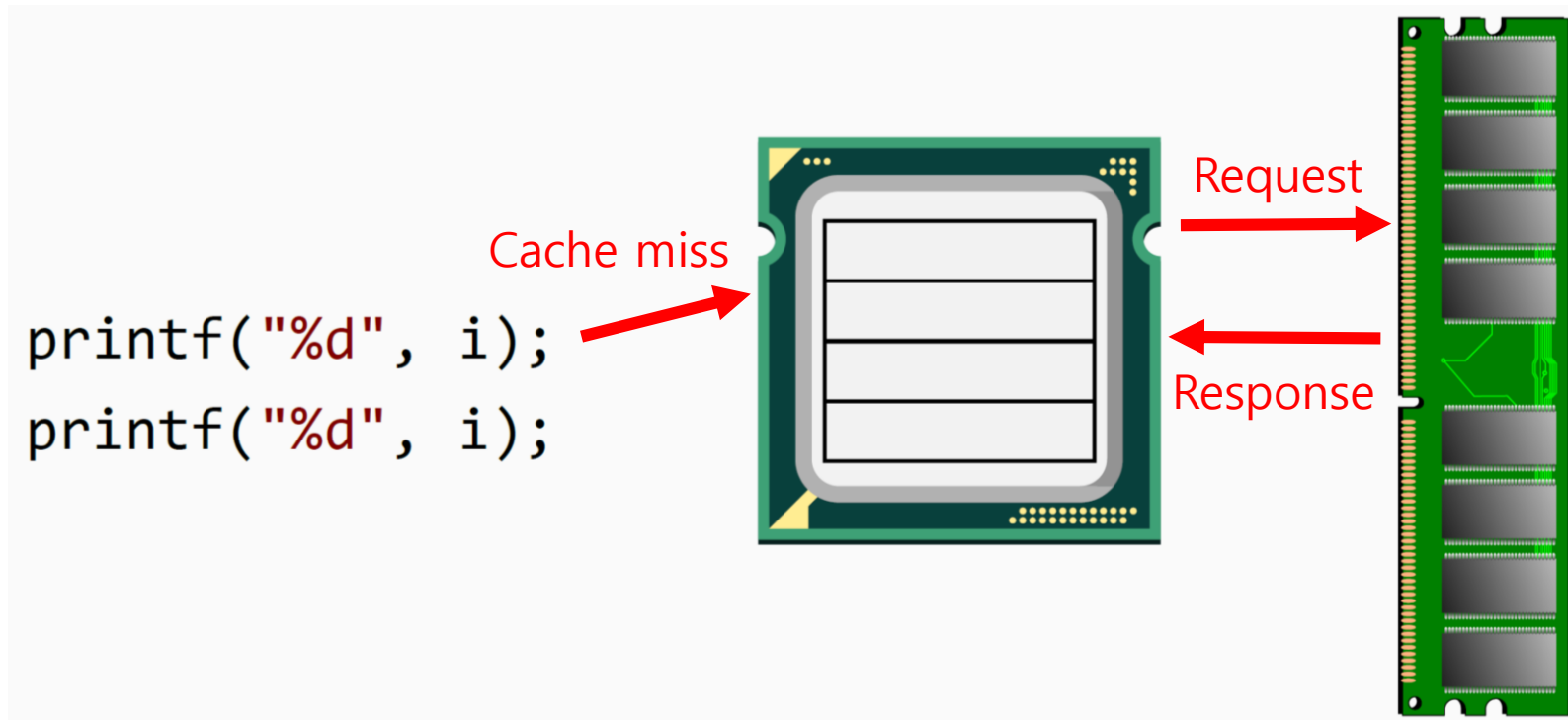
Cache Side-Channel

- Timing channels in CPU Cache



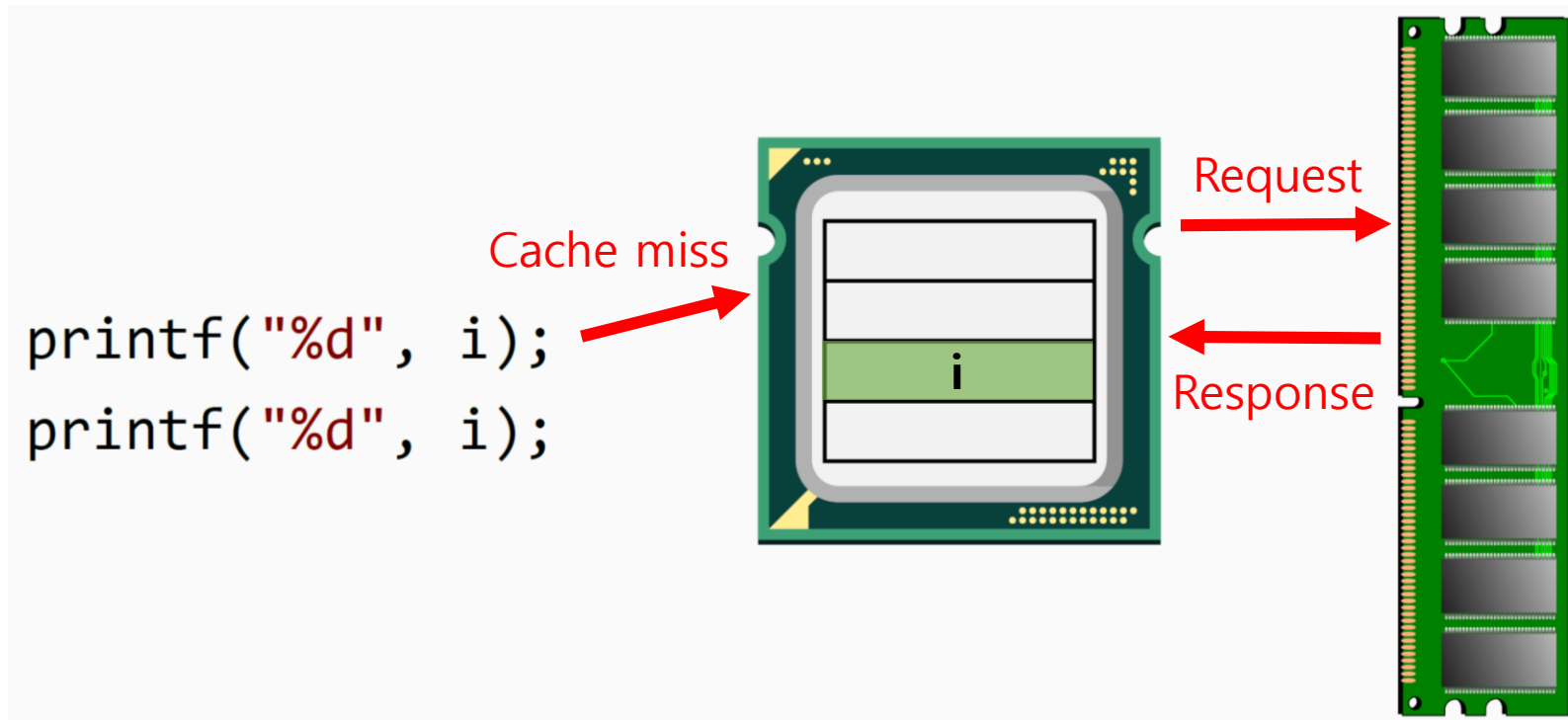
Cache Side-Channel

- Timing channels in CPU Cache



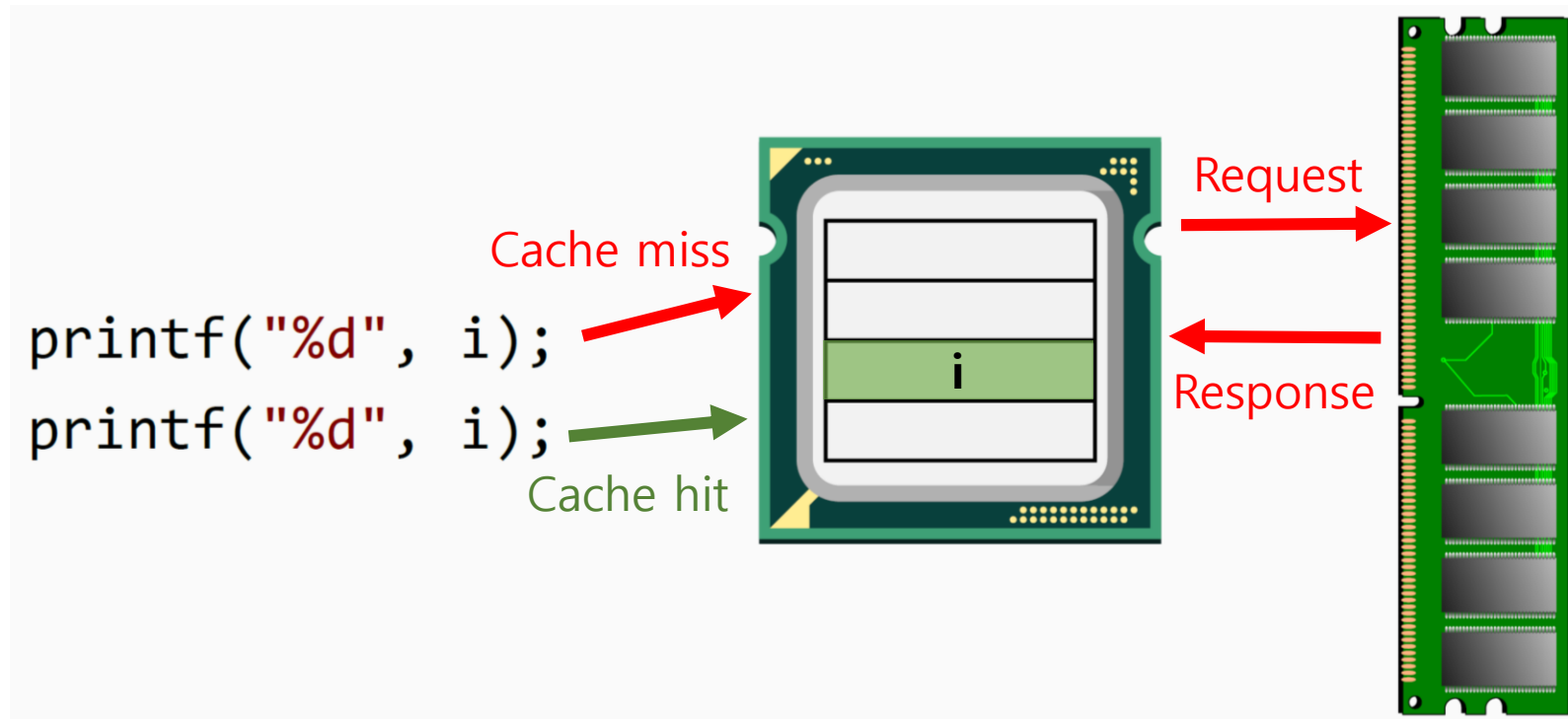
Cache Side-Channel

- Timing channels in CPU Cache



Cache Side-Channel

- Timing channels in CPU Cache



Cache Side-Channel

- Timing channels in CPU Cache

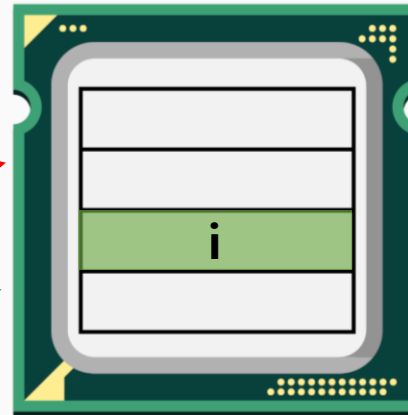
DRAM access,
slow

```
printf("%d", i);
```

```
printf("%d", i);
```

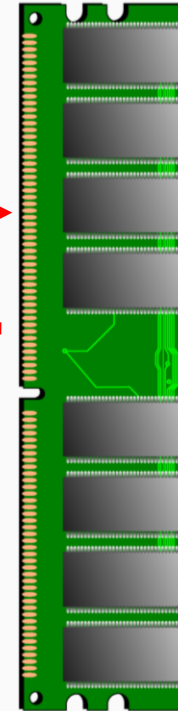
Cache miss

Cache hit



Request

Response



Cache Side-Channel

- Timing channels in CPU Cache

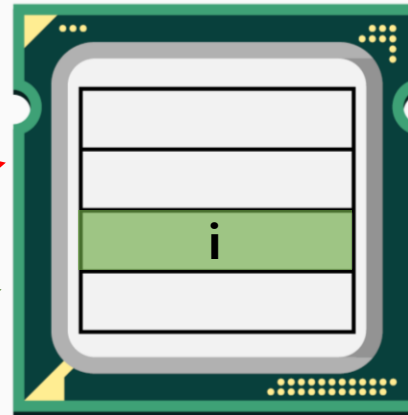
DRAM access,
slow

```
printf("%d", i);  
printf("%d", i);
```

No DRAM access,
much faster

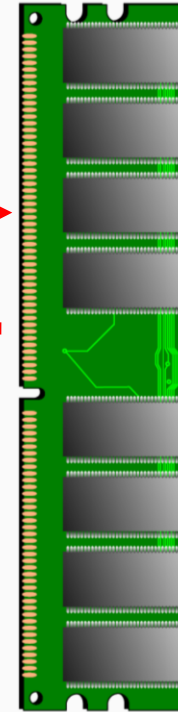
Cache miss

Cache hit



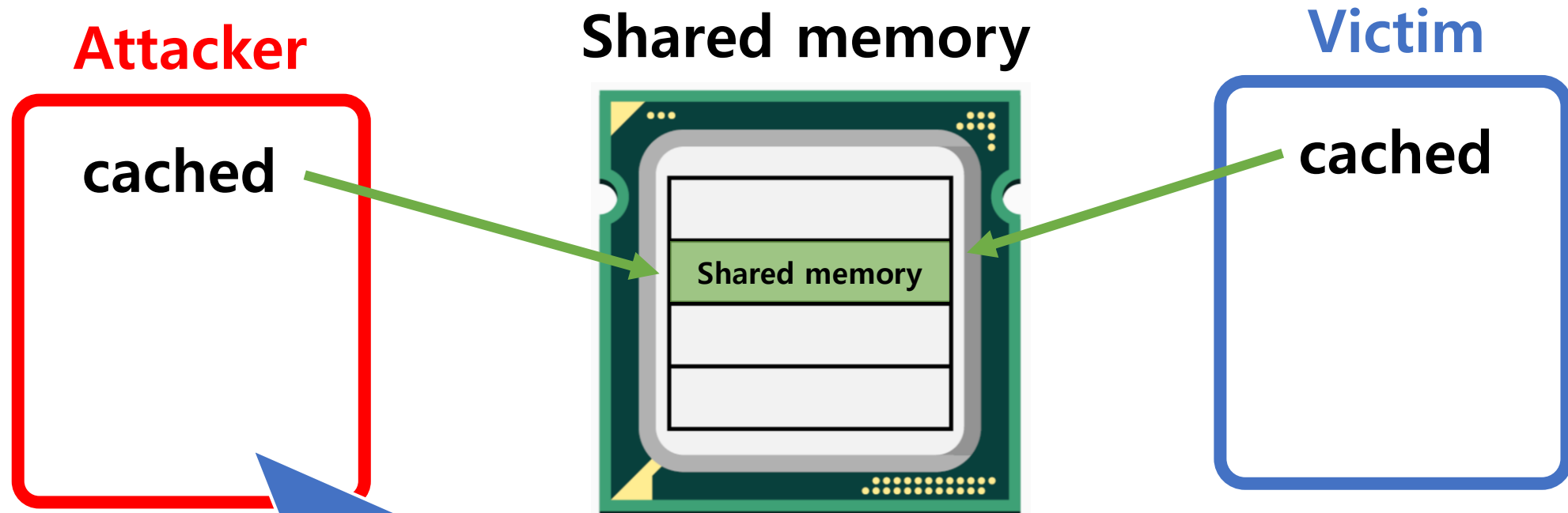
Request

Response



Cache Side-Channel

- Flush+Reload attack

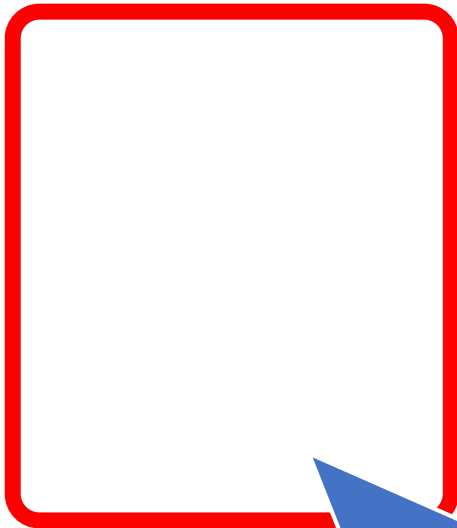


Fast if victim accessed data,
slow otherwise

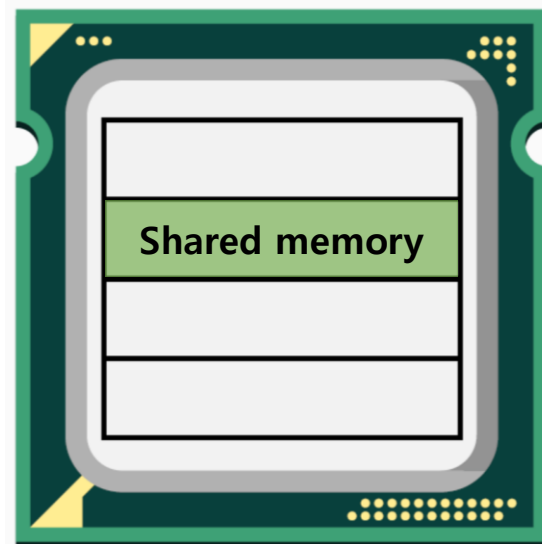
Cache Side-Channel

- Flush+Reload attack

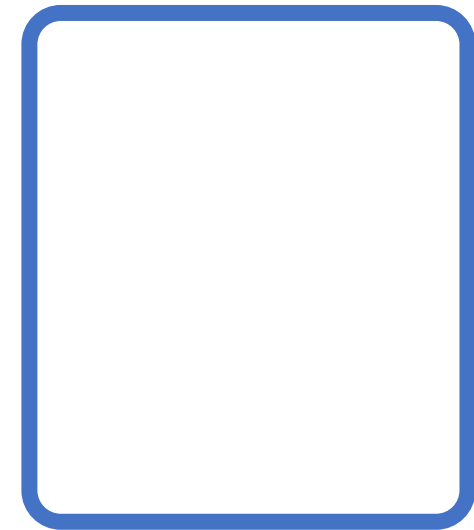
Attacker



Shared memory



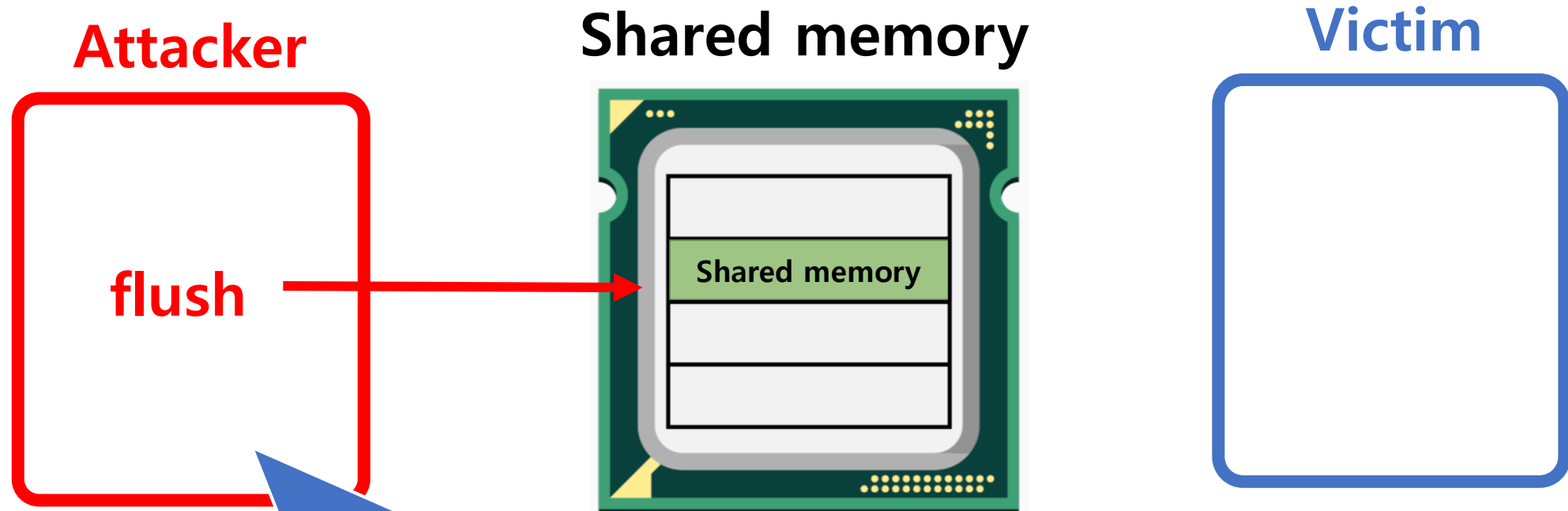
Victim



Fast if victim accessed data,
slow otherwise

Cache Side-Channel

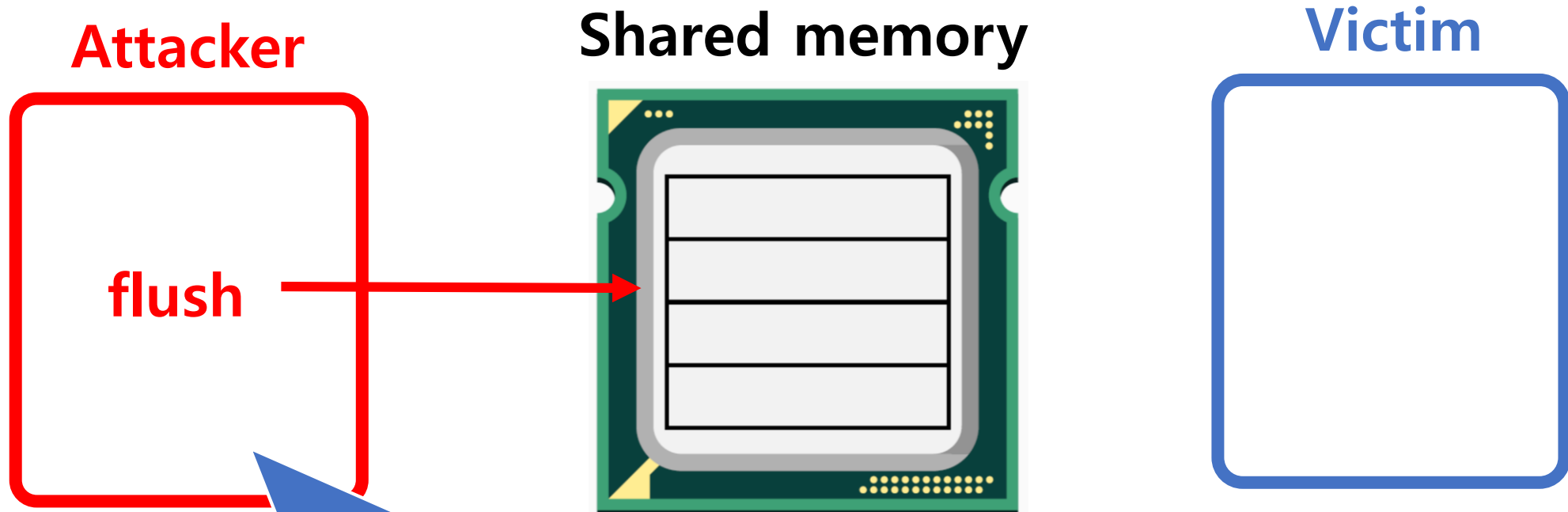
- Flush+Reload attack



Fast if victim accessed data,
slow otherwise

Cache Side-Channel

- Flush+Reload attack

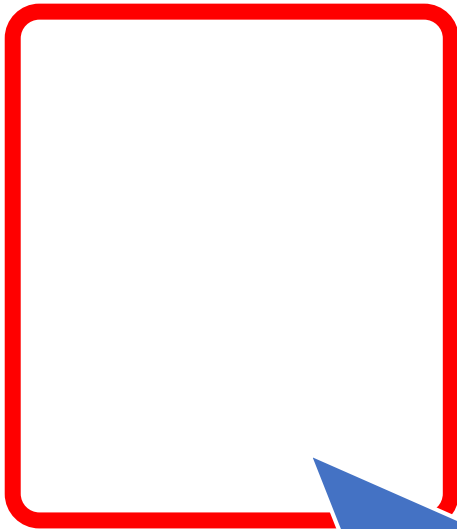


Fast if victim accessed data,
slow otherwise

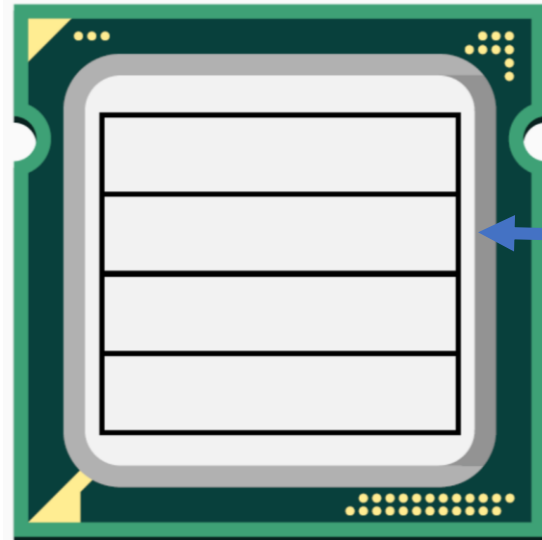
Cache Side-Channel

- Flush+Reload attack

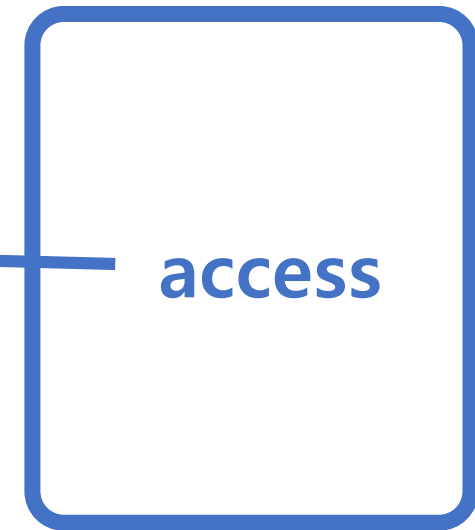
Attacker



Shared memory



Victim



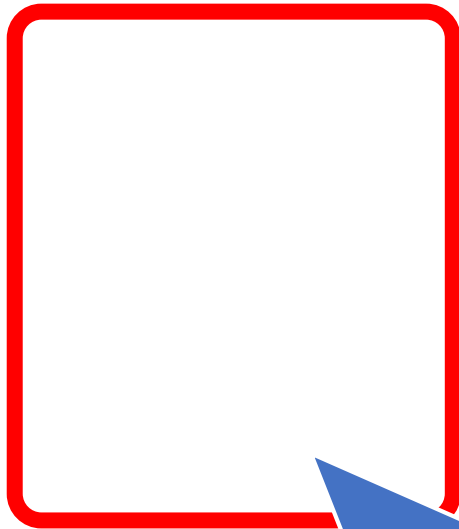
access

Fast if victim accessed data,
slow otherwise

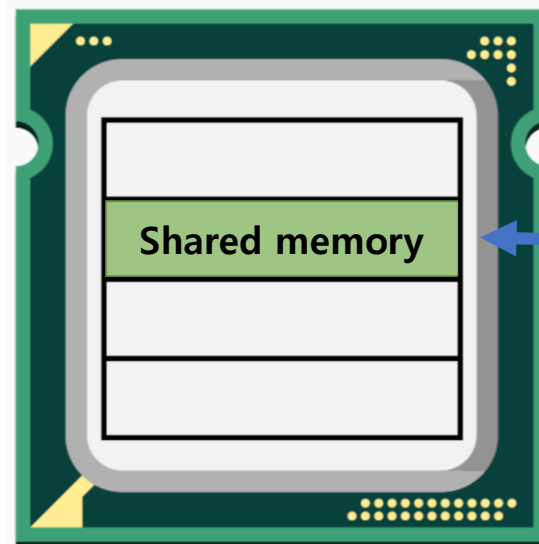
Cache Side-Channel

- Flush+Reload attack

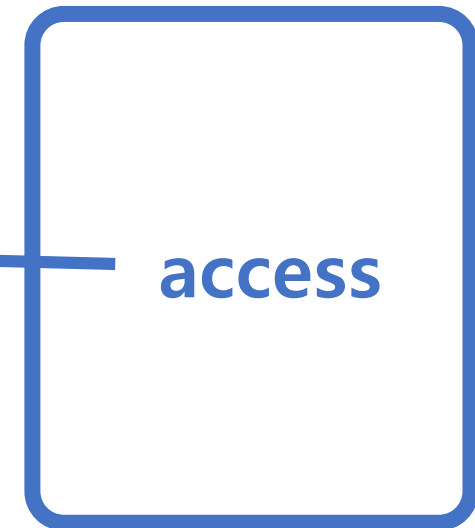
Attacker



Shared memory



Victim



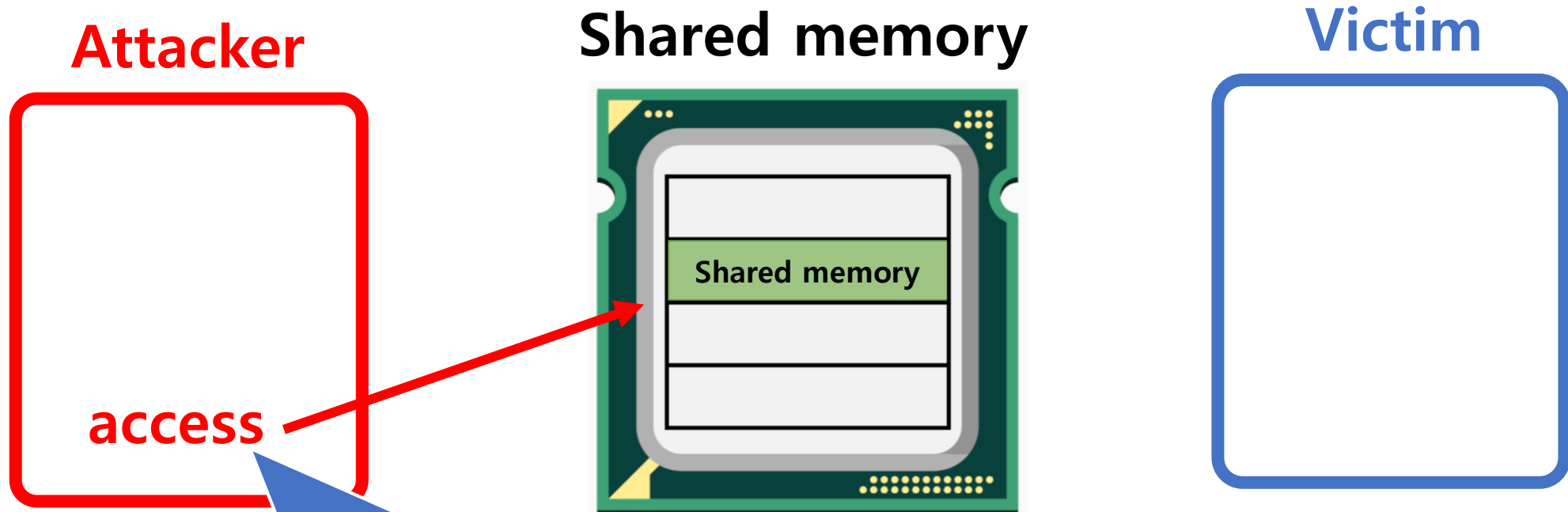
access



Fast if victim accessed data,
slow otherwise

Cache Side-Channel

- Flush+Reload attack



Fast if victim accessed data,
slow otherwise

Meltdown: Out-of-order execution

- Out-of-order execution
 - Out-of-order instructions leave micro-architectural traces
 - Storing values in cache
 - Give such instructions a name: **transient instructions**

Meltdown

- Permission check for transient instructions is only done
 - when committing them
- Suppose we are running a user-level program below

```
char data = *(char*)0xffffffff81a000e0;  
array[data * 4096] = 0;
```

Meltdown

- Permission check for transient instructions is only done
 - when committing them
- Suppose we are running a user-level program below

```
char data = *(char*)0xffffffff81a000e0;  
array[data * 4096] = 0;
```

Meltdown

- Permission check for transient instructions is only done
 - when committing them
- Suppose we are running a user-level program below

Fetching a kernel address.
Should not be allowed.

```
char data = *(char*)0xffffffff81a000e0;  
array[data * 4096] = 0;
```

Meltdown

- Permission check for transient instructions is only done
 - when committing them
- Suppose we are running a user-level program below

Fetching a kernel address.
Should not be allowed.

Permission checks will be done later

```
char data = *(char*)0xffffffff81a000e0;  
array[data * 4096] = 0;
```


Meltdown

- Permission check for transient instructions is only done
 - when committing them
- Suppose we are running a user-level program below

Fetching a kernel address.
Should not be allowed.

Permission checks will be done later

```
char data = *(char*)0xffffffff81a000e0;
```

```
array[data * 4096] = 0;
```

Meltdown

- Permission check for transient instructions is only done
 - when committing them
- Suppose we are running a user-level program below

Fetching a kernel address.
Should not be allowed.

Permission checks will be done later

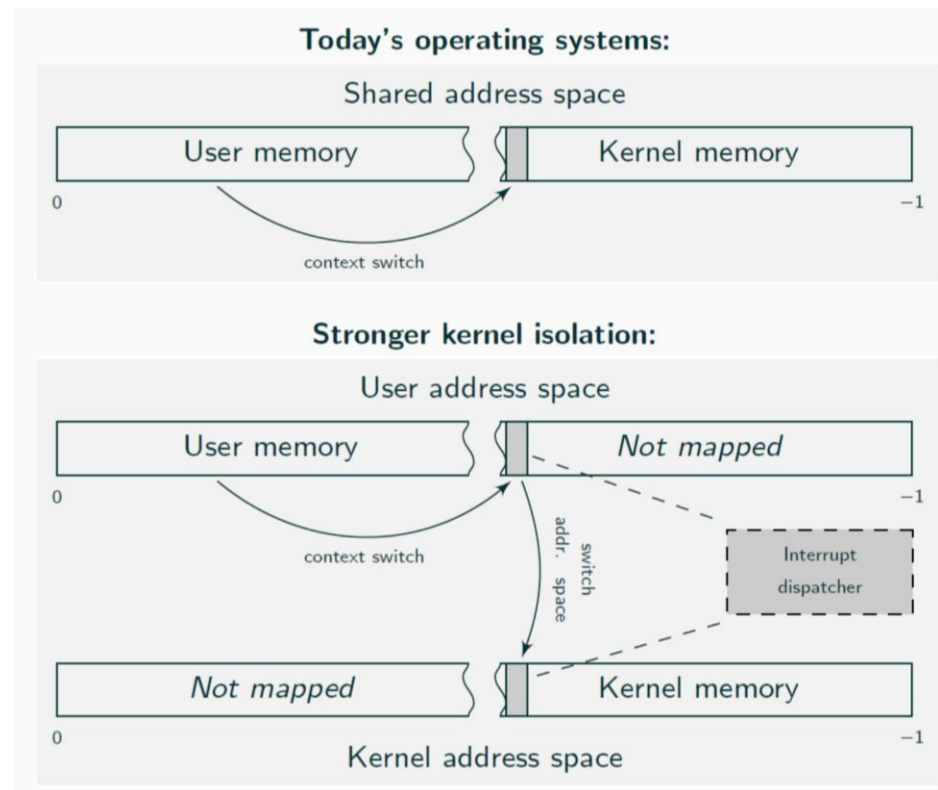
```
char data = *(char*)0xffffffff81a000e0;
```

```
array[data * 4096] = 0;
```

kernel's data value will be stored in array, which
can be retrieved using flush+reload

Mitigating Meltdown

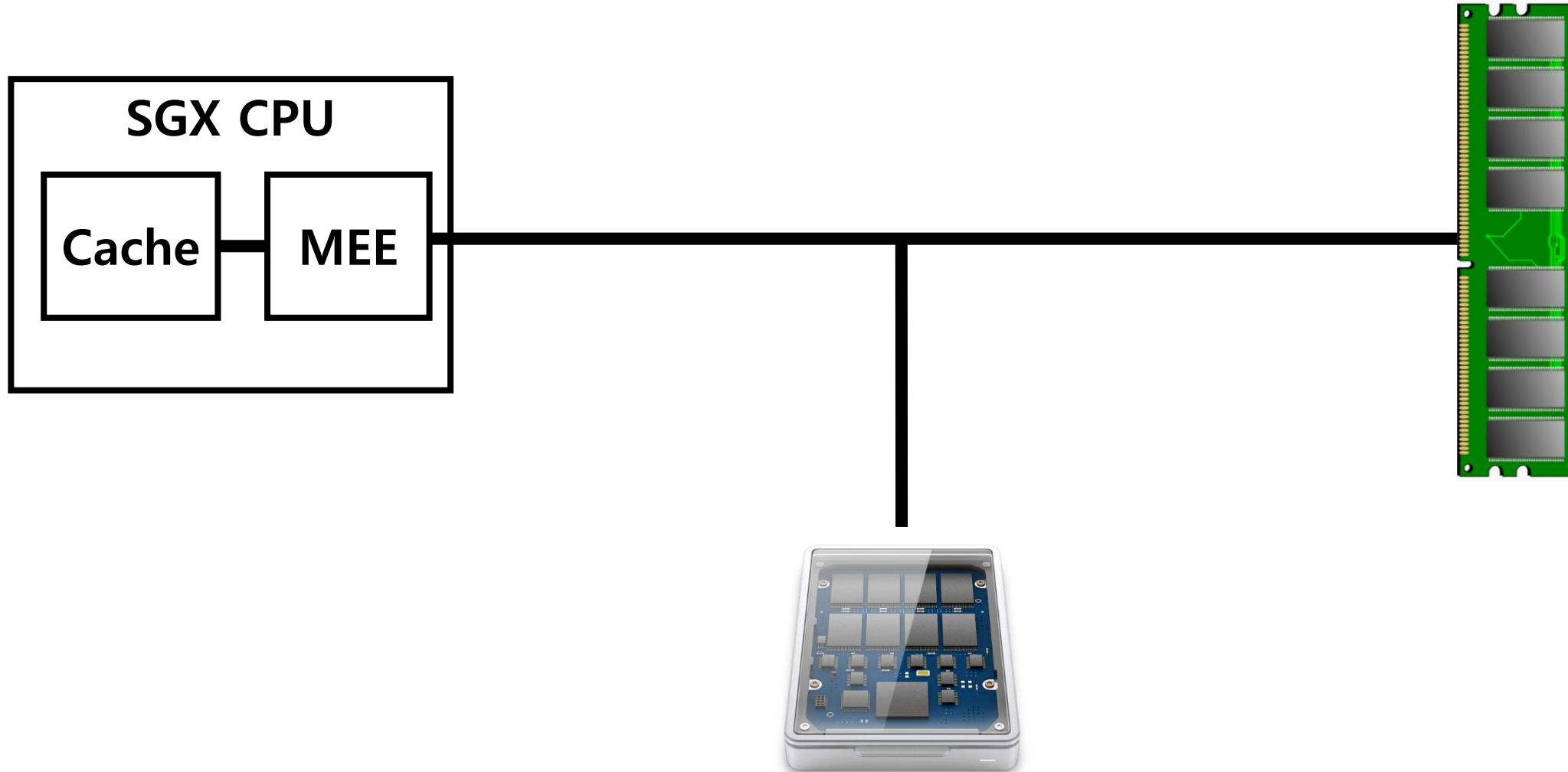
- Kernel Page Table Isolation
 - KAISER [ESSoS 17]



Side Channels in SGX

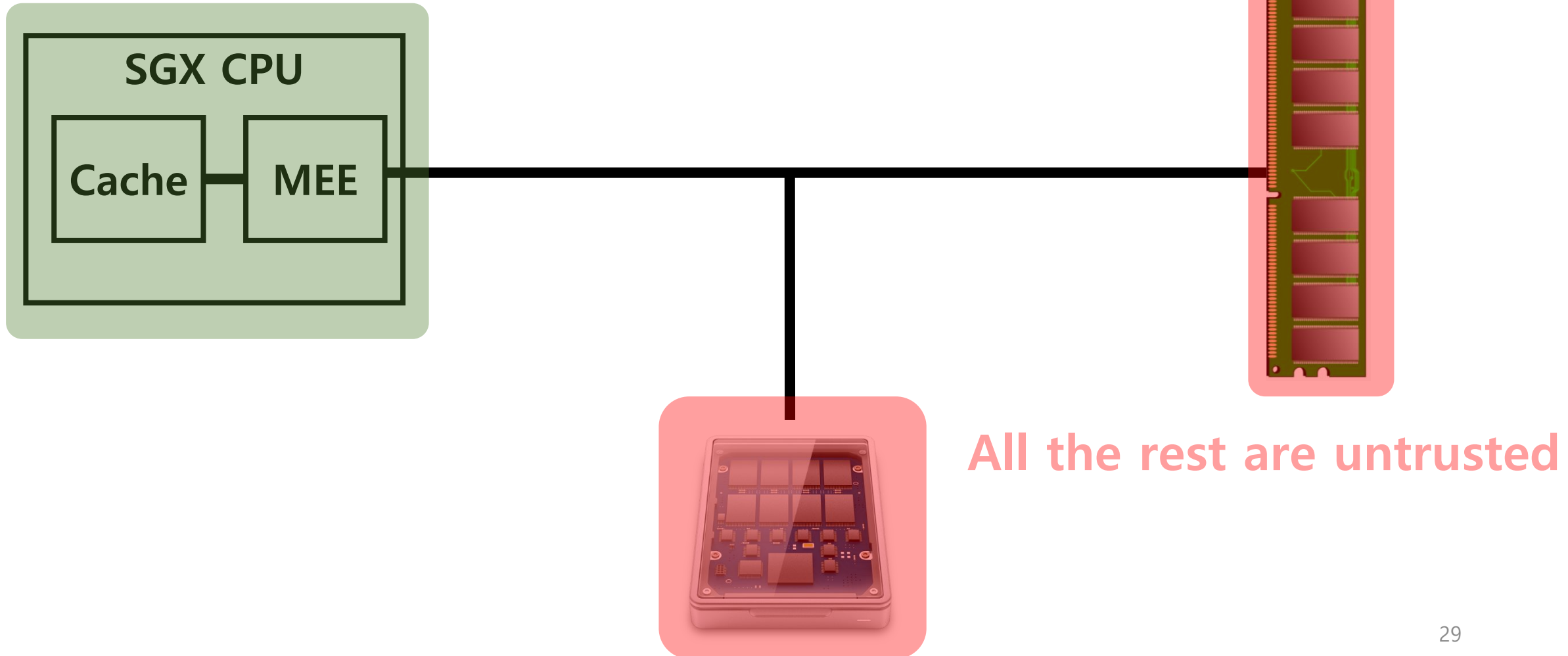
- Page fault
 - Controlled Channel Attack [S&P 15]
 - Cache
 - Software Grand Exposure [WOOT 17]
 - Branch prediction
 - Branch shadowing [Security 17]
 - Transient out-of-order execution
 - Foreshadow [Security 18]
 - Bus snooping
- ➔ All of these are about memory access

SGX's Threat Model



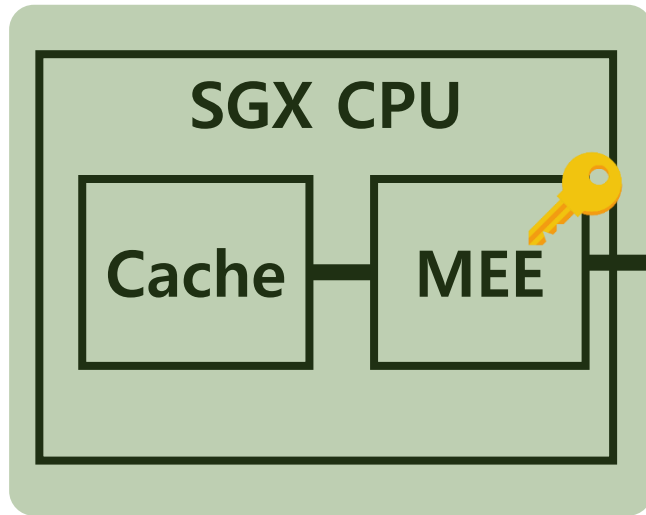
SGX's Threat Model

Only CPU is trusted

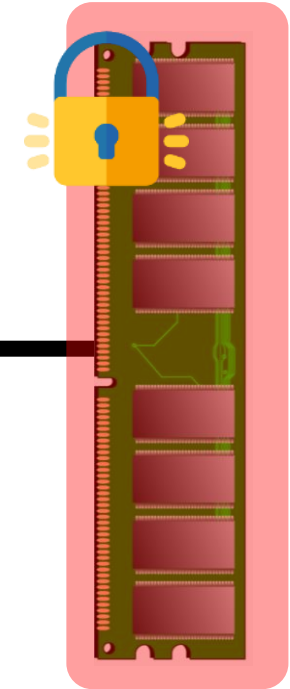


SGX's Threat Model

Only CPU is trusted

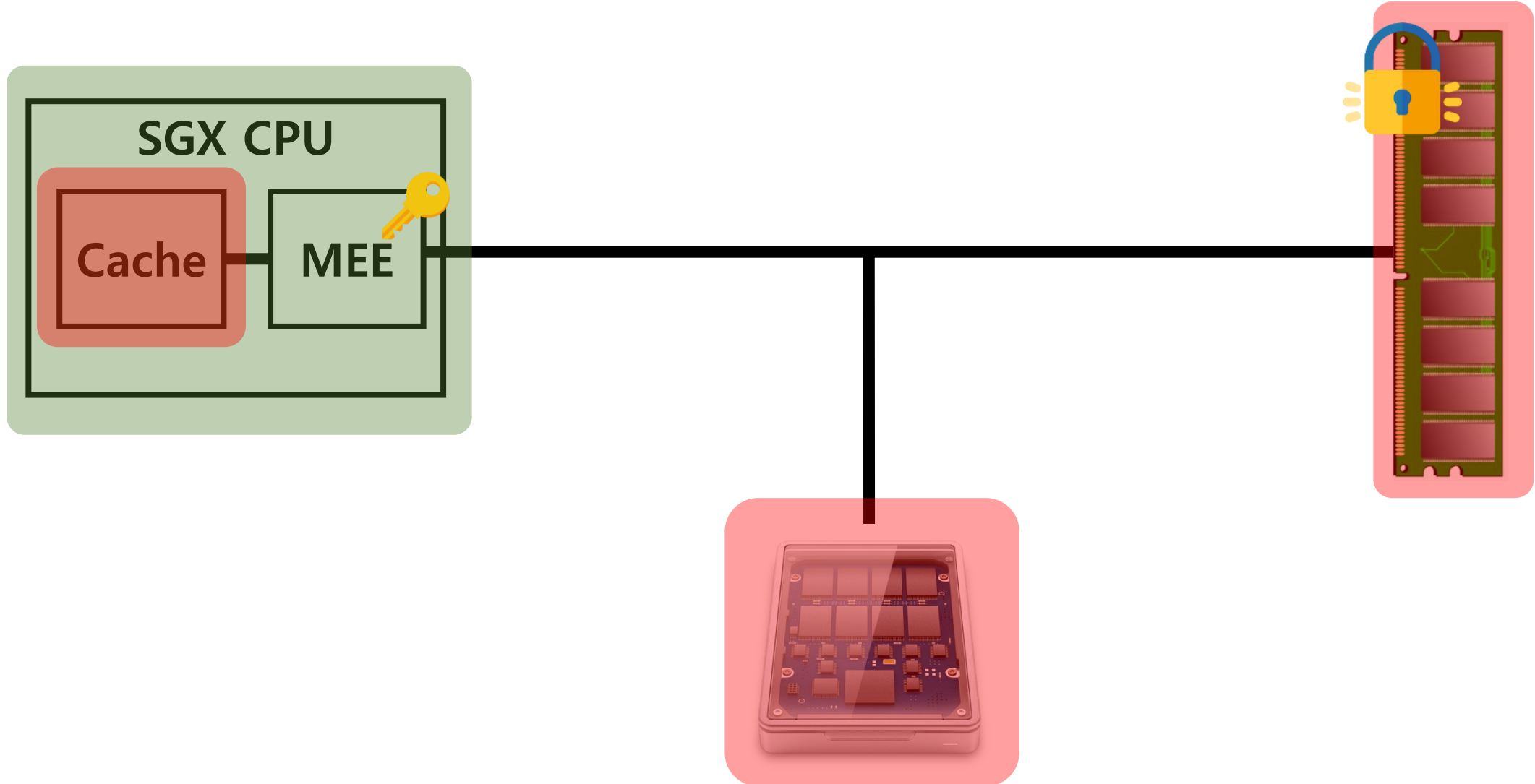


Any data leaving CPU is encrypted by Memory Encryption Engine (MEE)

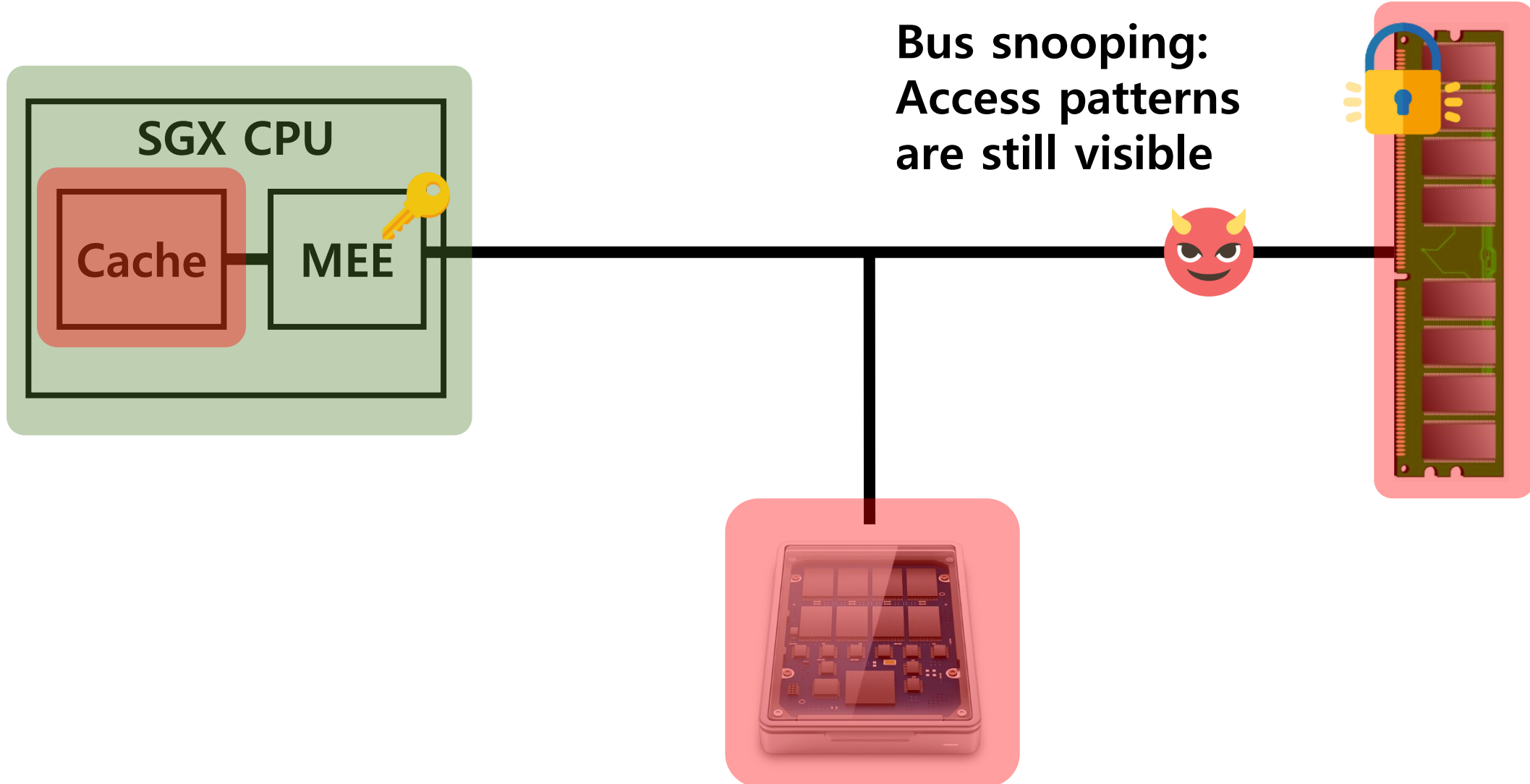


All the rest are untrusted

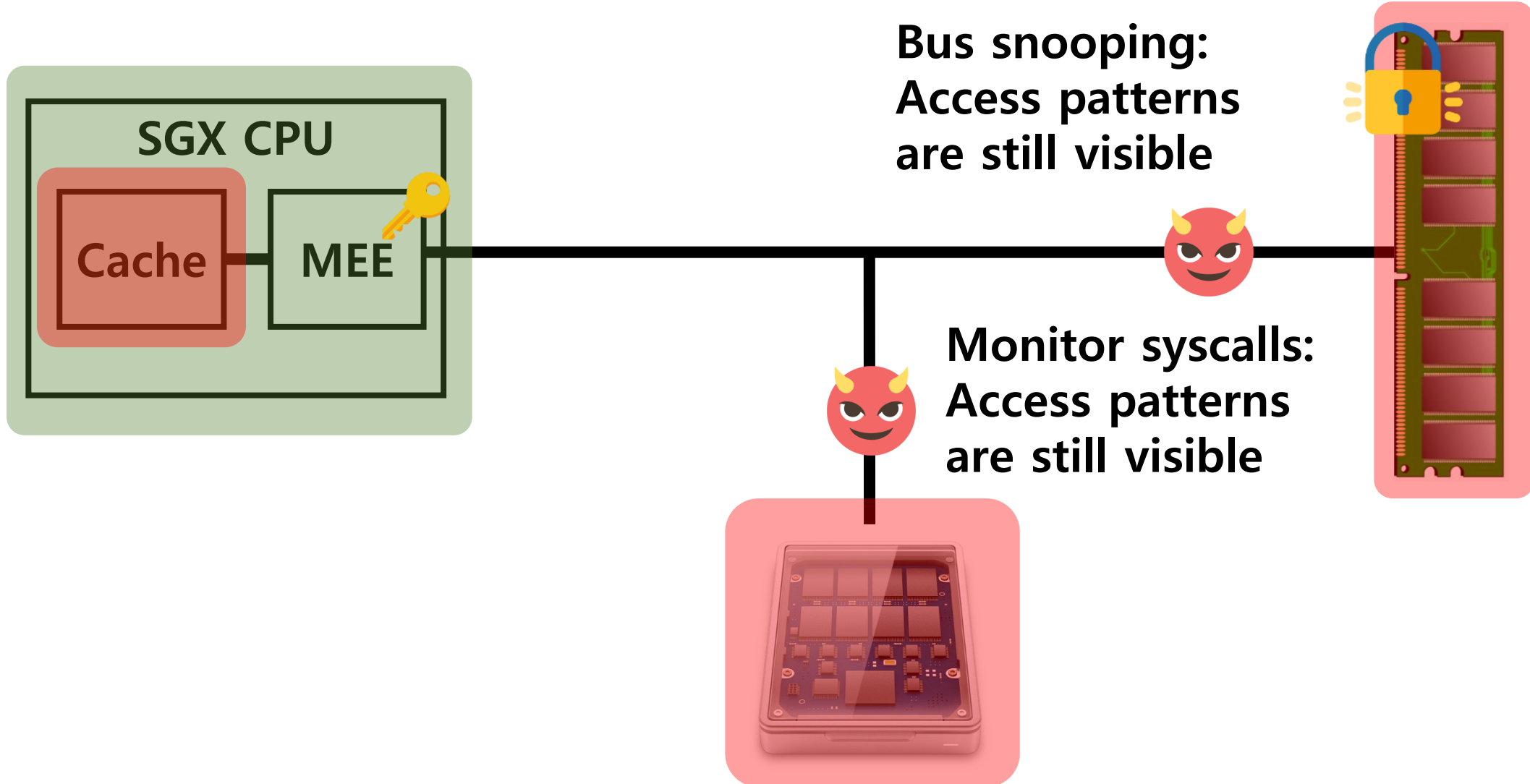
Attacking SGX



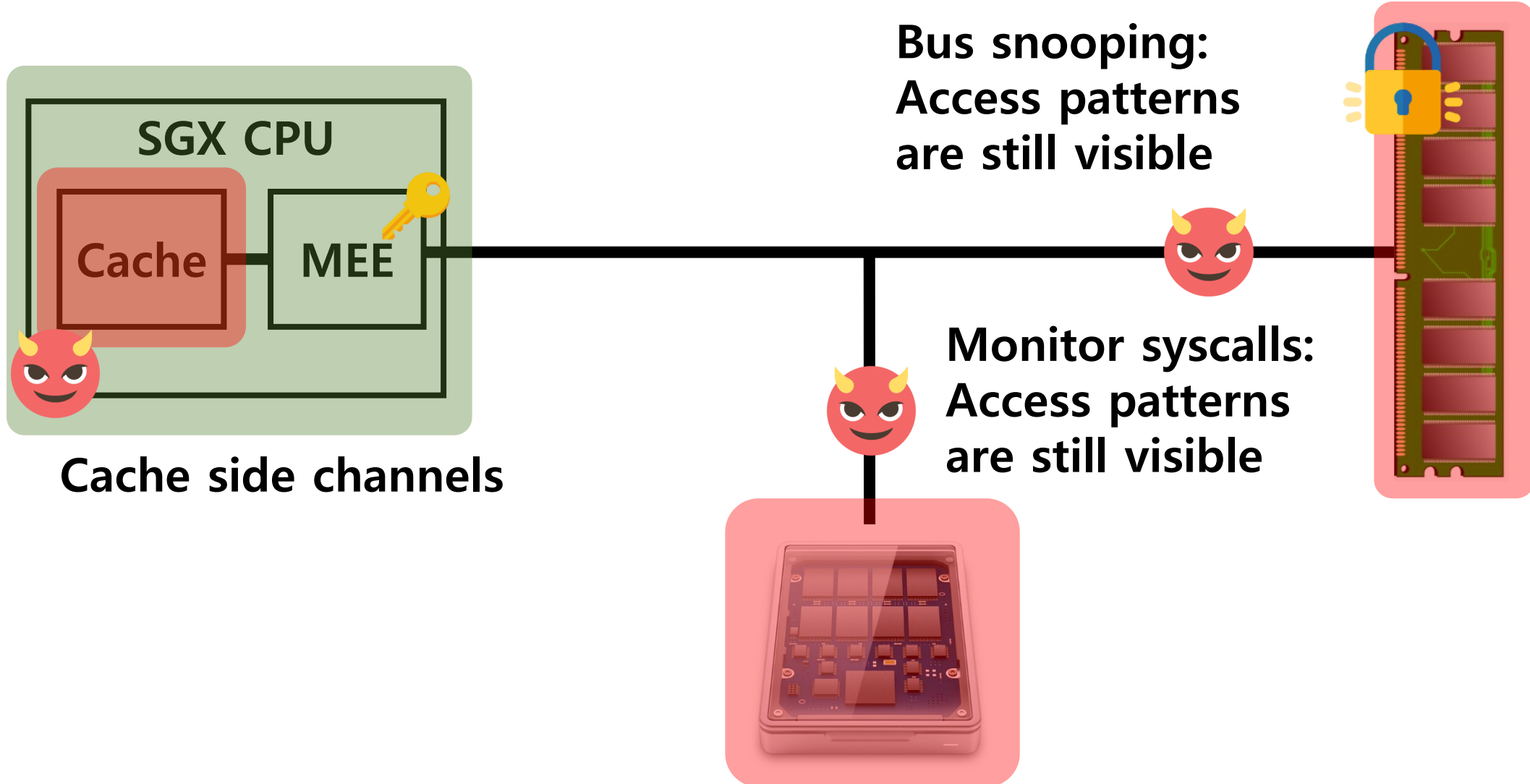
Attacking SGX



Attacking SGX

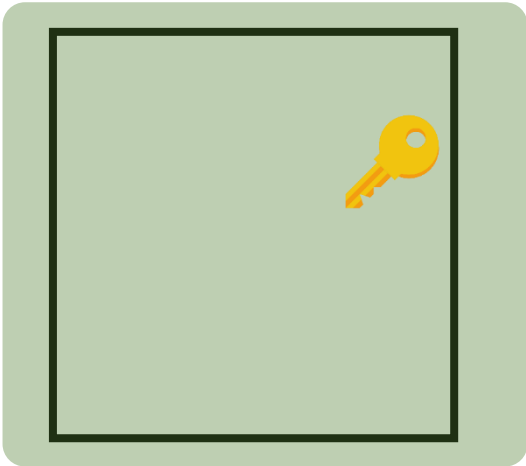


Attacking SGX

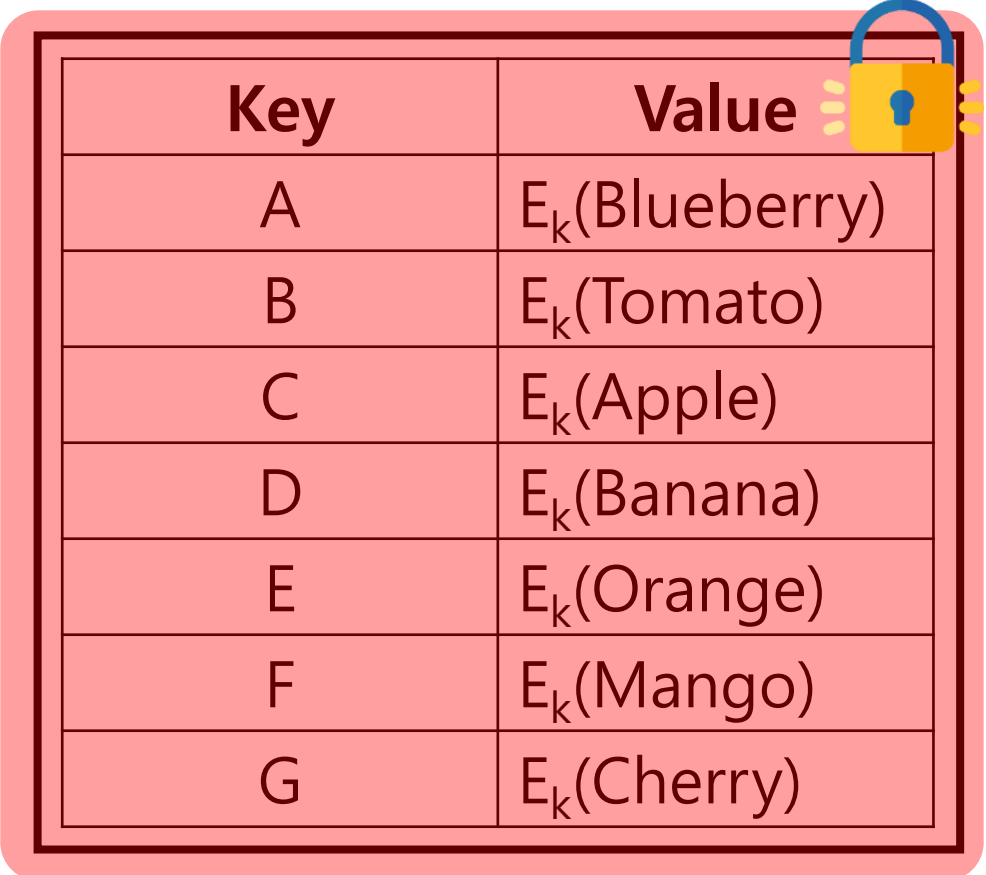


Why Does Access Patterns Matter?

Client

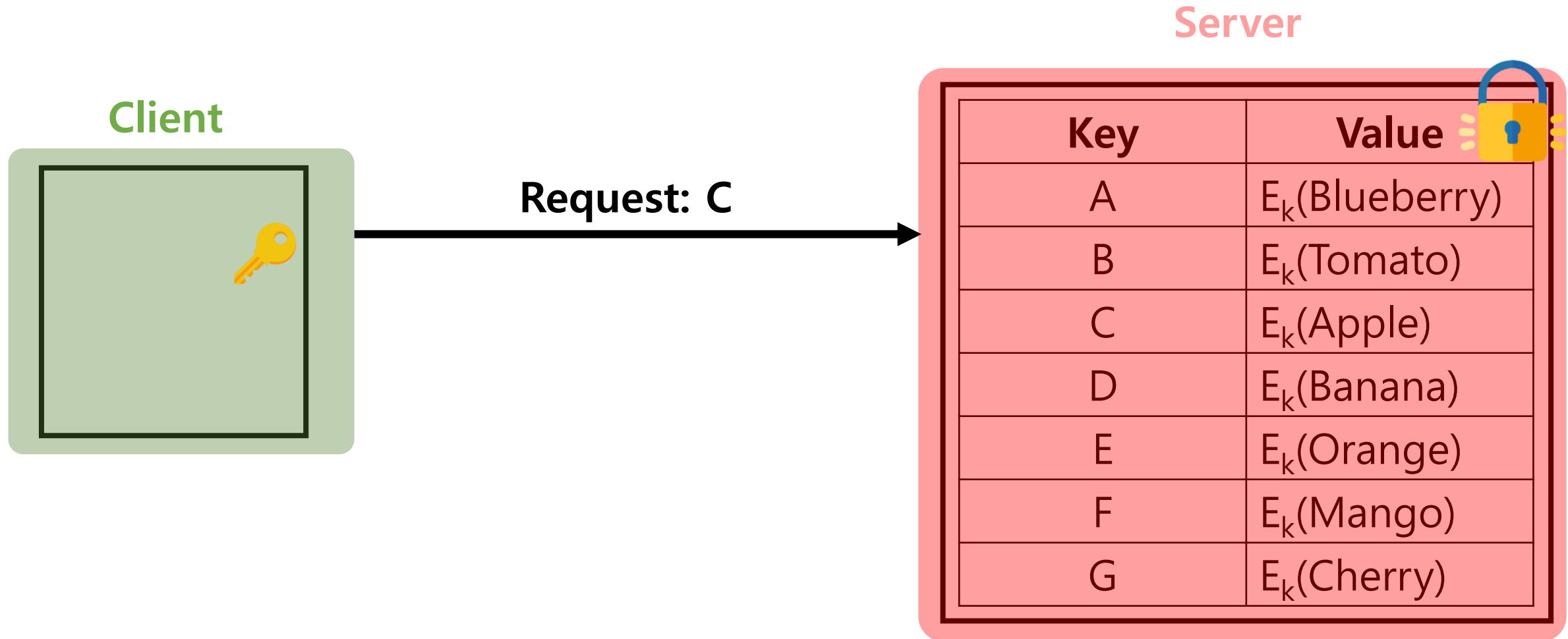


Server

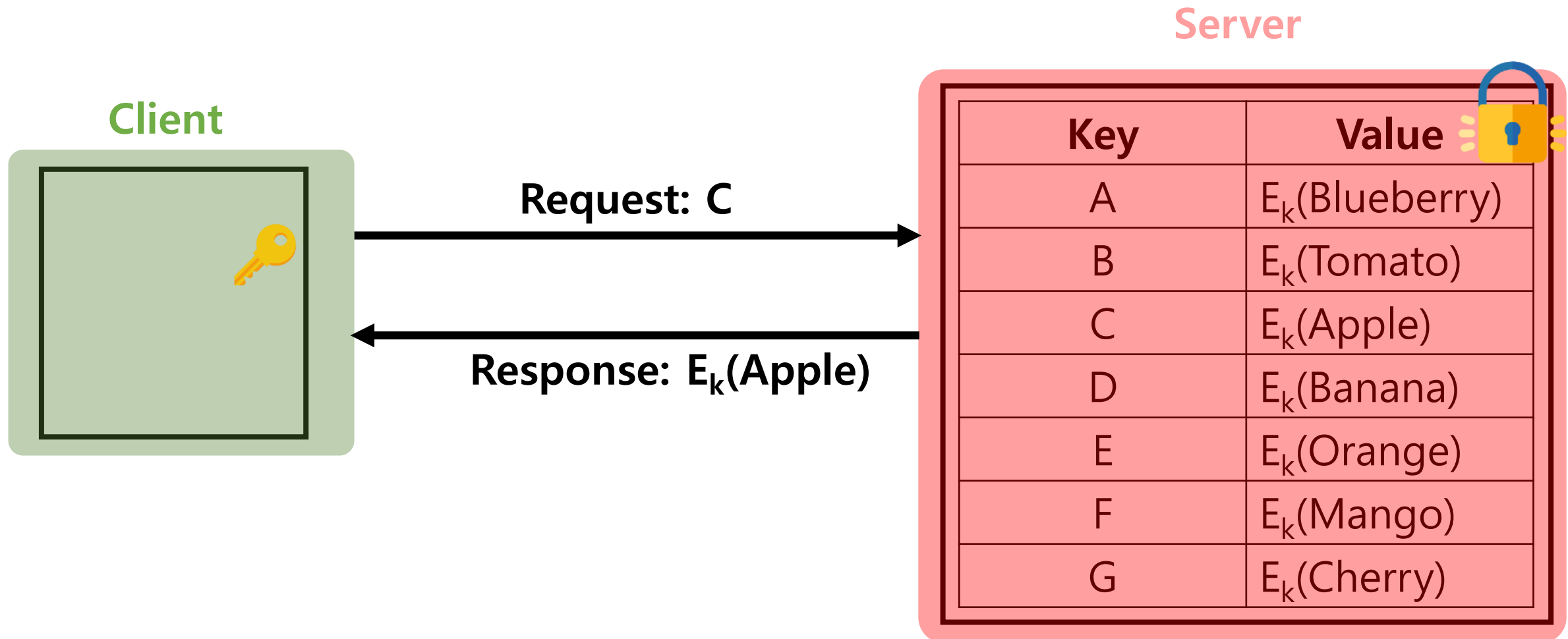


Key	Value
A	$E_k(\text{Blueberry})$
B	$E_k(\text{Tomato})$
C	$E_k(\text{Apple})$
D	$E_k(\text{Banana})$
E	$E_k(\text{Orange})$
F	$E_k(\text{Mango})$
G	$E_k(\text{Cherry})$

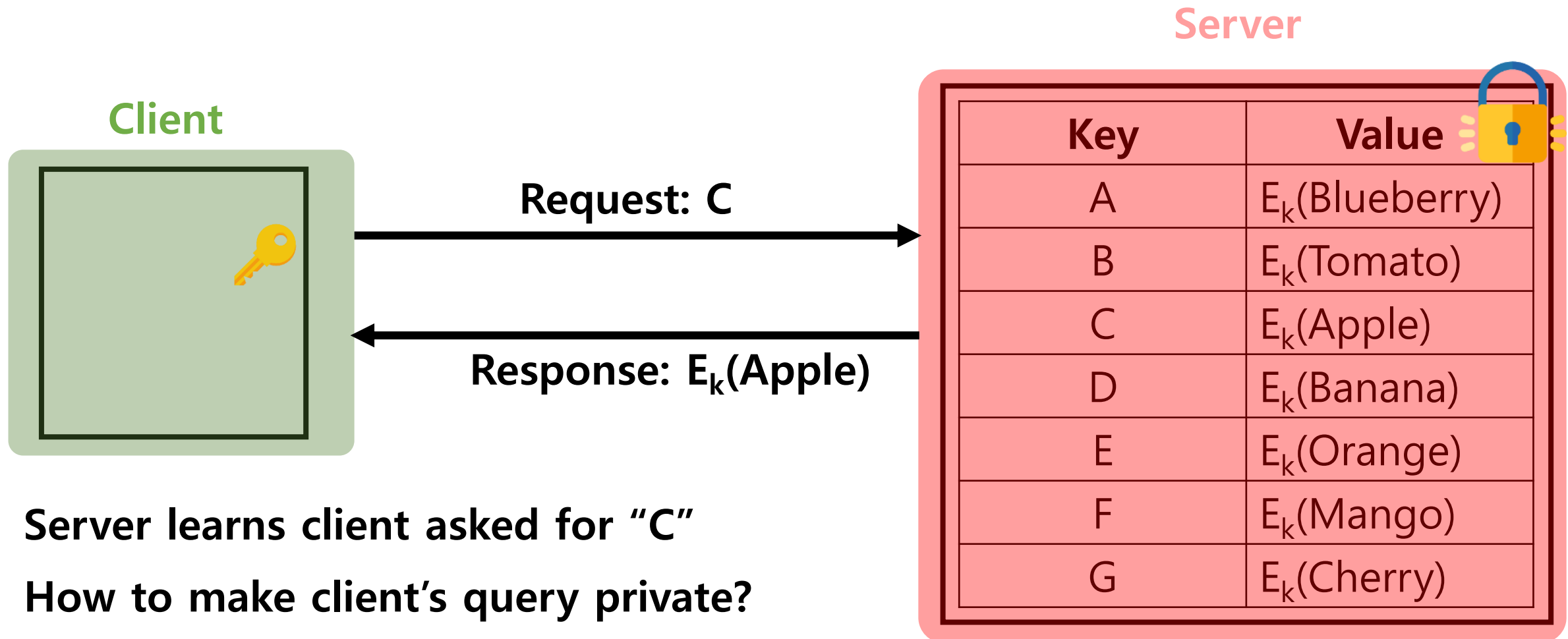
Why Does Access Patterns Matter?



Why Does Access Patterns Matter?

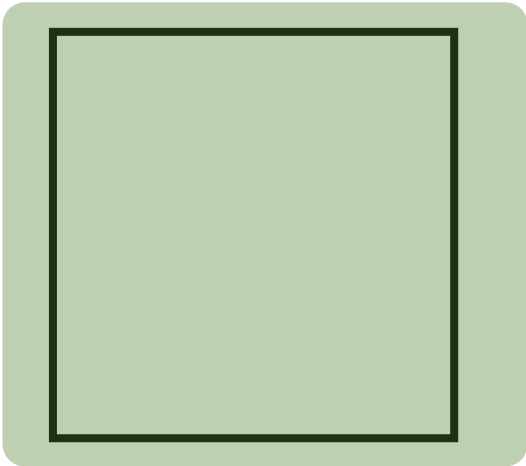


Why Does Access Patterns Matter?

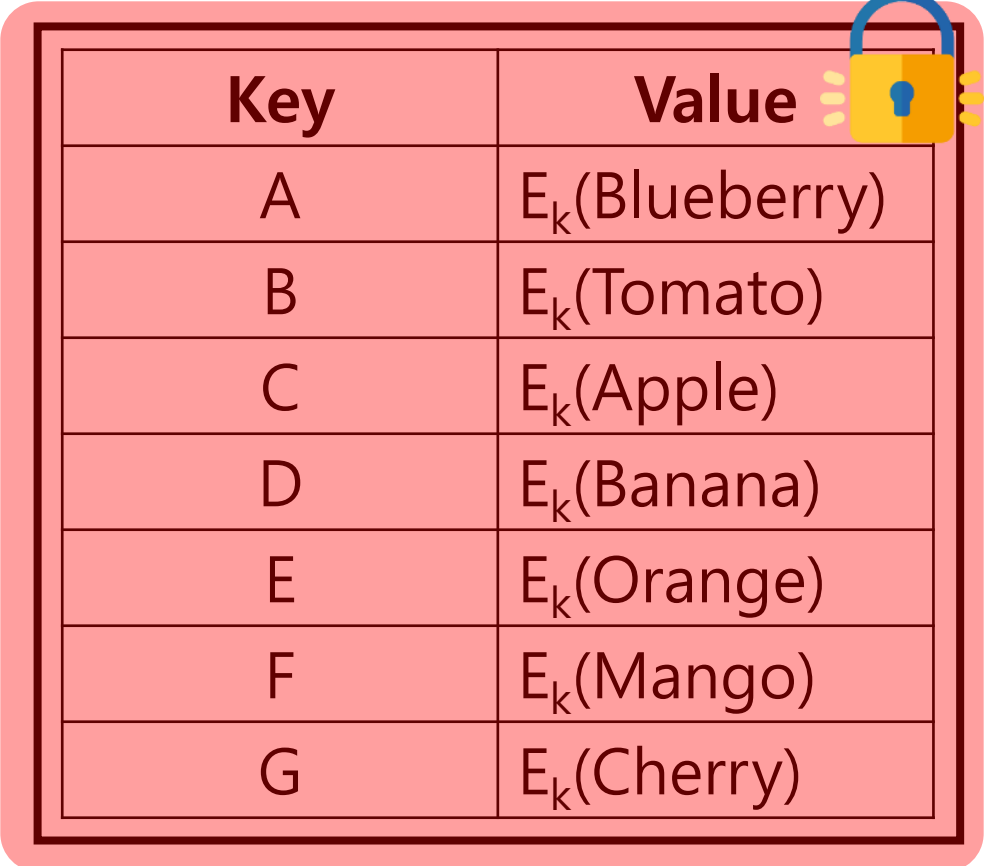


Easy Solution: Ask Everything

Client

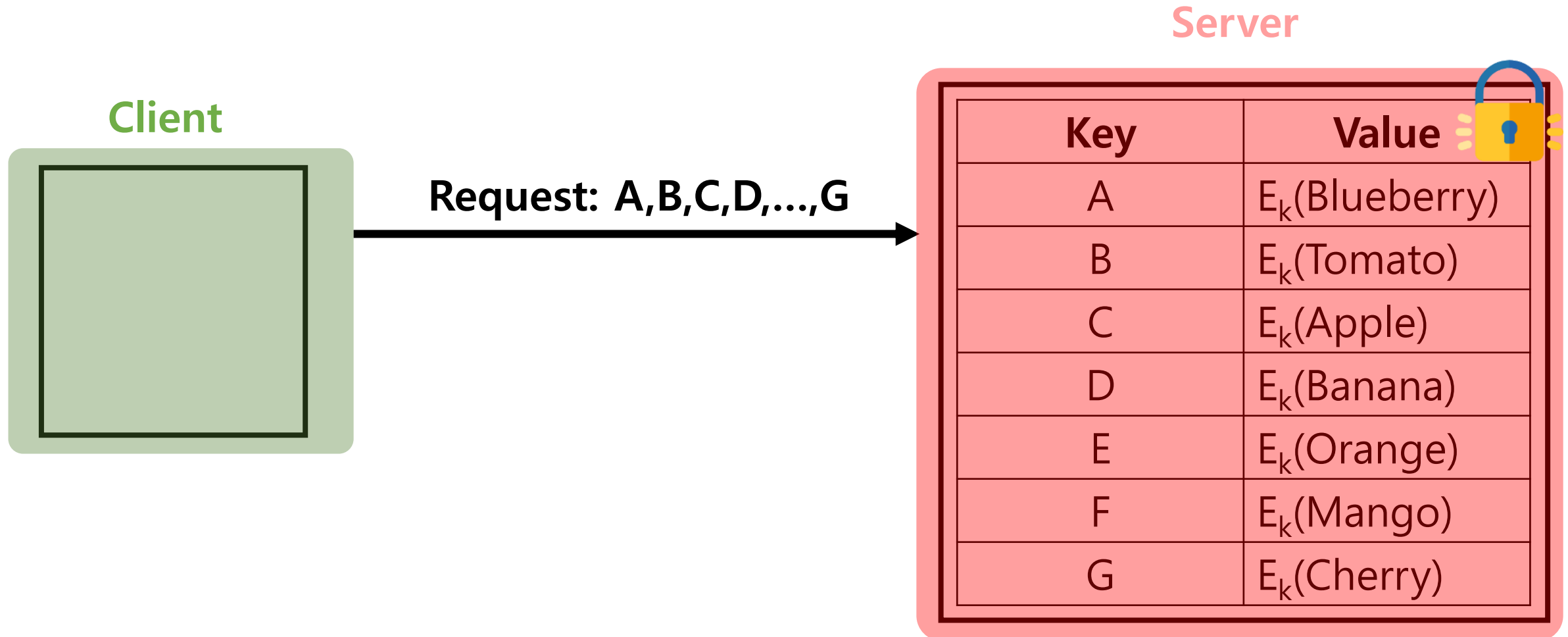


Server

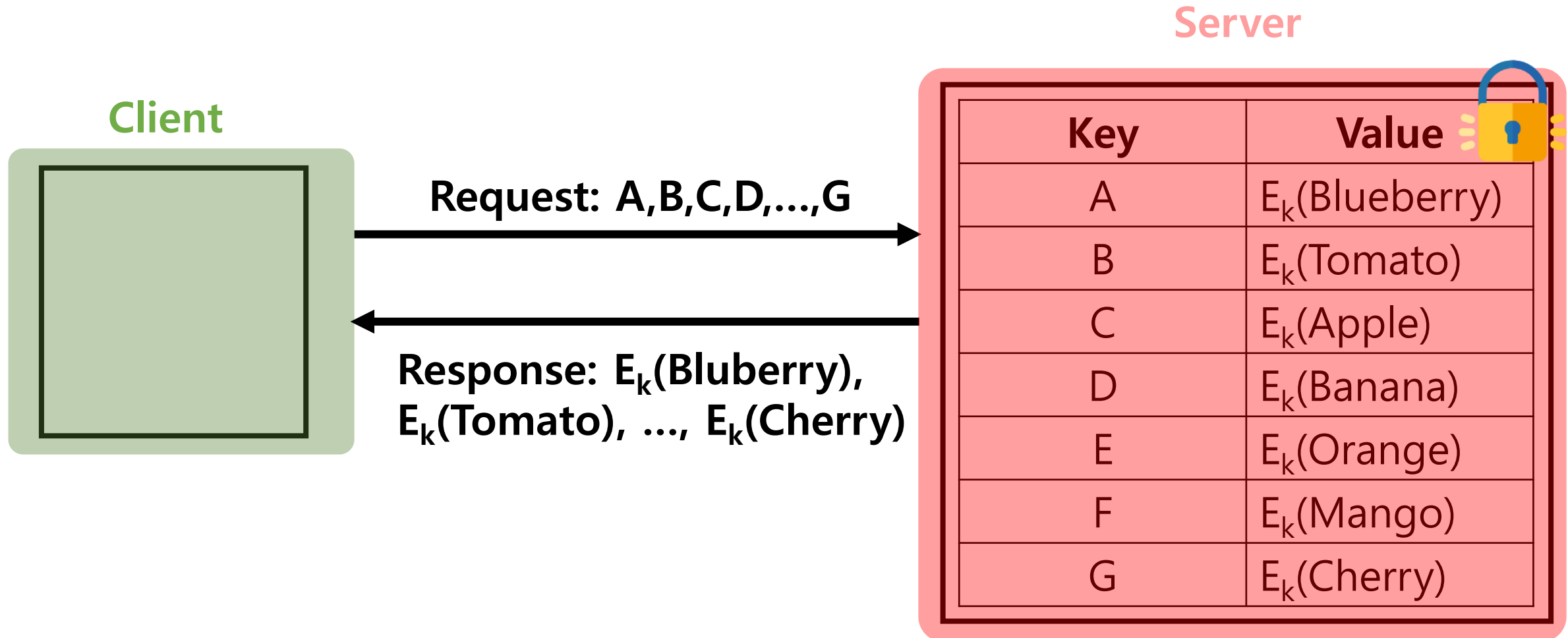


Key	Value
A	$E_k(\text{Blueberry})$
B	$E_k(\text{Tomato})$
C	$E_k(\text{Apple})$
D	$E_k(\text{Banana})$
E	$E_k(\text{Orange})$
F	$E_k(\text{Mango})$
G	$E_k(\text{Cherry})$

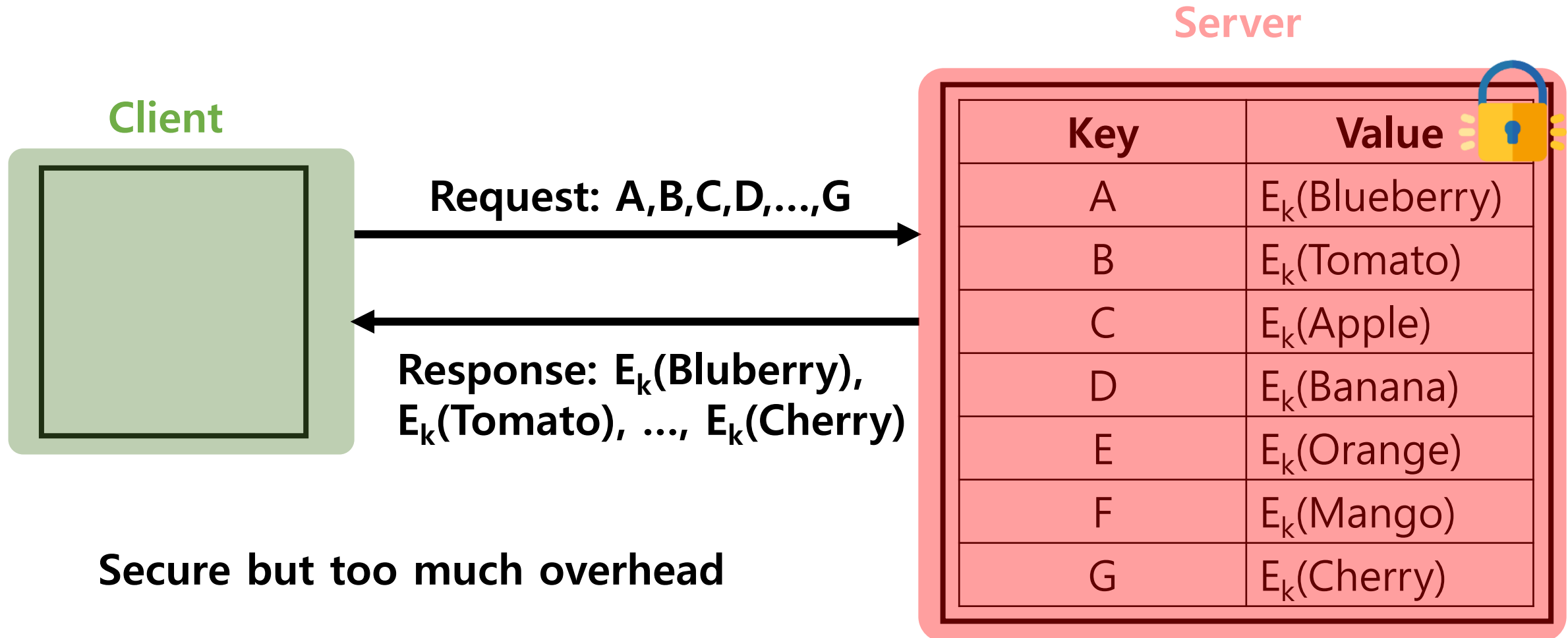
Easy Solution: Ask Everything



Easy Solution: Ask Everything

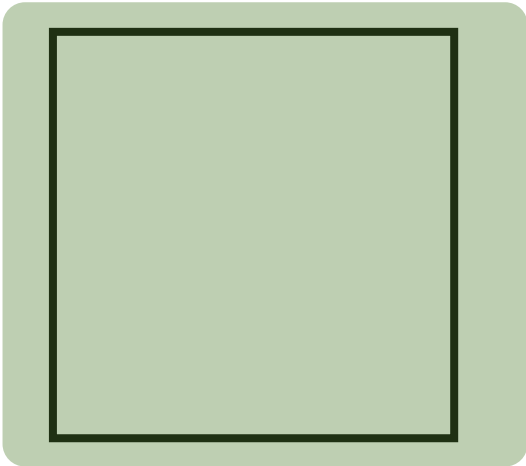


Easy Solution: Ask Everything

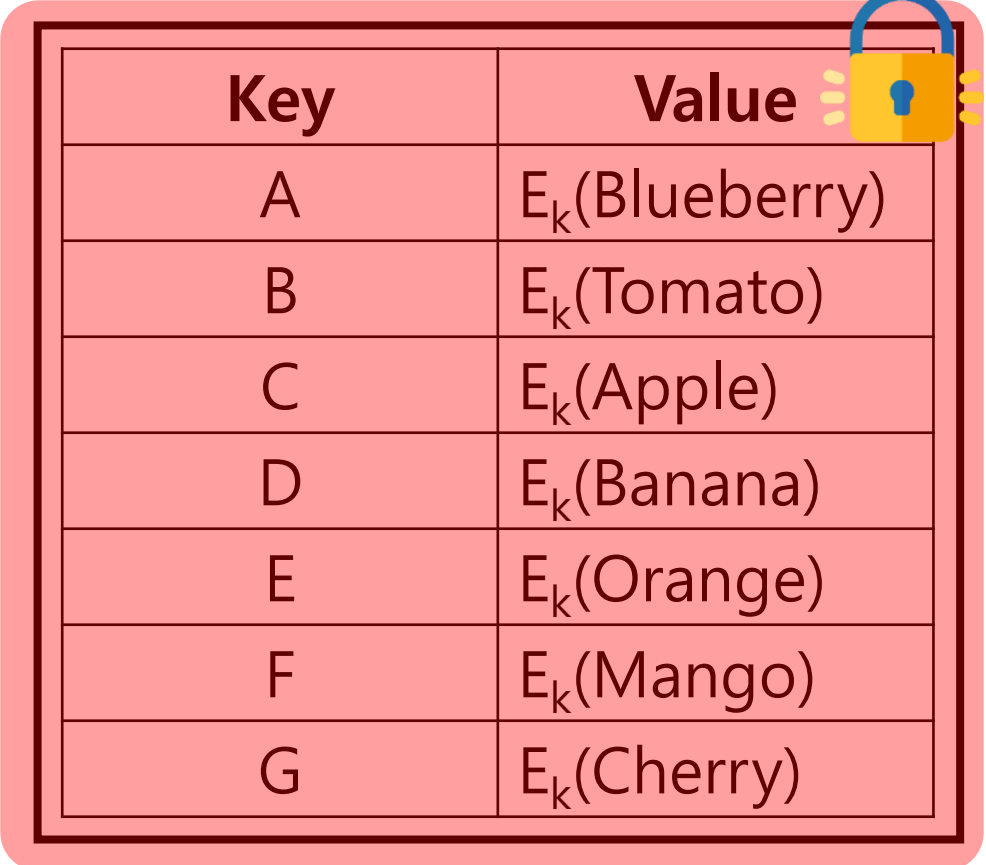


Better Solution: Ask k tuples [S&P 98]

Client

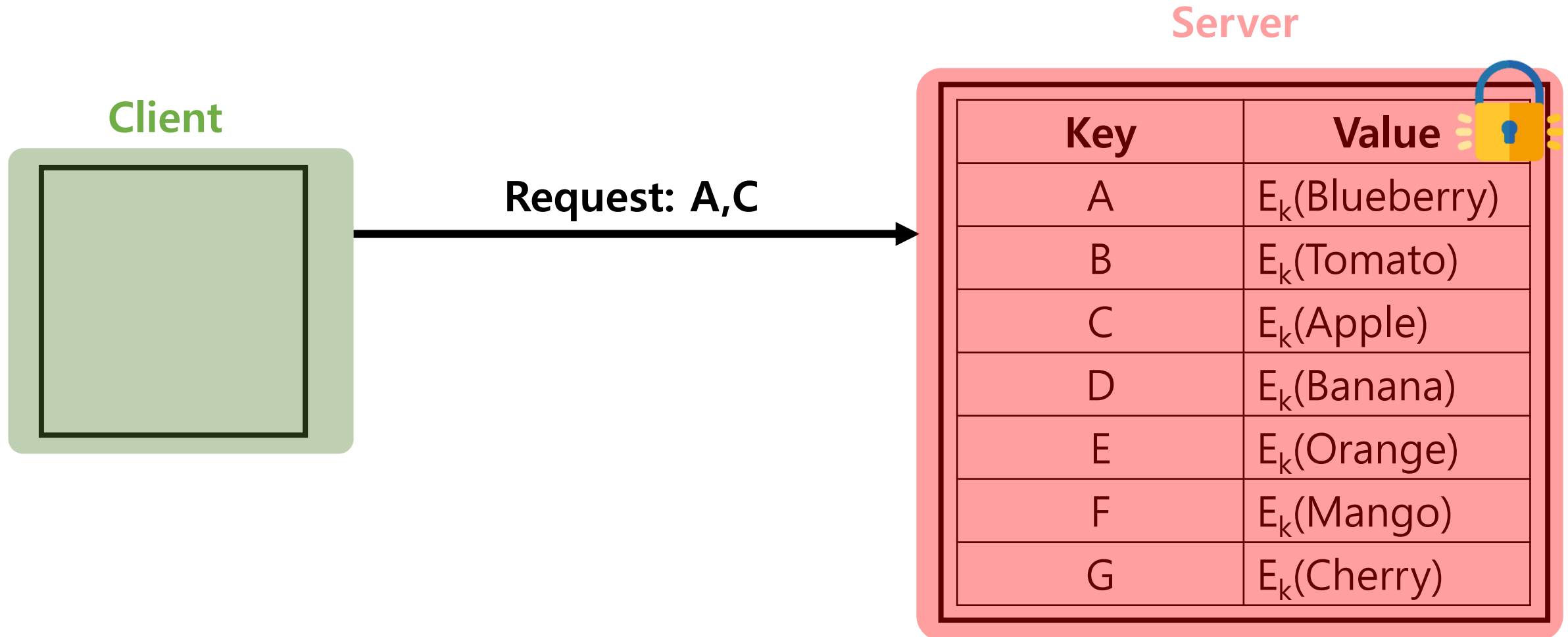


Server

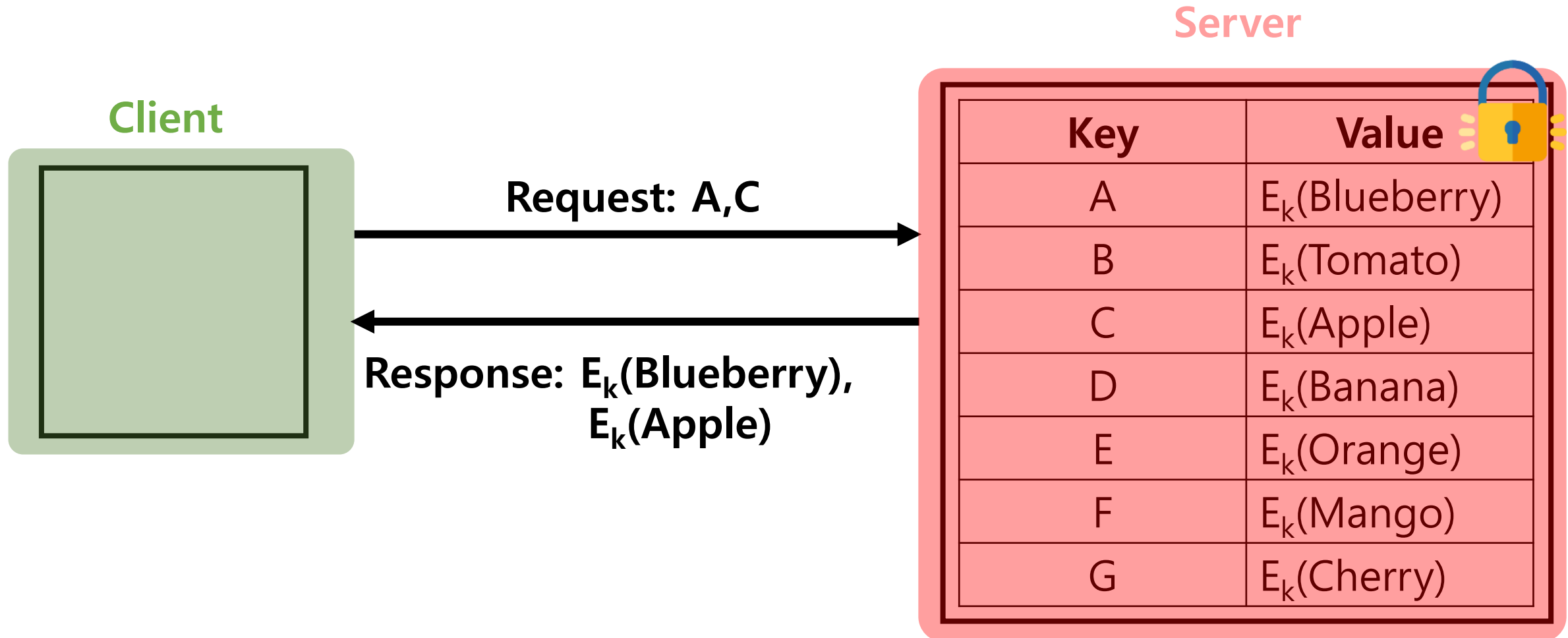


Key	Value
A	$E_k(\text{Blueberry})$
B	$E_k(\text{Tomato})$
C	$E_k(\text{Apple})$
D	$E_k(\text{Banana})$
E	$E_k(\text{Orange})$
F	$E_k(\text{Mango})$
G	$E_k(\text{Cherry})$

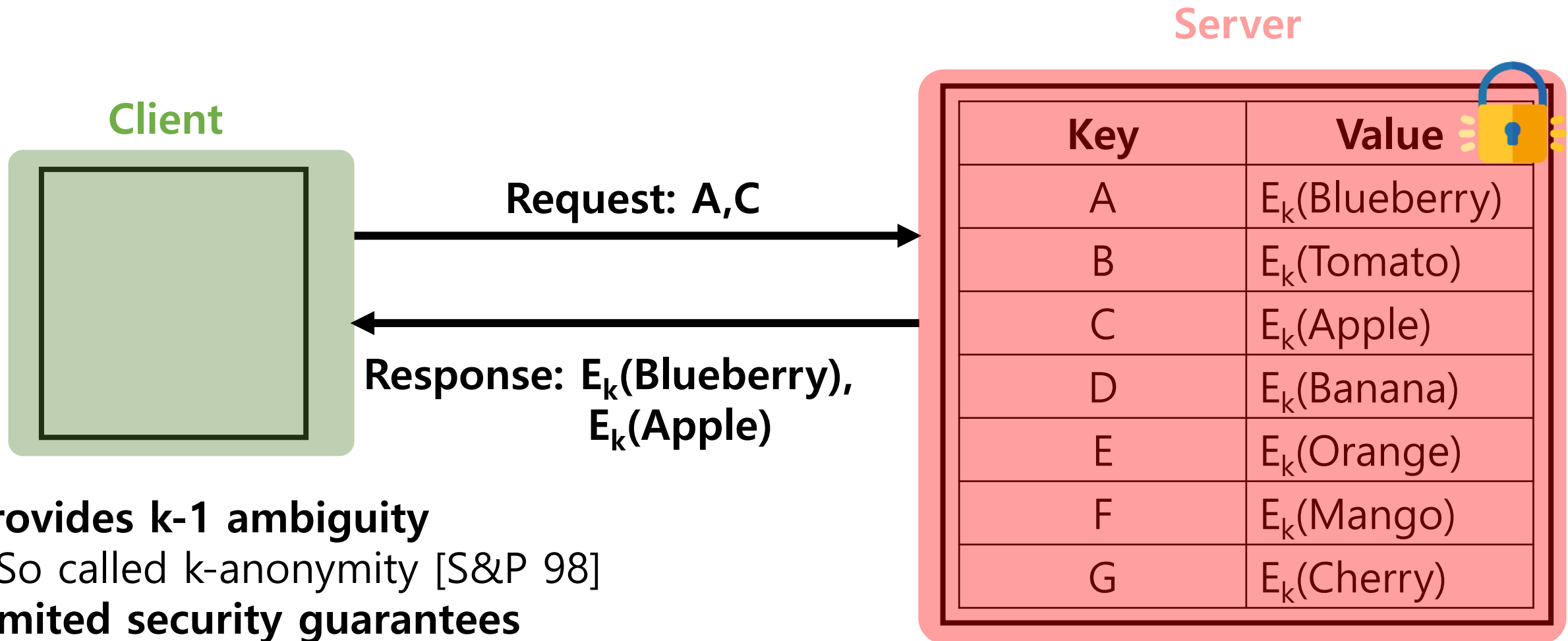
Better Solution: Ask k tuples [S&P 98]



Better Solution: Ask k tuples [S&P 98]



Better Solution: Ask k tuples [S&P 98]



Provides k-1 ambiguity

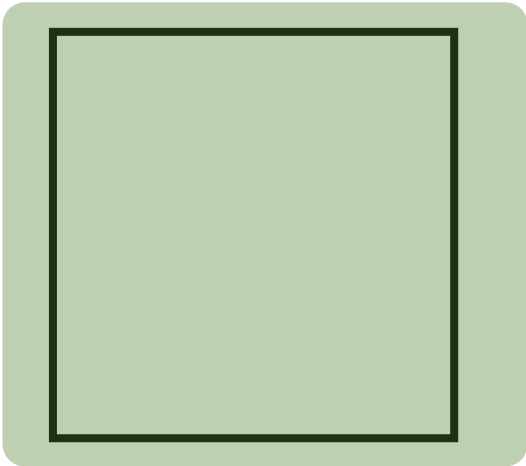
- So called k-anonymity [S&P 98]

Limited security guarantees


- See l-diversity [ICDE 06], t-closeness [ICDE 07]

Oblivious RAM (ORAM): Idea Sketch

Client

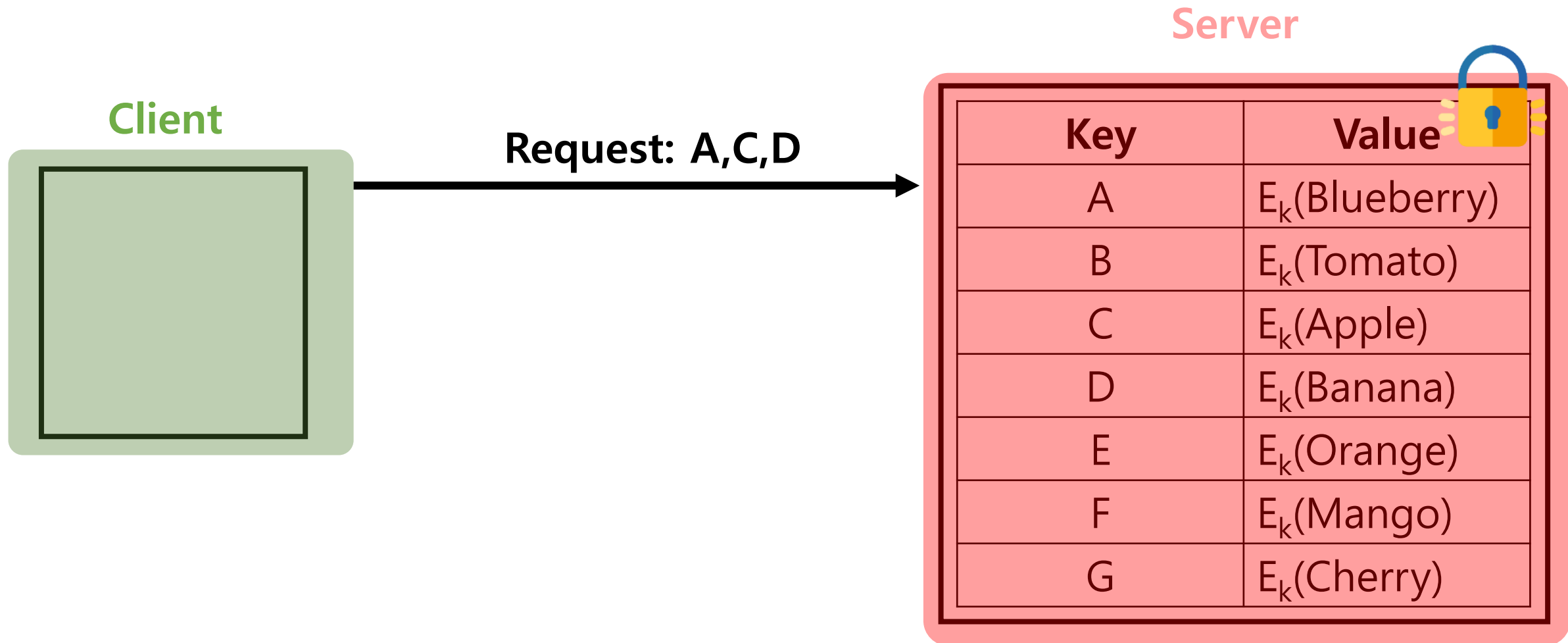


Server

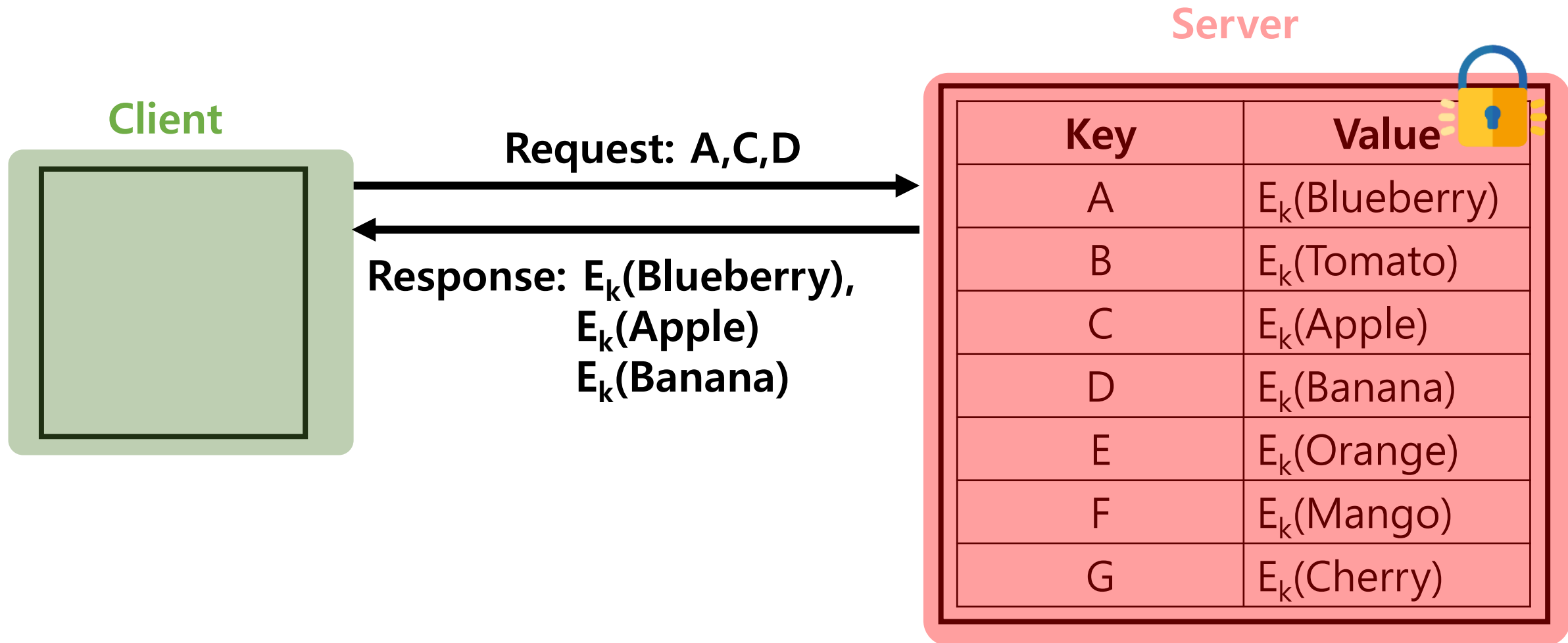


Key	Value
A	$E_k(\text{Blueberry})$
B	$E_k(\text{Tomato})$
C	$E_k(\text{Apple})$
D	$E_k(\text{Banana})$
E	$E_k(\text{Orange})$
F	$E_k(\text{Mango})$
G	$E_k(\text{Cherry})$

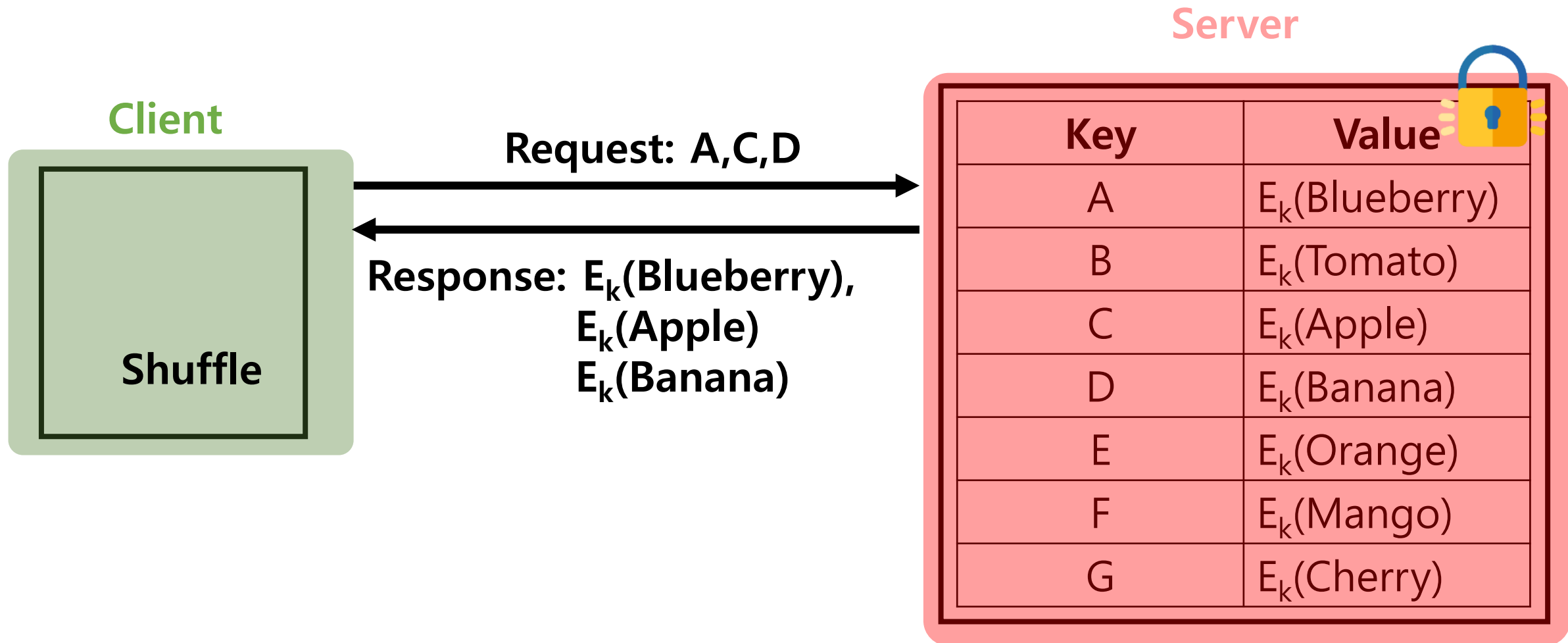
Oblivious RAM (ORAM): Idea Sketch



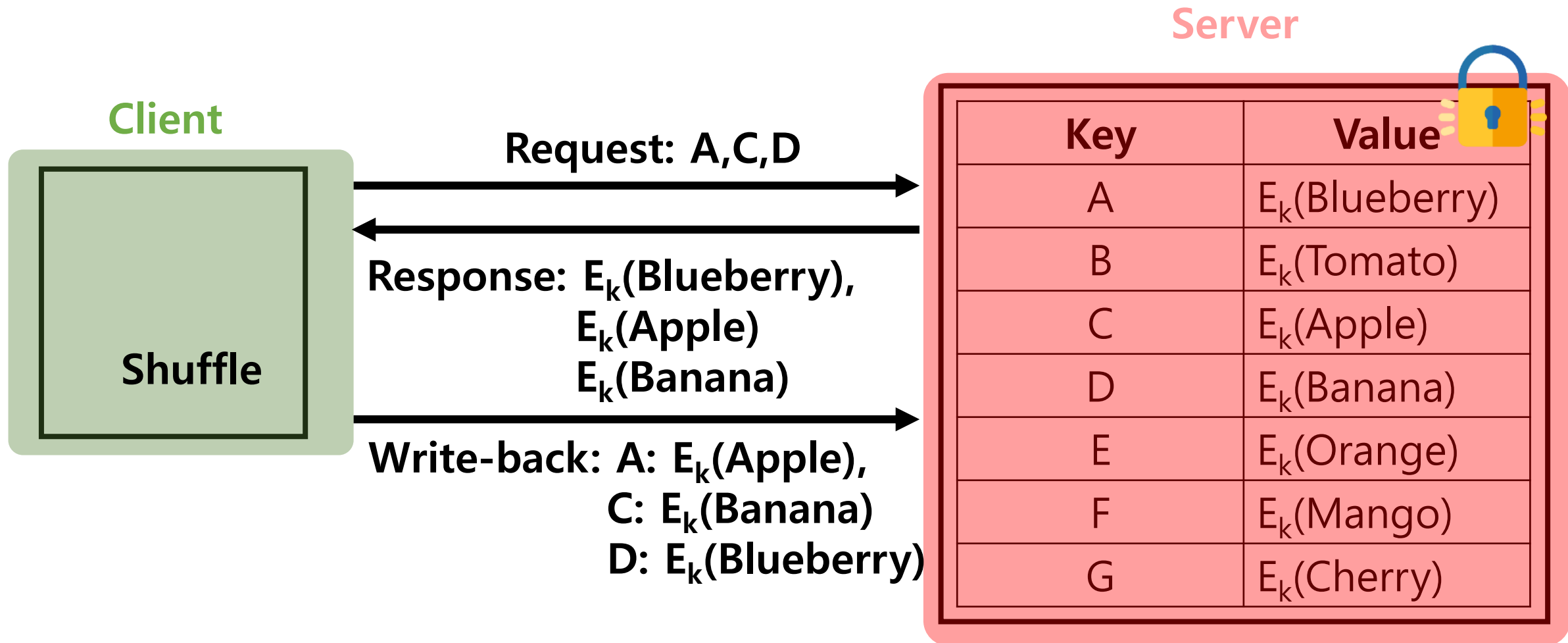
Oblivious RAM (ORAM): Idea Sketch



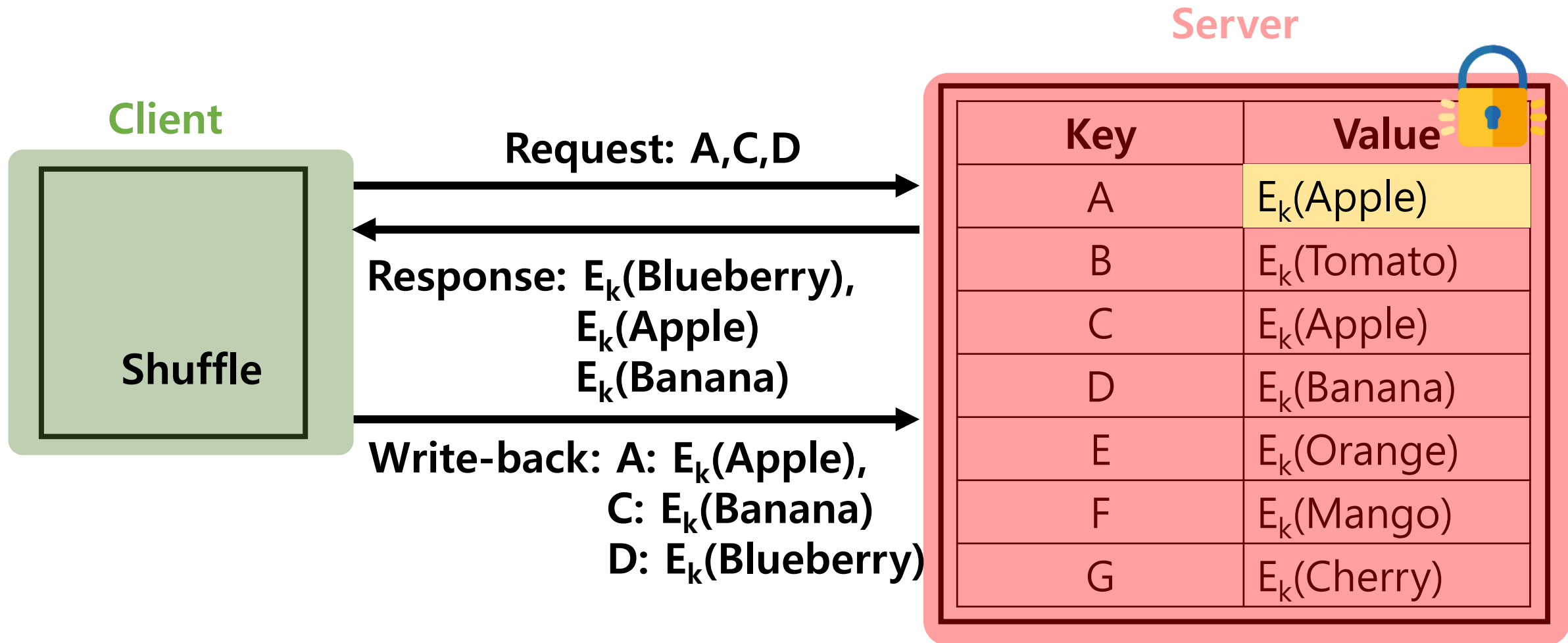
Oblivious RAM (ORAM): Idea Sketch



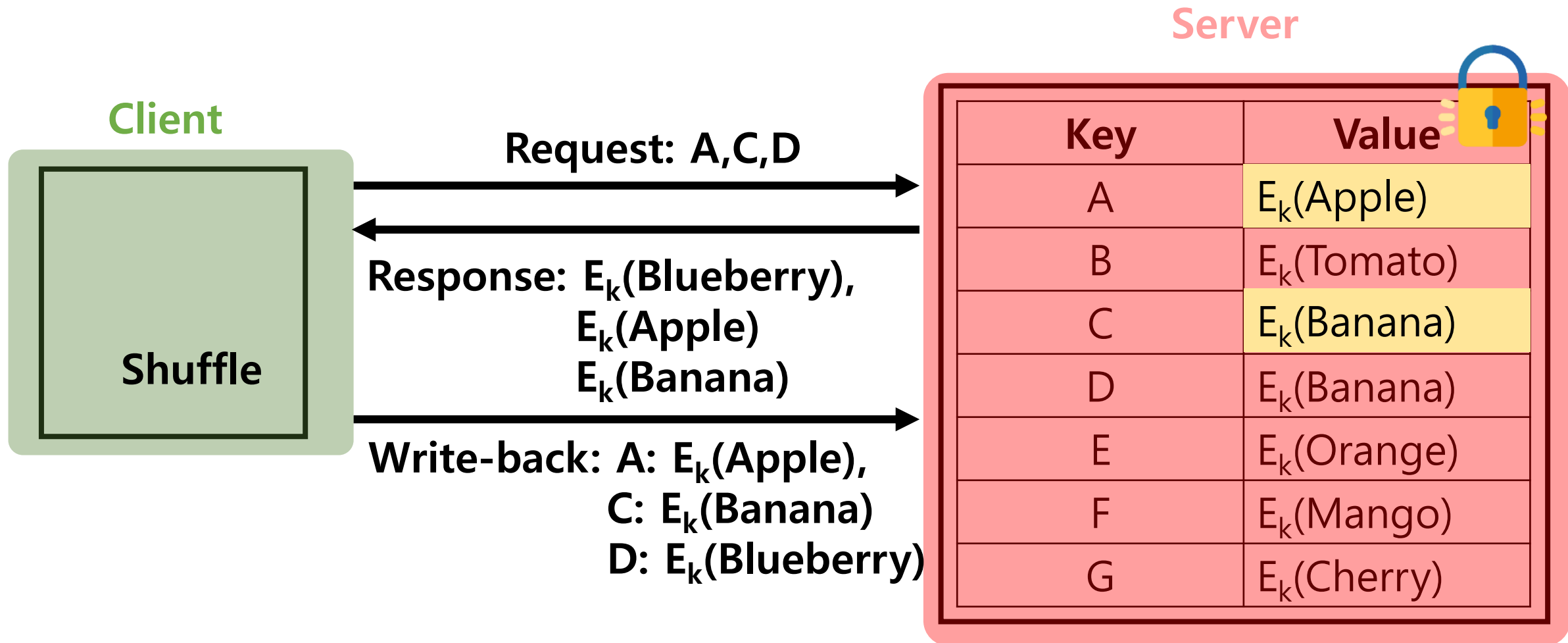
Oblivious RAM (ORAM): Idea Sketch



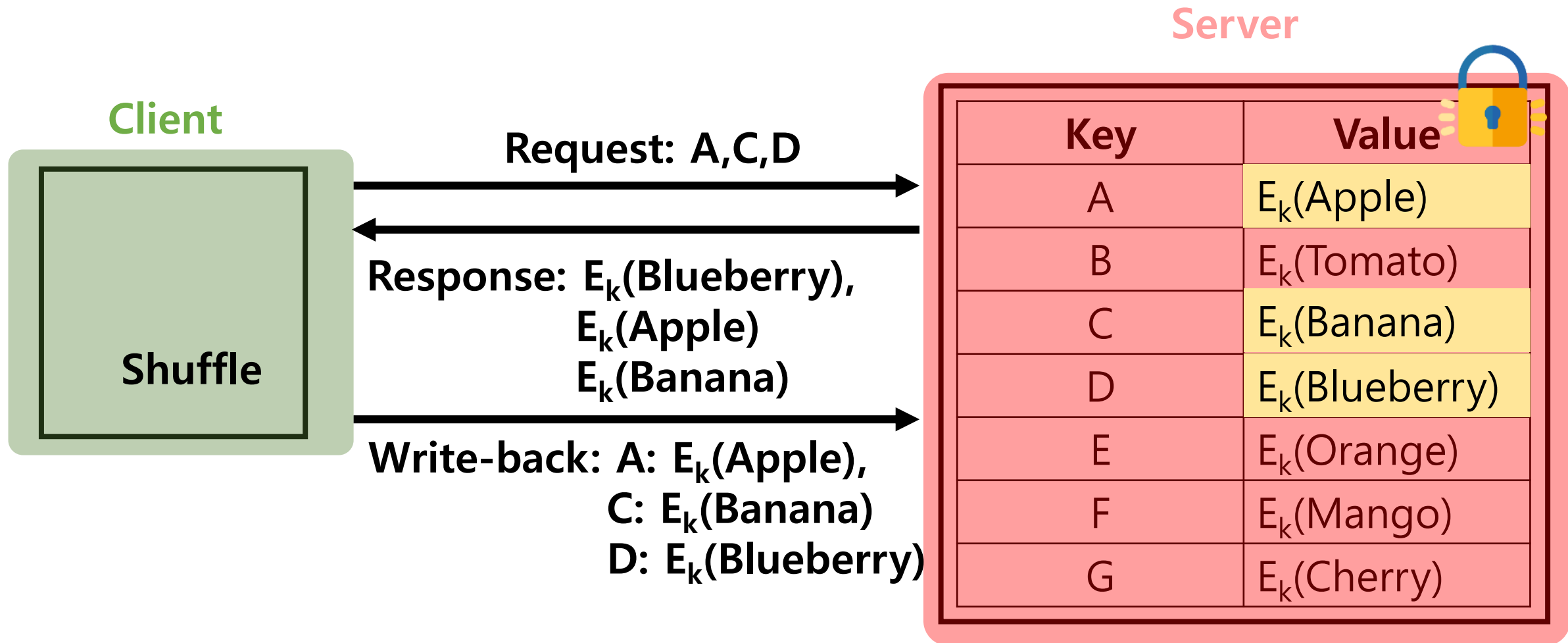
Oblivious RAM (ORAM): Idea Sketch



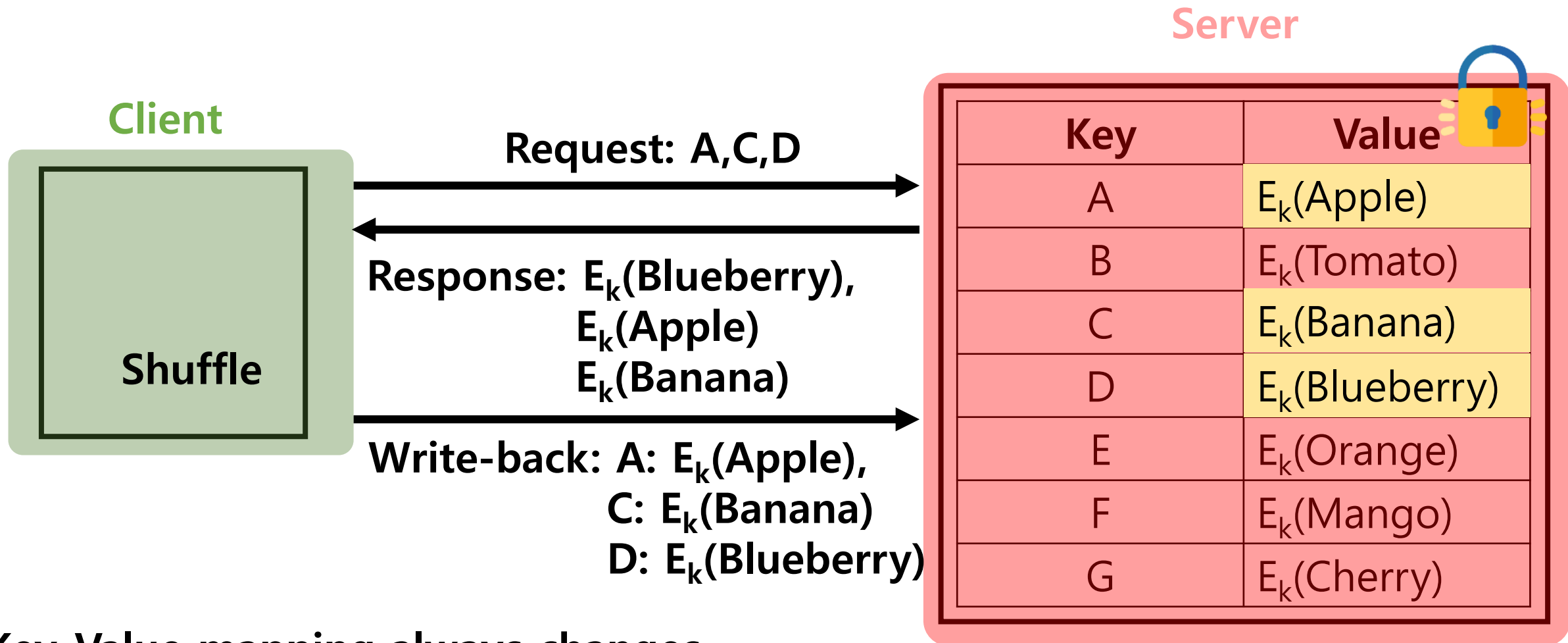
Oblivious RAM (ORAM): Idea Sketch



Oblivious RAM (ORAM): Idea Sketch



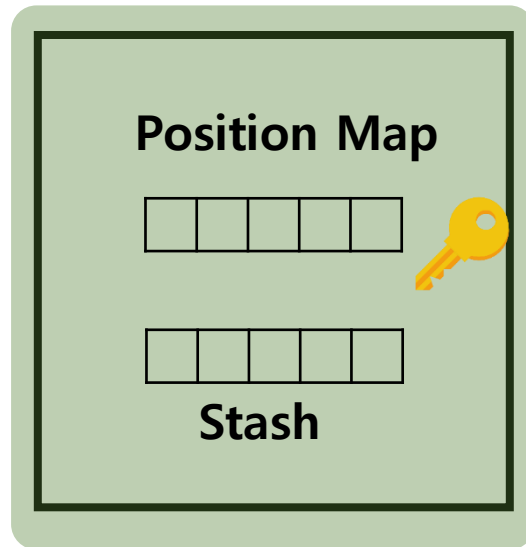
Oblivious RAM (ORAM): Idea Sketch



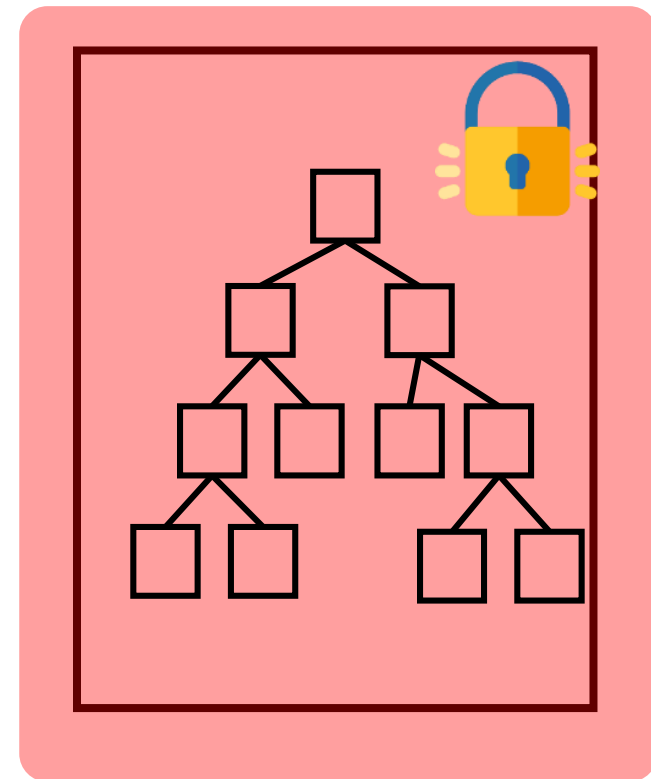
Key-Value mapping always changes

Path ORAM [CCS 13]

ORAM Client

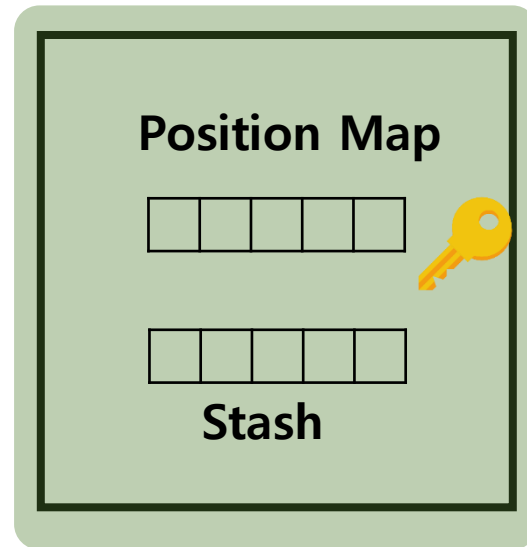


ORAM Server

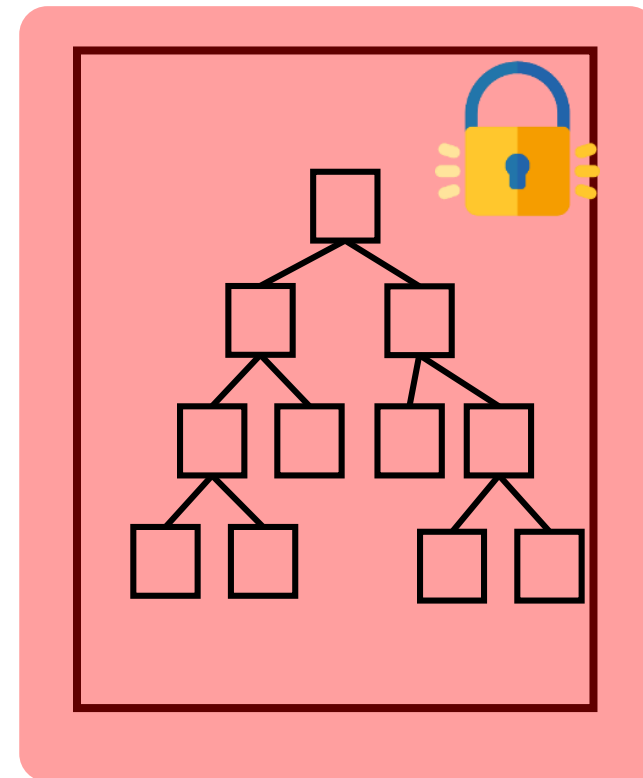


Path ORAM [CCS 13]

ORAM Client



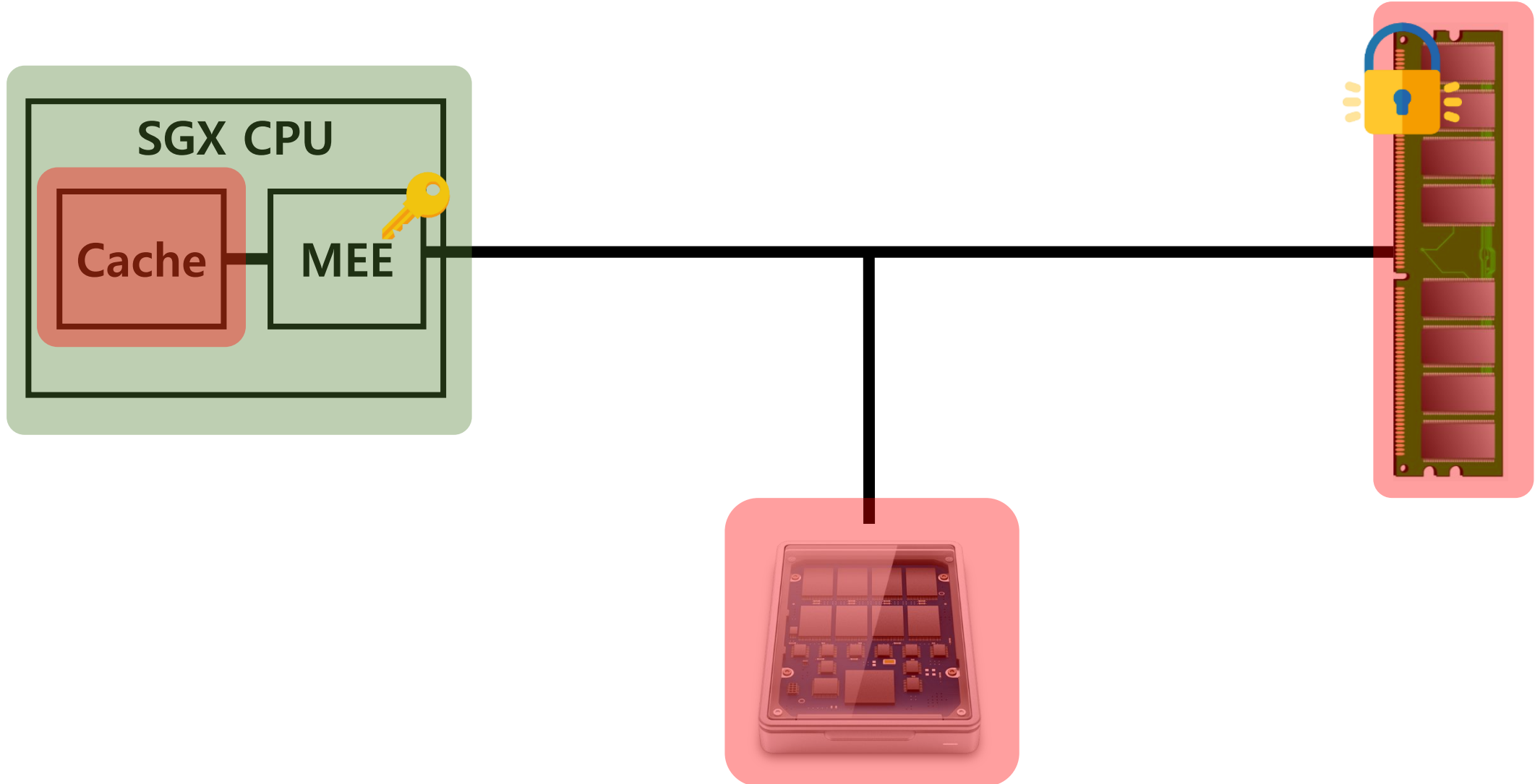
ORAM Server



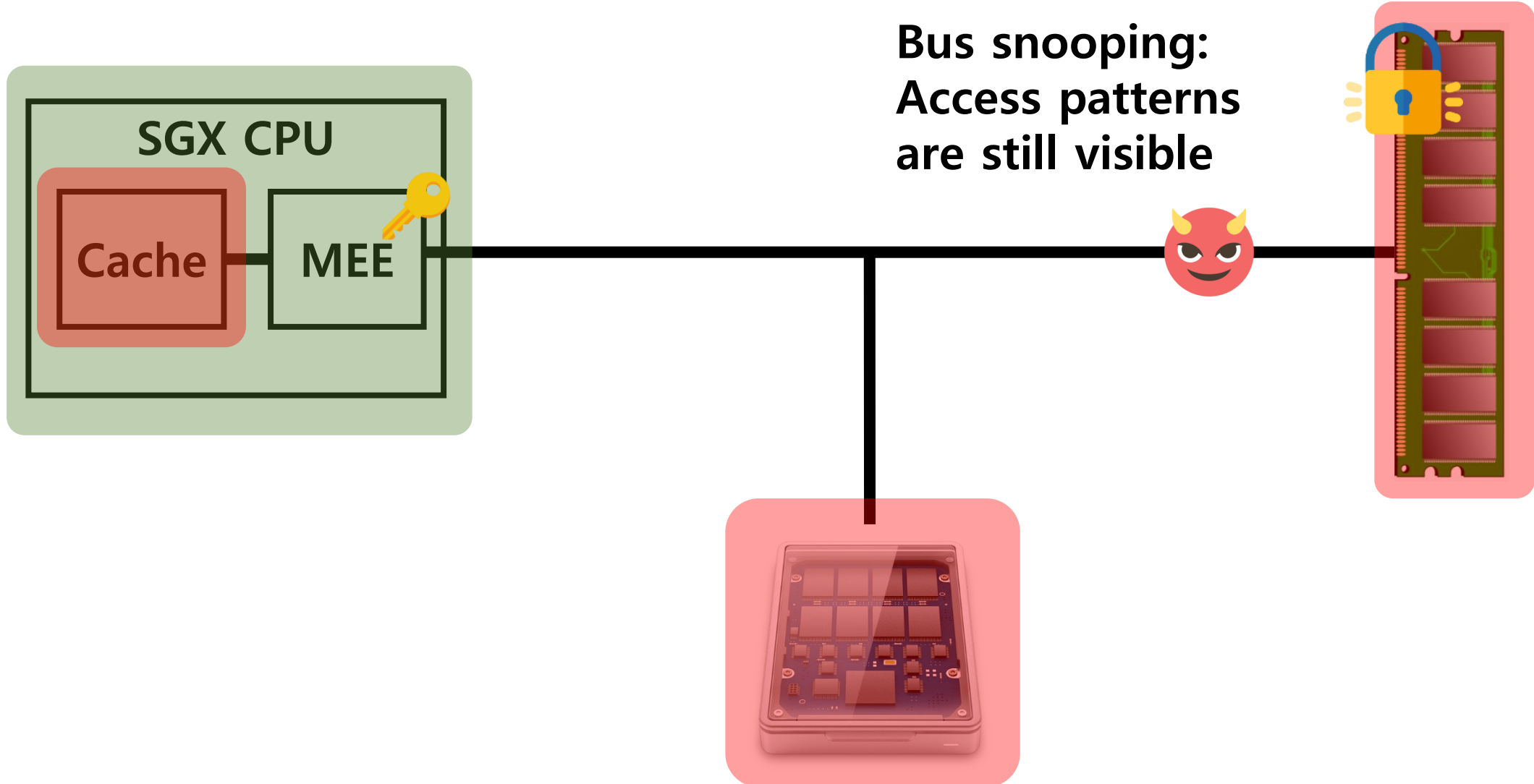
Tree-like data structures

- Client: Position map, stash
- Server: ORAM Tree with real/dummy nodes

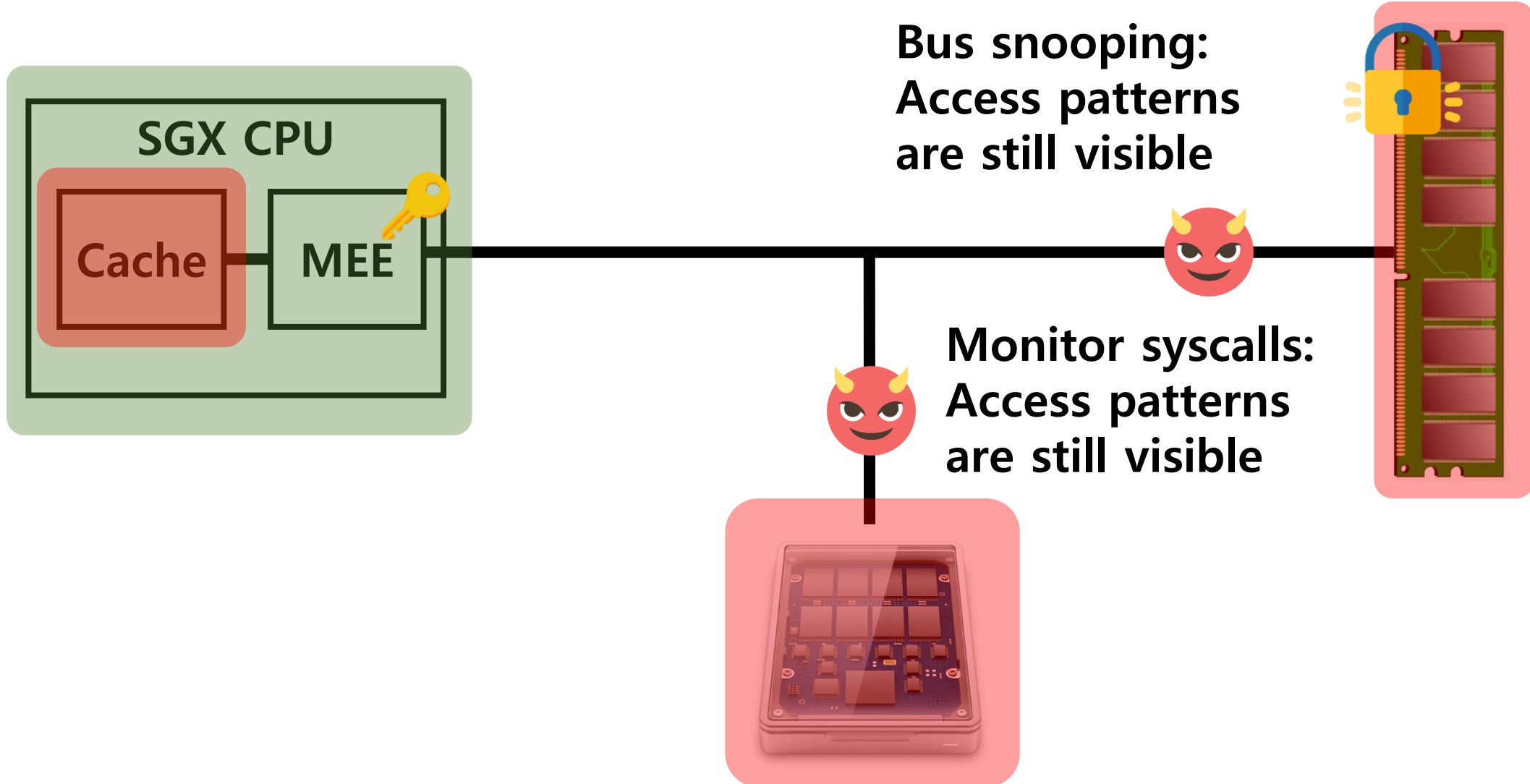
ORAM-based solutions for Memory Access



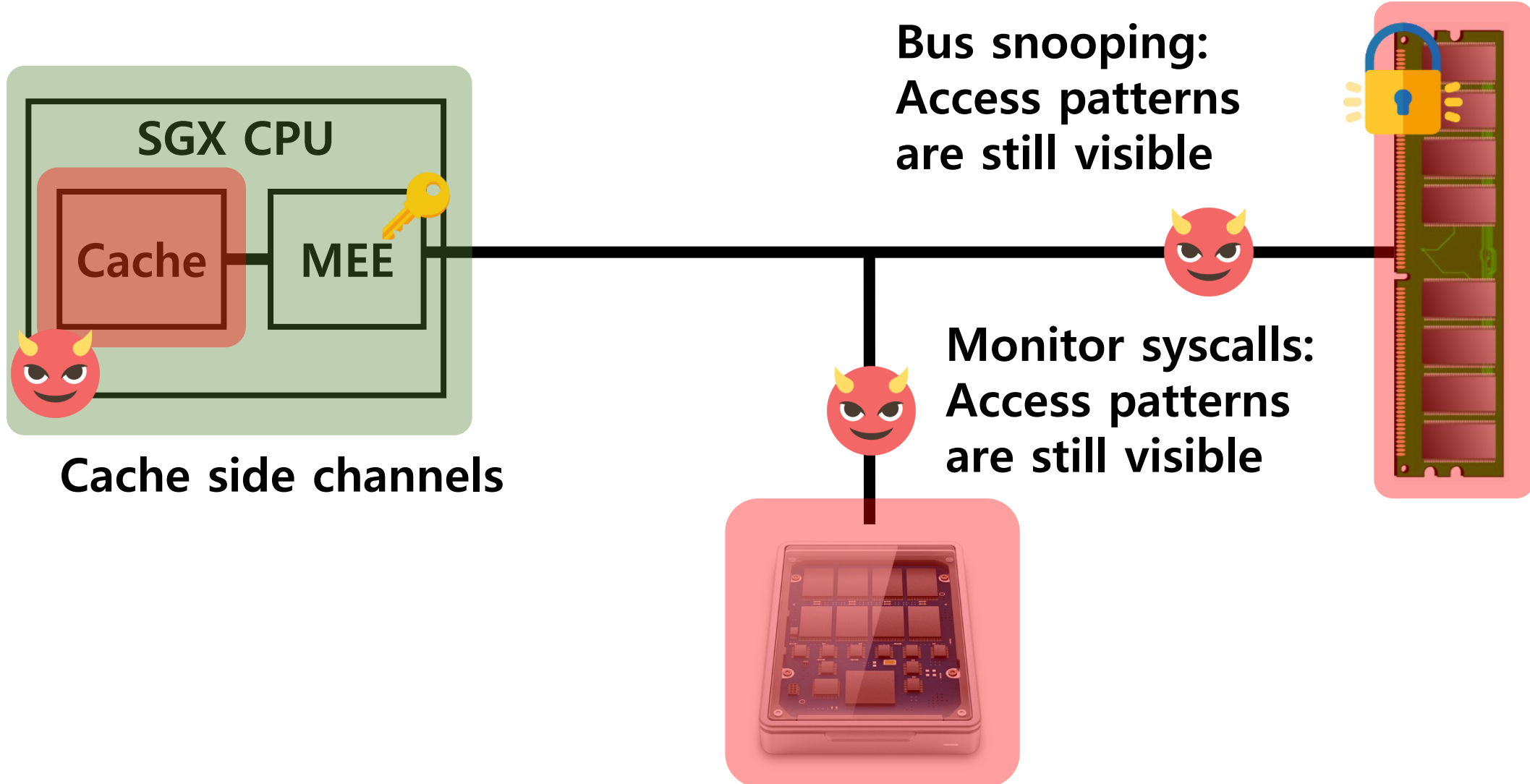
ORAM-based solutions for Memory Access



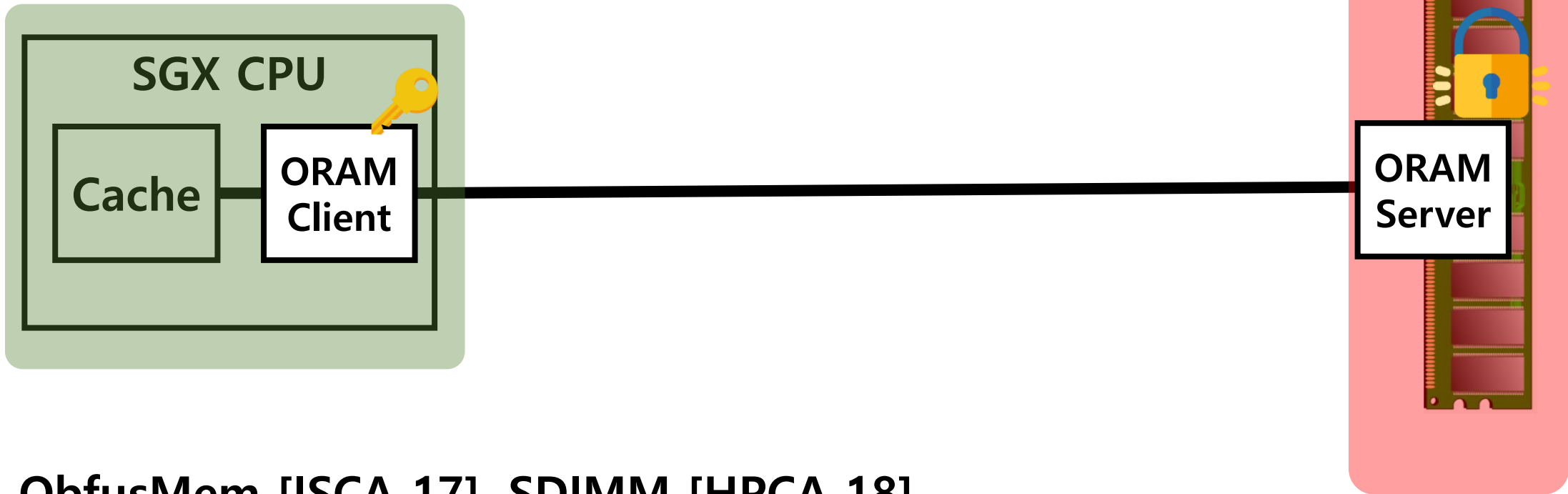
ORAM-based solutions for Memory Access



ORAM-based solutions for Memory Access

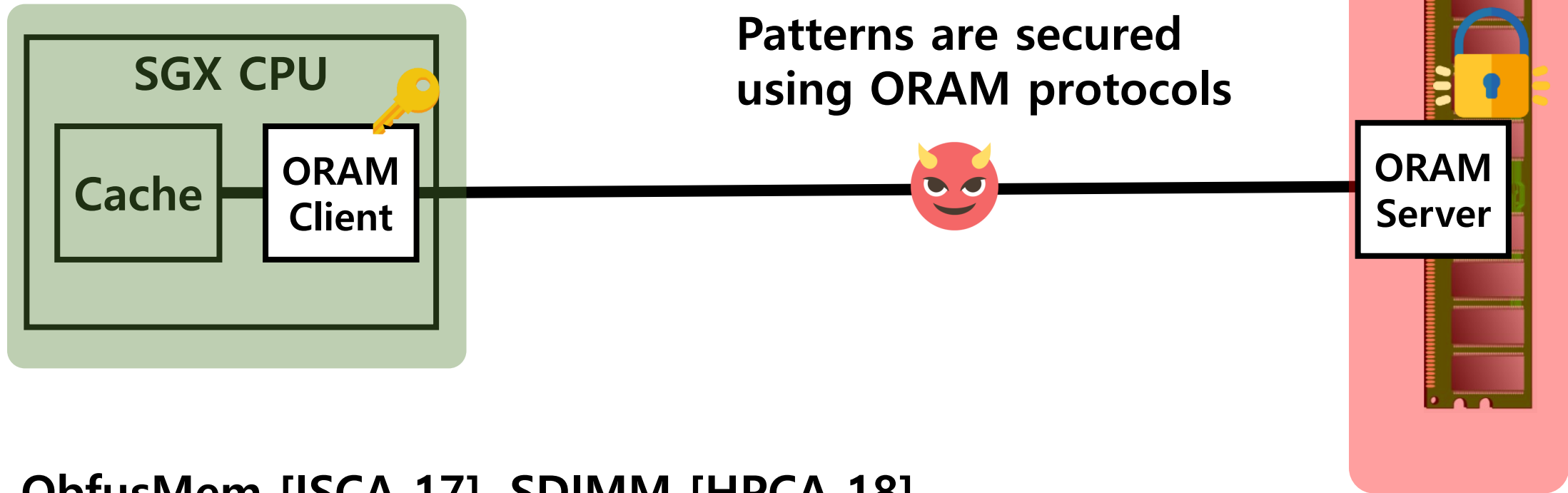


Mitigation: ORAM-based Memory Controller



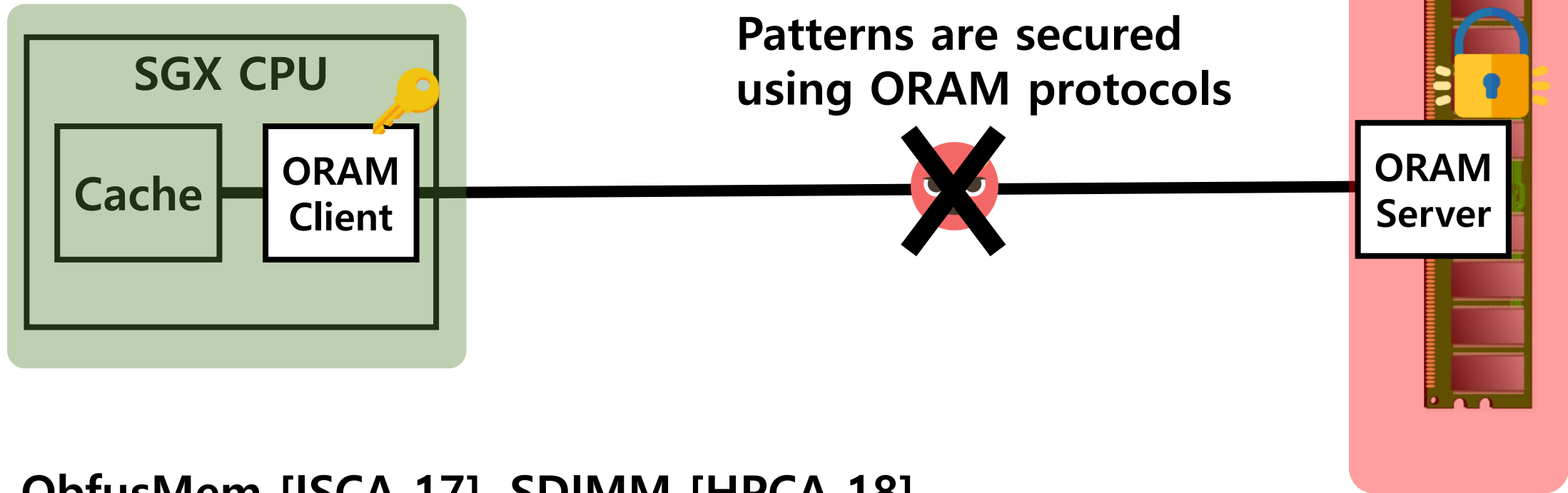
ObfusMem [ISCA 17], SDIMM [HPCA 18]
- ORAM-based Memory Controller

Mitigation: ORAM-based Memory Controller



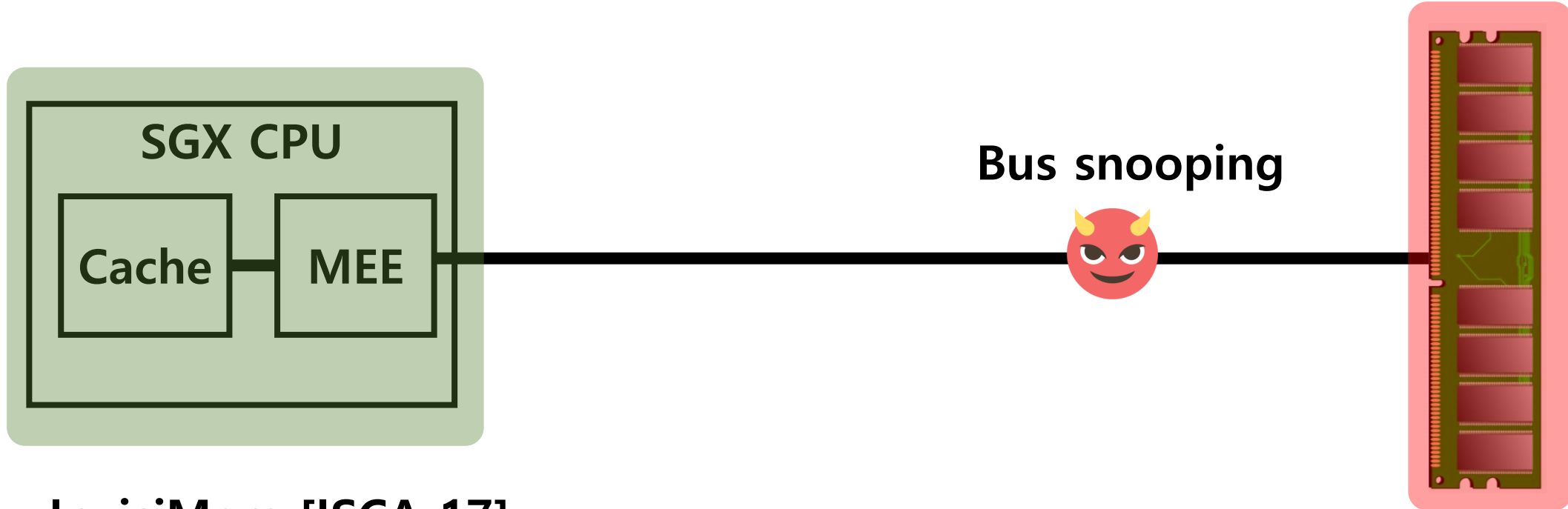
ObfusMem [ISCA 17], SDIMM [HPCA 18]
- ORAM-based Memory Controller

Mitigation: ORAM-based Memory Controller



ObfusMem [ISCA 17], SDIMM [HPCA 18]
- ORAM-based Memory Controller

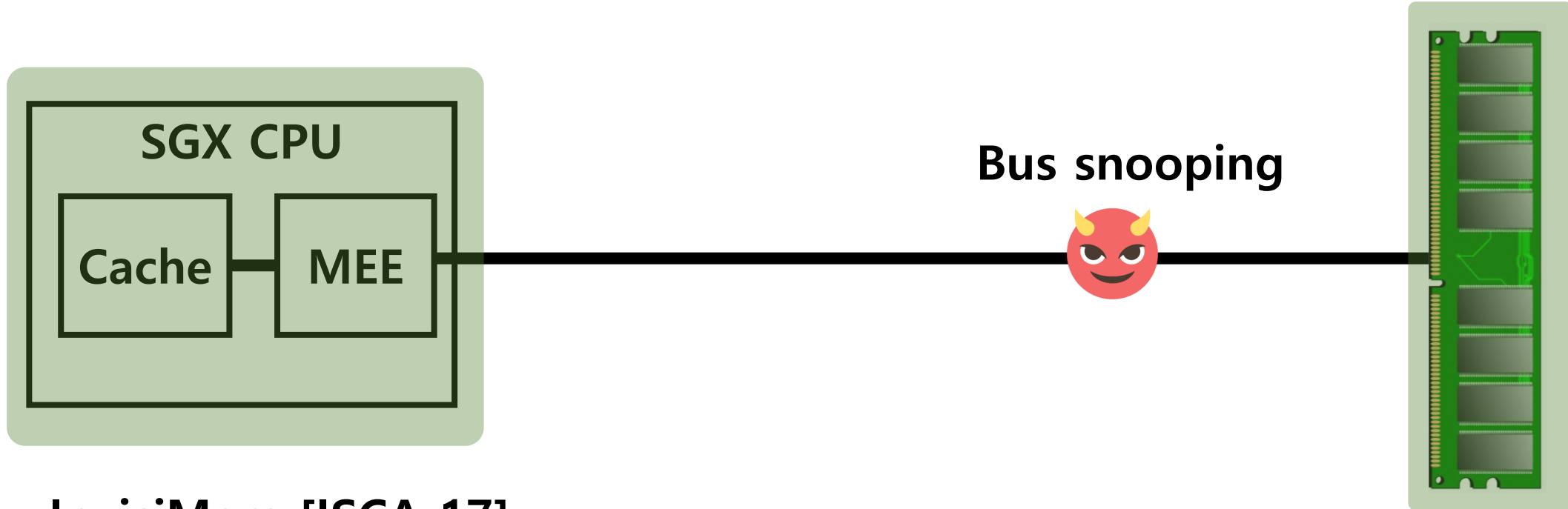
Mitigation: Place Trust in DRAM



InvisiMem [ISCA 17]

- Place trust in DRAM
- All address and data bus traffics are encrypted
 - ➔ Note: SGX only encrypts values in data bus
- Communication patterns are normalized

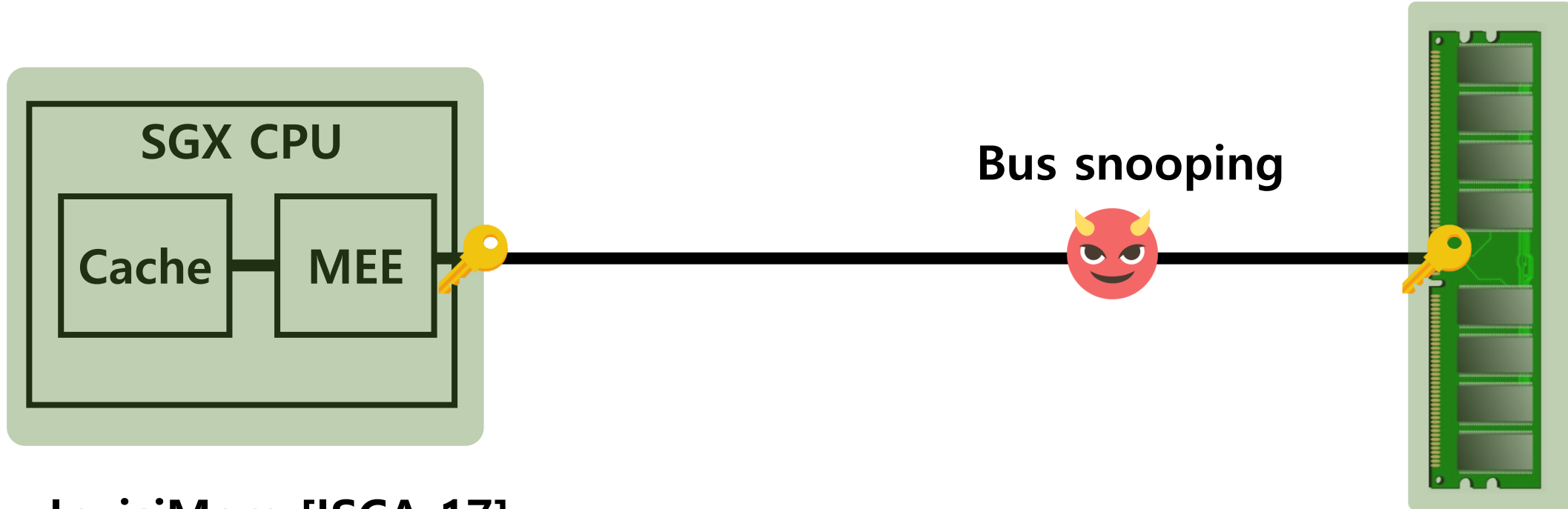
Mitigation: Place Trust in DRAM



InvisiMem [ISCA 17]

- Place trust in DRAM
- All address and data bus traffics are encrypted
 - ➔ Note: SGX only encrypts values in data bus
- Communication patterns are normalized

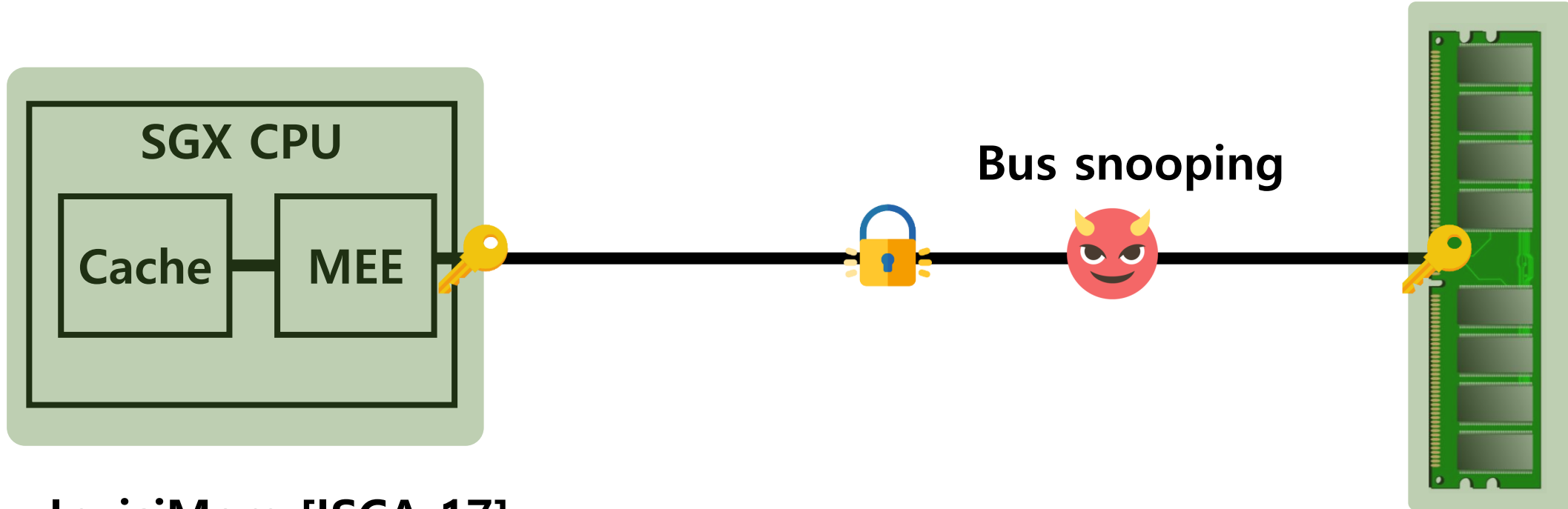
Mitigation: Place Trust in DRAM



InvisiMem [ISCA 17]

- Place trust in DRAM
- All address and data bus traffics are encrypted
 - ➔ Note: SGX only encrypts values in data bus
- Communication patterns are normalized

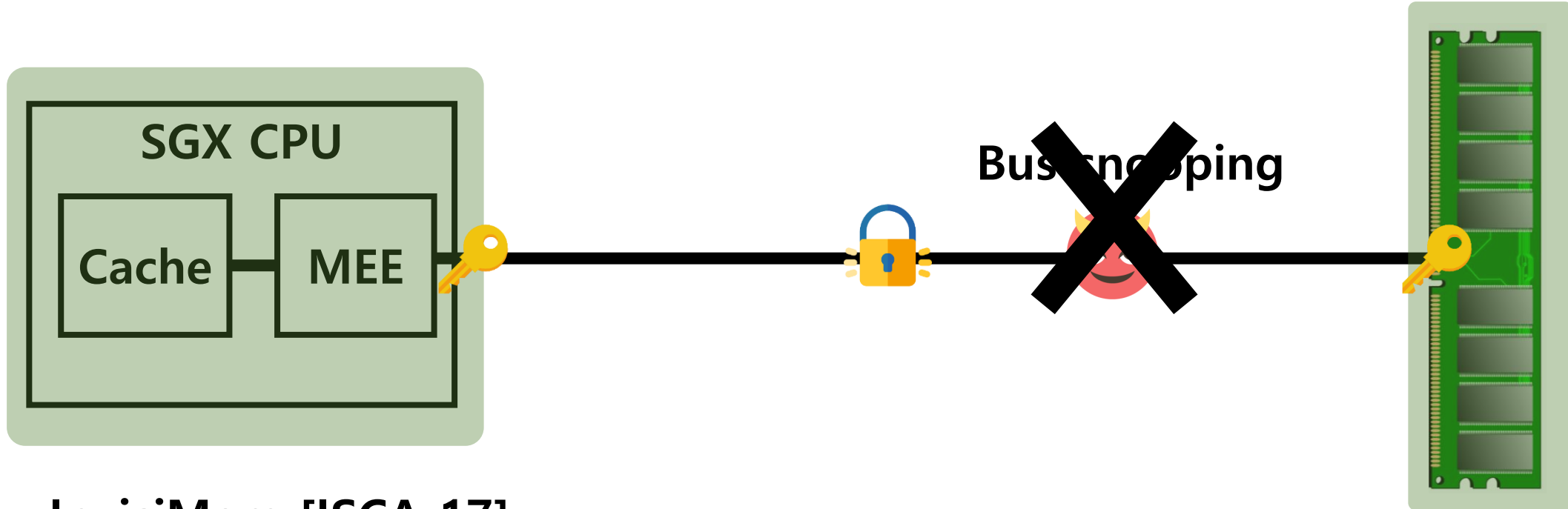
Mitigation: Place Trust in DRAM



InvisiMem [ISCA 17]

- Place trust in DRAM
- All address and data bus traffics are encrypted
 - ➔ Note: SGX only encrypts values in data bus
- Communication patterns are normalized

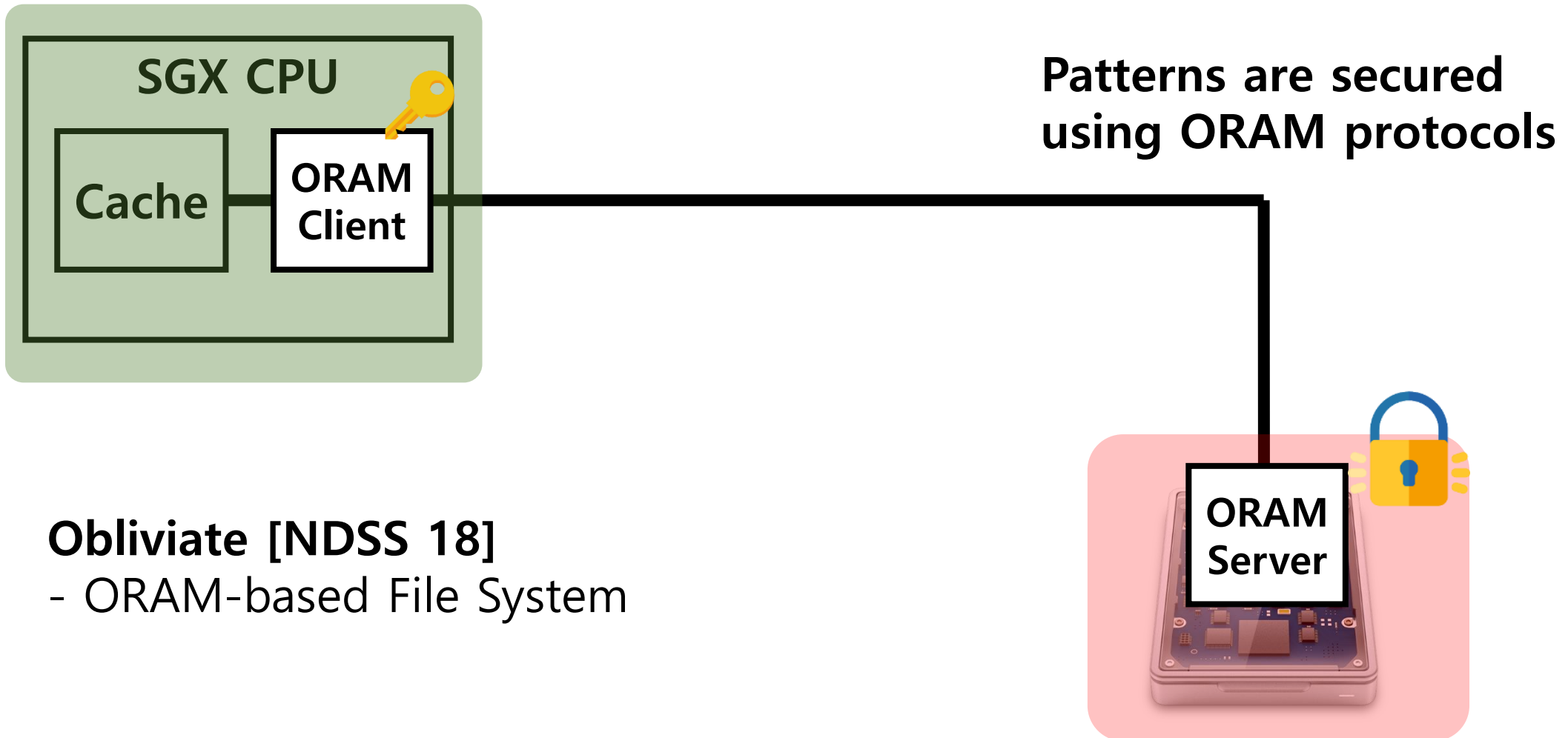
Mitigation: Place Trust in DRAM



InvisiMem [ISCA 17]

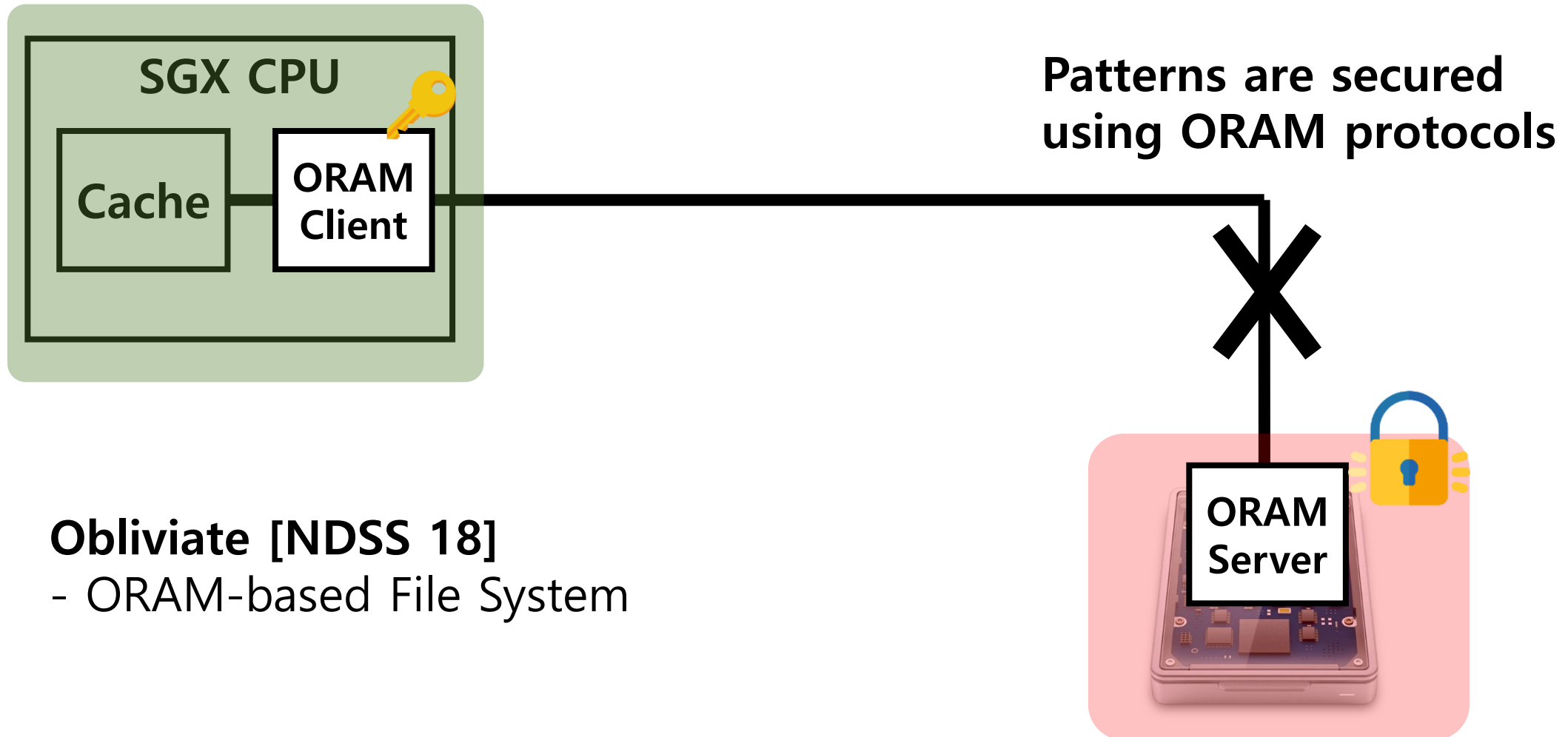
- Place trust in DRAM
- All address and data bus traffics are encrypted
 - ➔ Note: SGX only encrypts values in data bus
- Communication patterns are normalized

Mitigation: ORAM-based File System



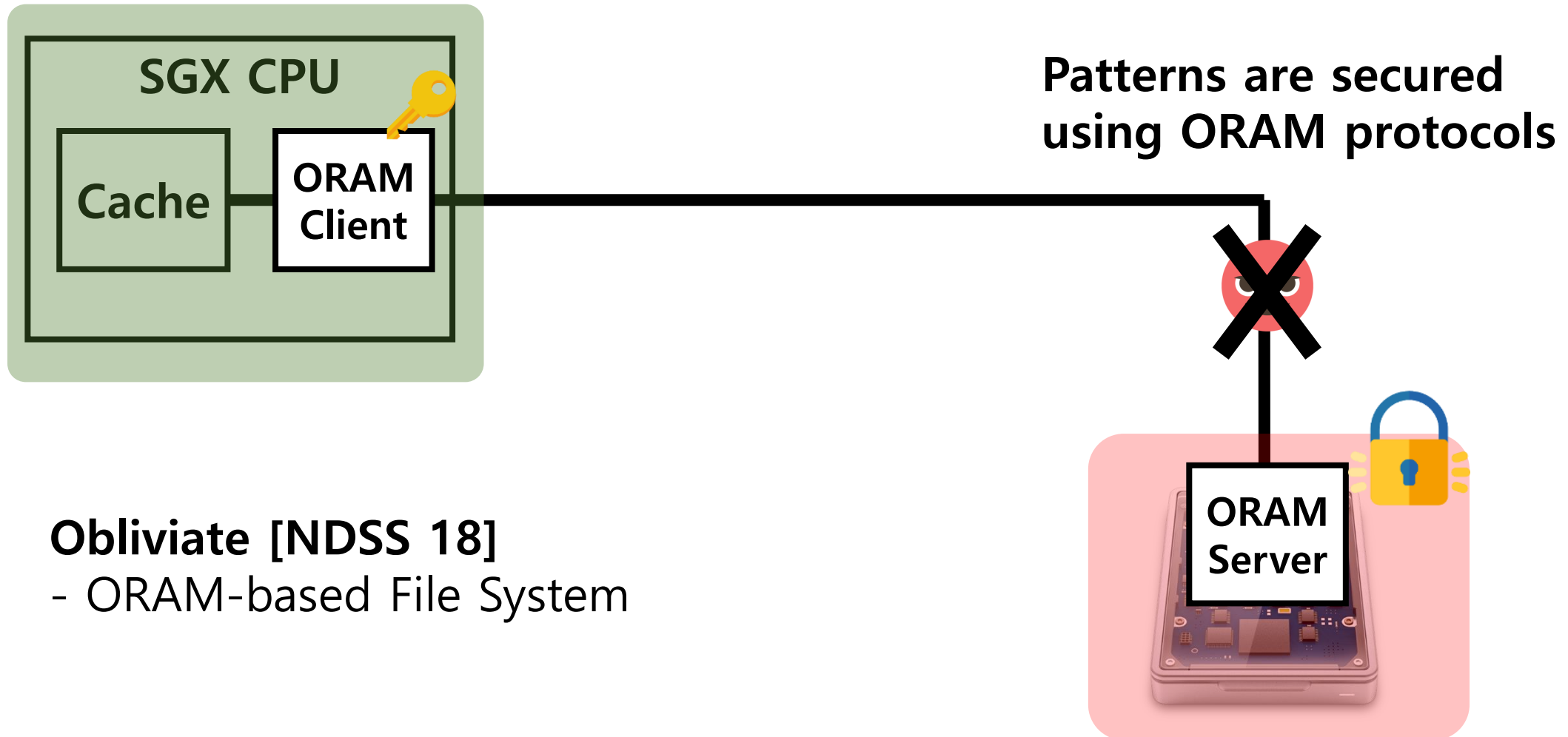
Obliviate [NDSS 18]
- ORAM-based File System

Mitigation: ORAM-based File System

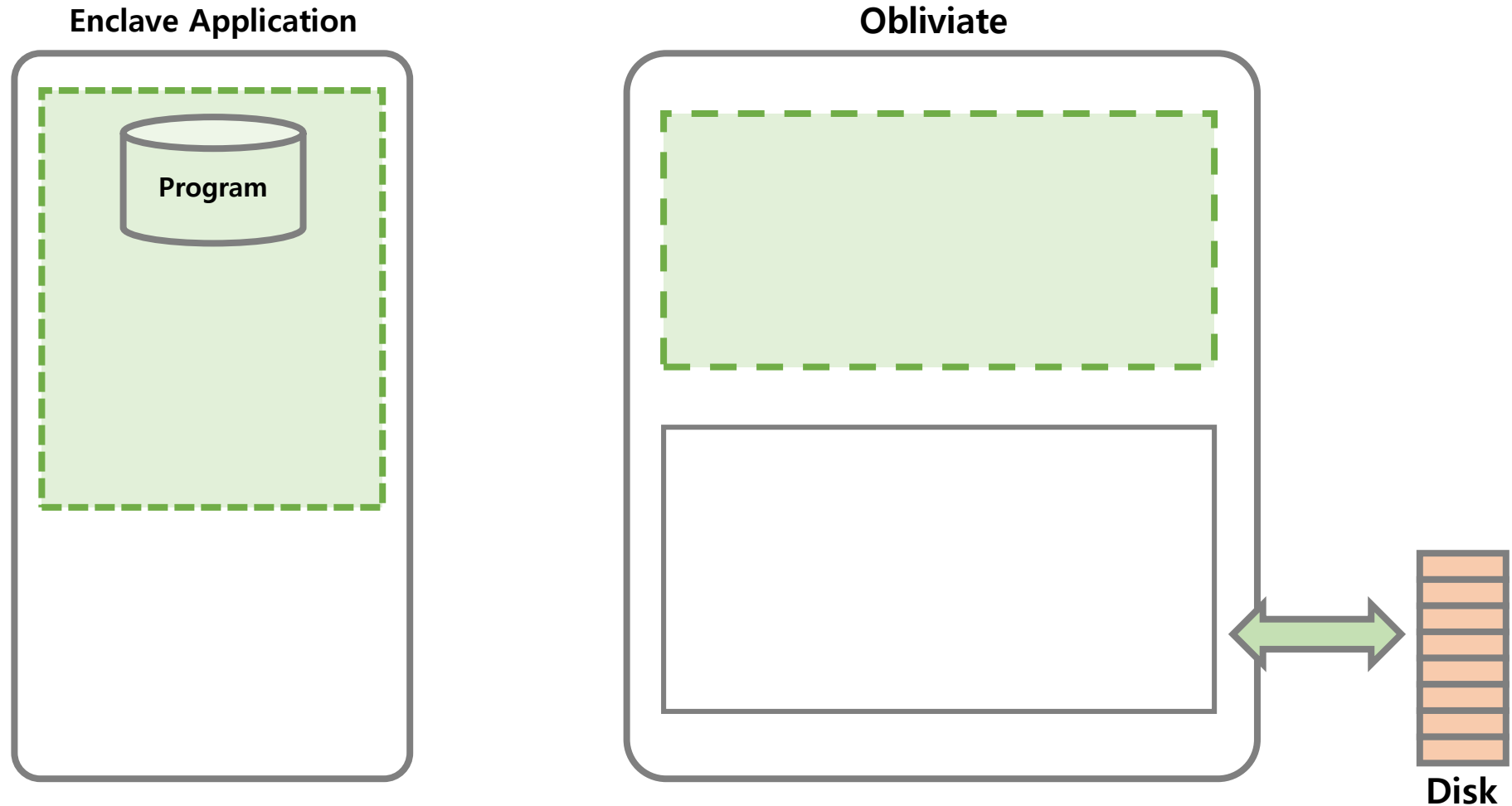


Obliviate [NDSS 18]
- ORAM-based File System

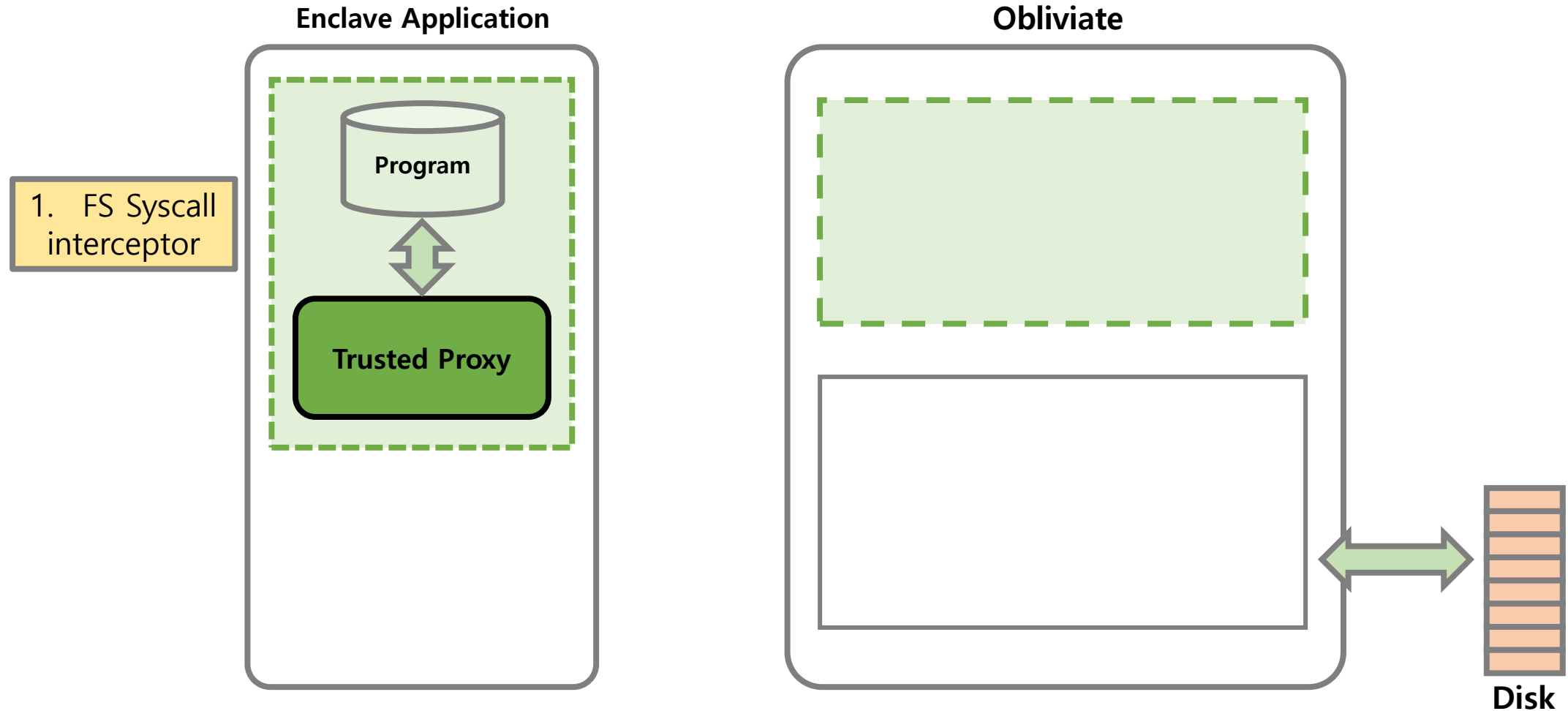
Mitigation: ORAM-based File System



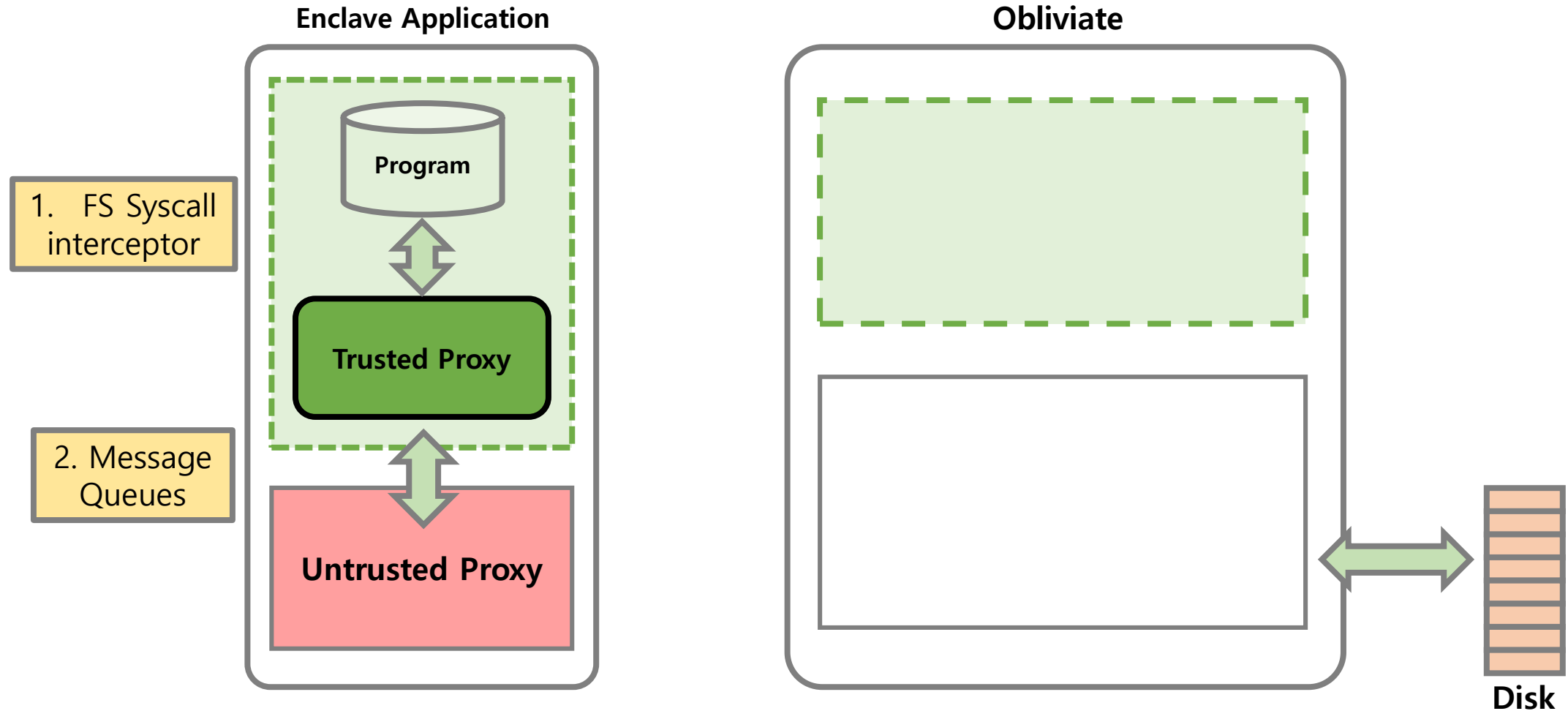
Obliviate [NDSS 18]: Memory charm against the OS



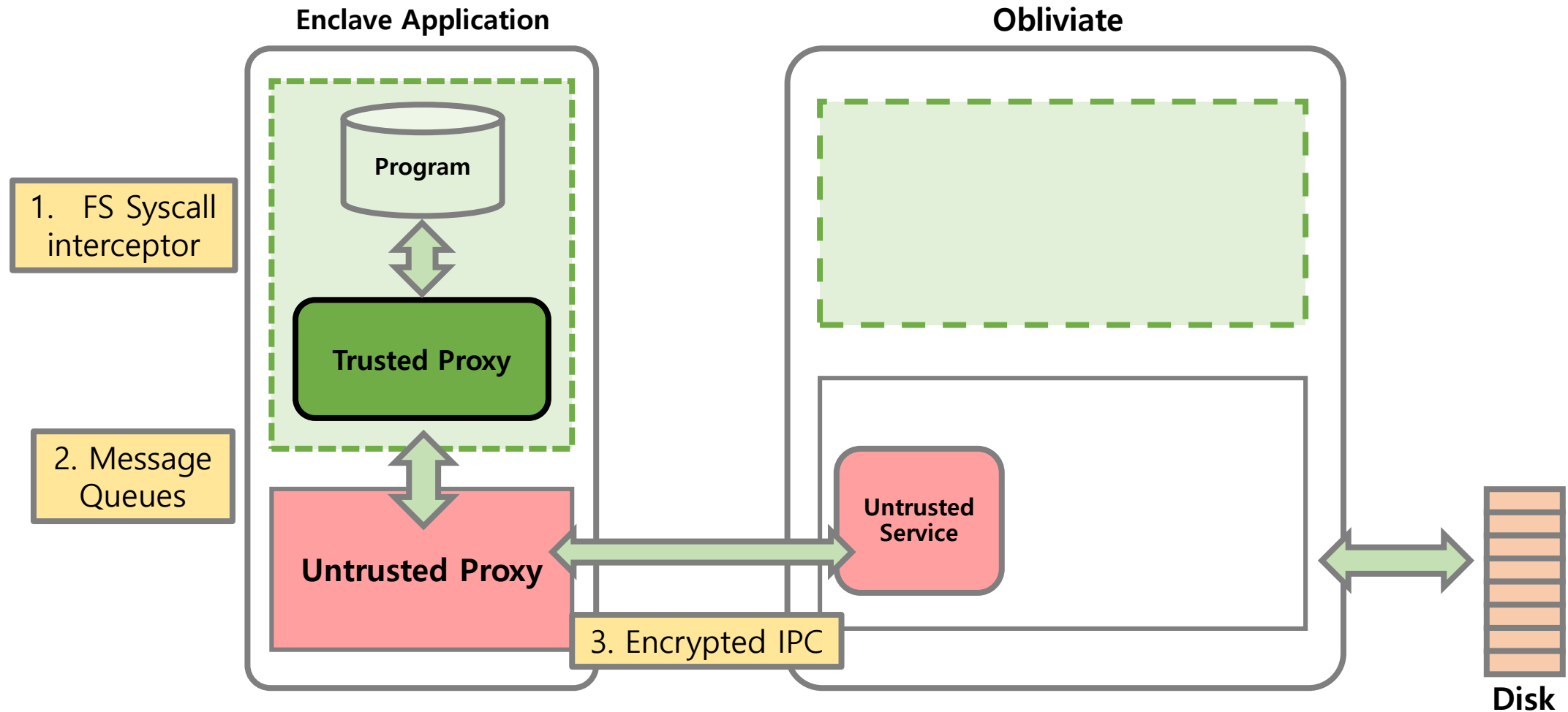
Obliviate [NDSS 18]: Memory charm against the OS



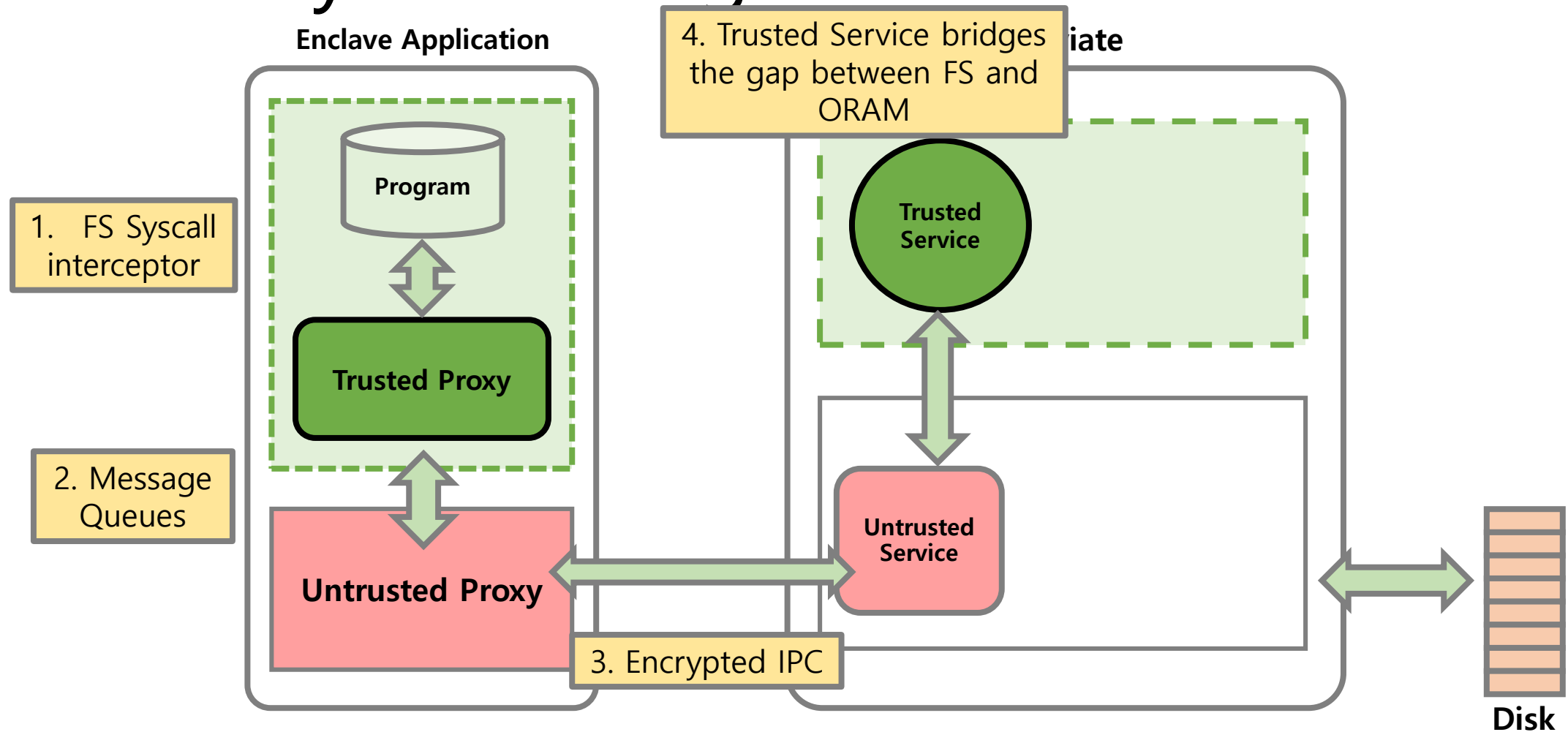
Obliviate [NDSS 18]: Memory charm against the OS



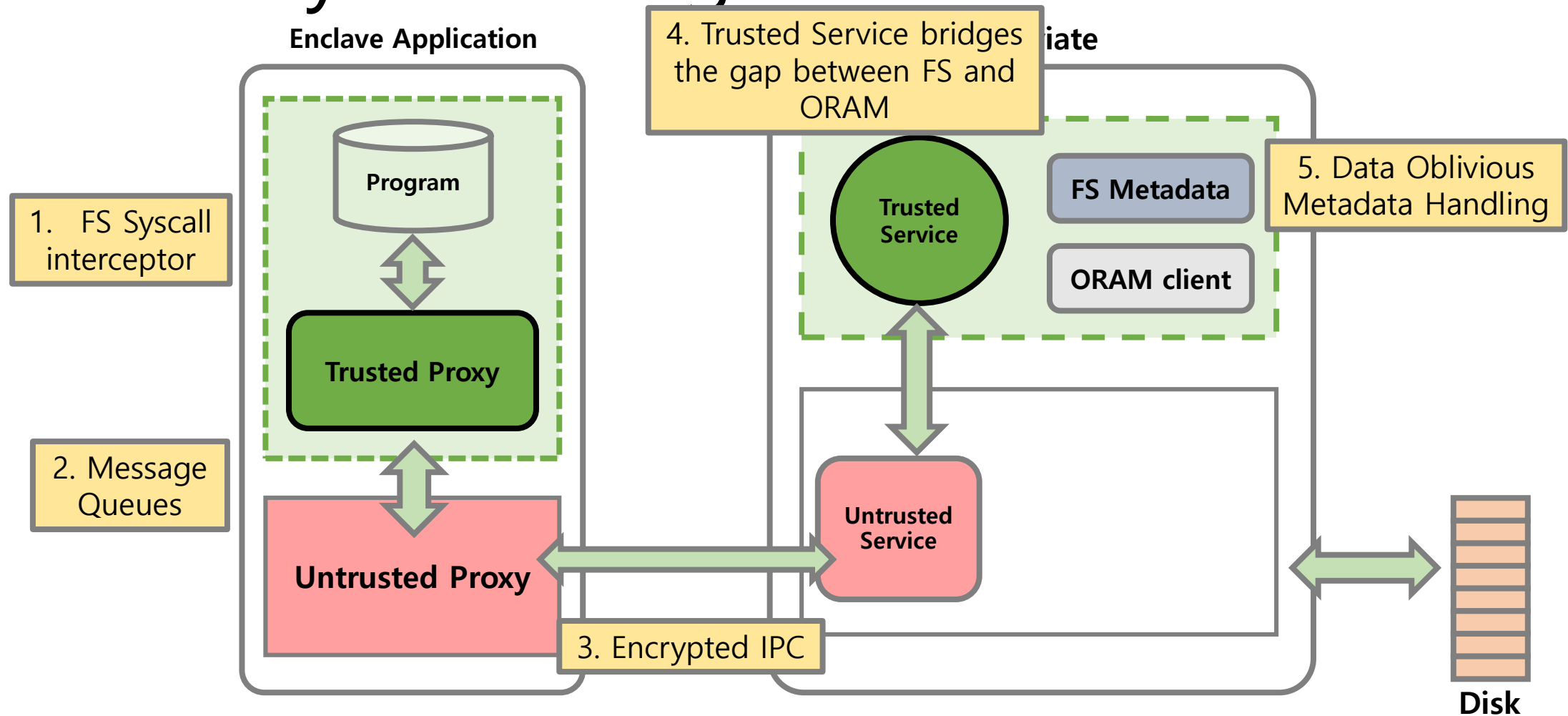
Obliviate [NDSS 18]: Memory charm against the OS



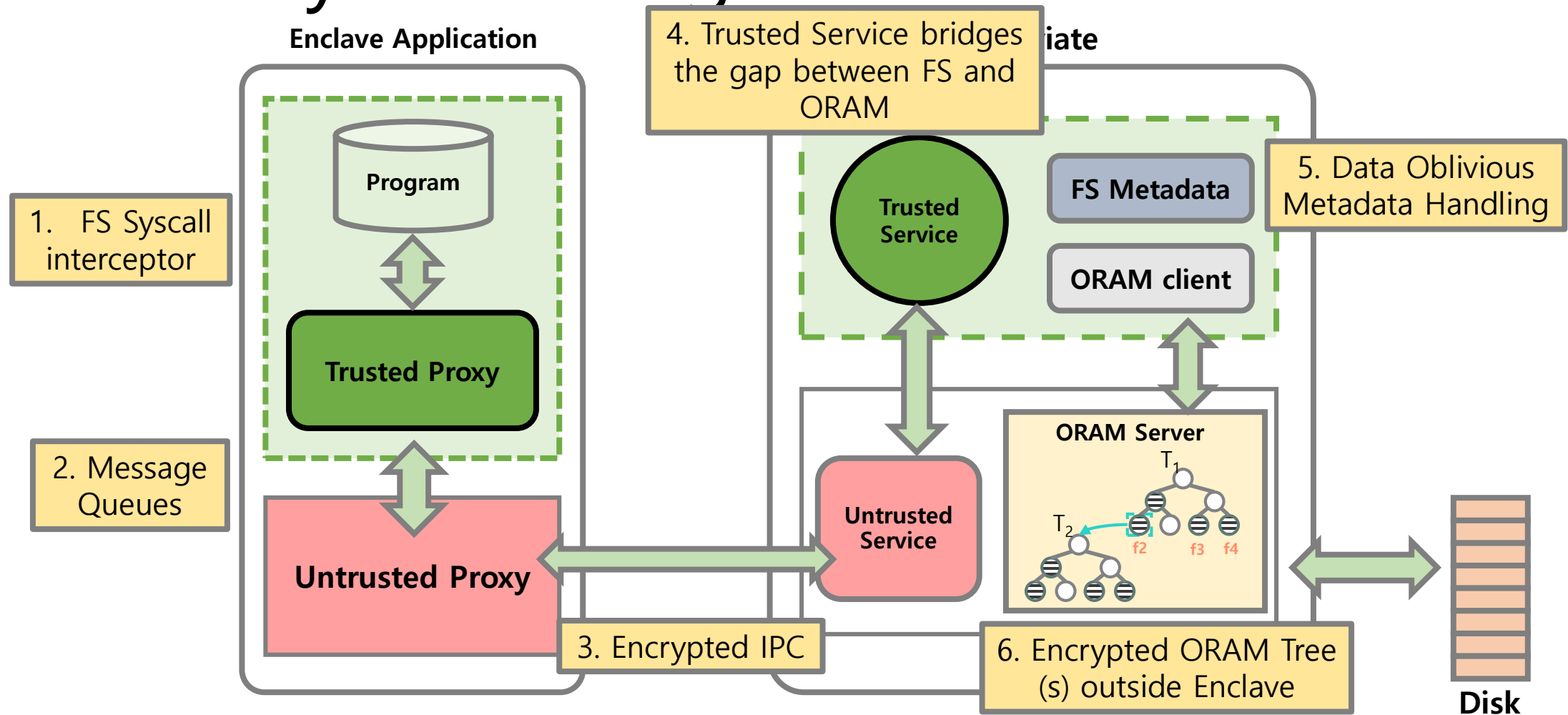
Obliviate [NDSS 18]: Memory charm against the OS



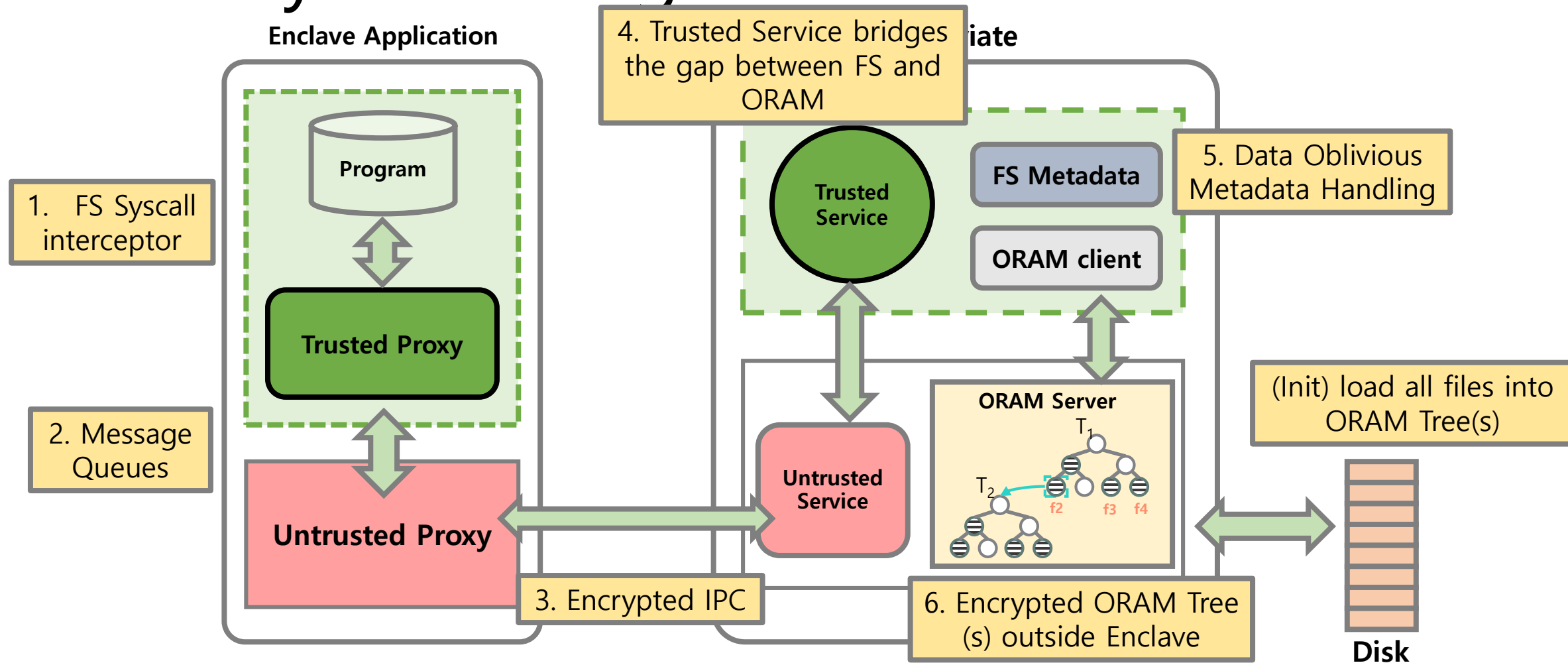
Obliviate [NDSS 18]: Memory charm against the OS



Obliviate [NDSS 18]: Memory charm against the OS



Obliviate [NDSS 18]: Memory charm against the OS



Conclusion

- Bug finding in file systems
 - Semantic, memory, concurrency, error code bugs
 - Semantic inconsistency inference
 - Fuzzing
- Attacks and Defenses
 - Ransomware
 - Cold boot attacks
 - Side channels

감사합니다.

이병영
서울대학교 전기정보공학부
byoungyoung@snu.ac.kr