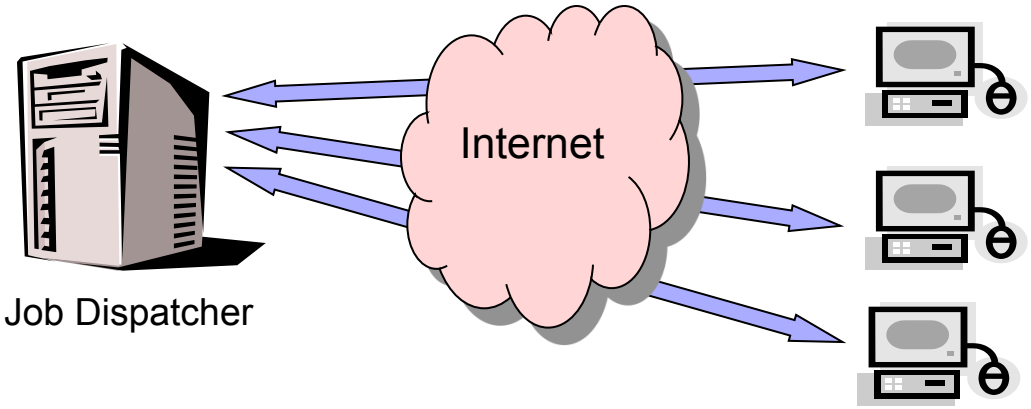
6.857 L17

Secure Processors

Srini Devadas

Distributed Computation

Example: Distributed Computation on the Internet (SETI@home, etc.)

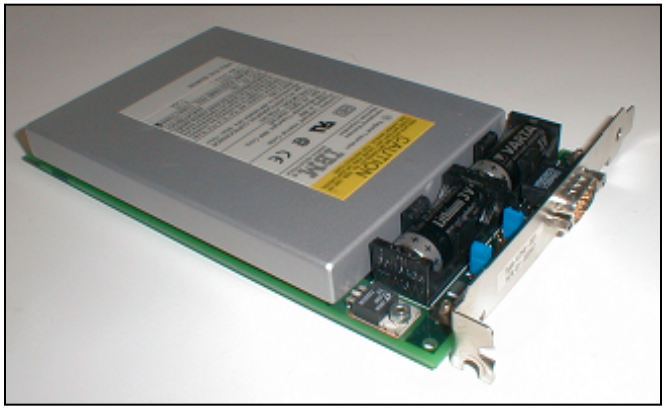



```
DistComp()
{
    x = Receive();
    result = Func(x);
    Send(result);
}
```

- Cannot trust owners
  - Cannot trust software and its result
- Need a secure platform
  - Dispatcher can authenticate “hardware” and “software”
  - Guarantees the integrity and privacy of “execution”

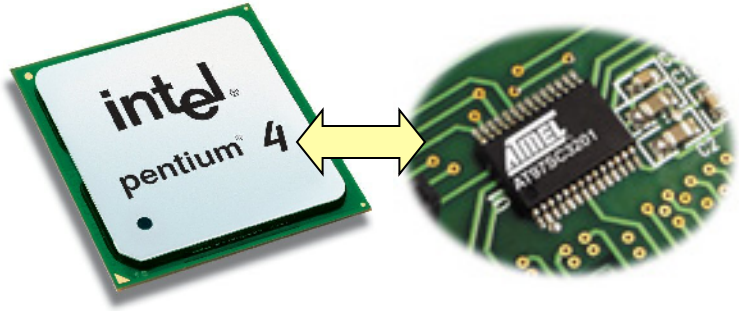
Some Approaches

Tamper-Proof Package: IBM 4758



♪ Sensors to detect attacks
- Expensive and non-scalable
- Continually battery-powered

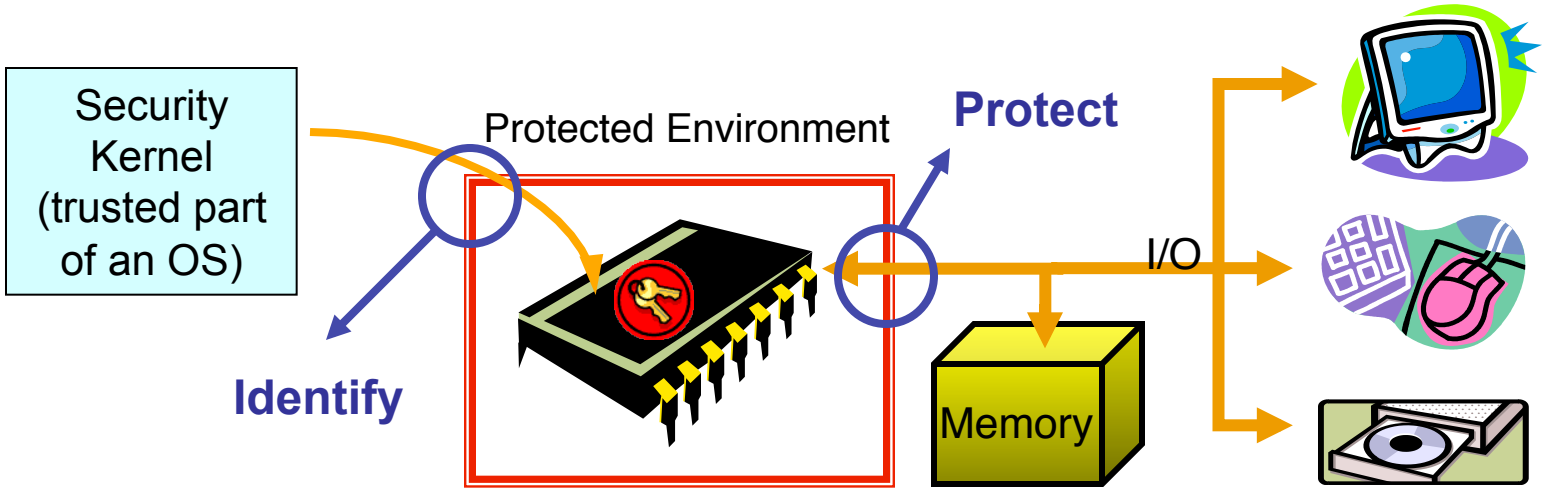
Trusted Platform Module (TPM)



♪ A separate chip (TPM) for security functions
- Decrypted "secondary" keys can be read out from the bus
- Because TPM is passive, can reset and modify registers
- Certifying operating system logistically difficult

Single-Chip Secure Processor

- Build a secure platform with a "single-chip" processor as the only trusted hardware component

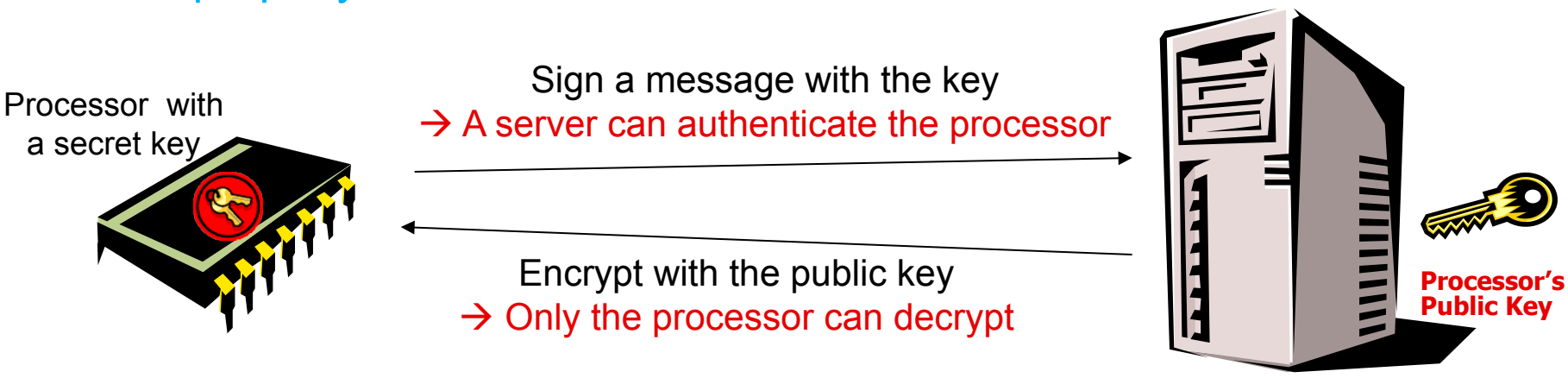


- A single chip is easier and cheaper to protect
- The processor can be authenticated, identifies the security kernel, and protects program state in off-chip memory

Authenticating the Processor

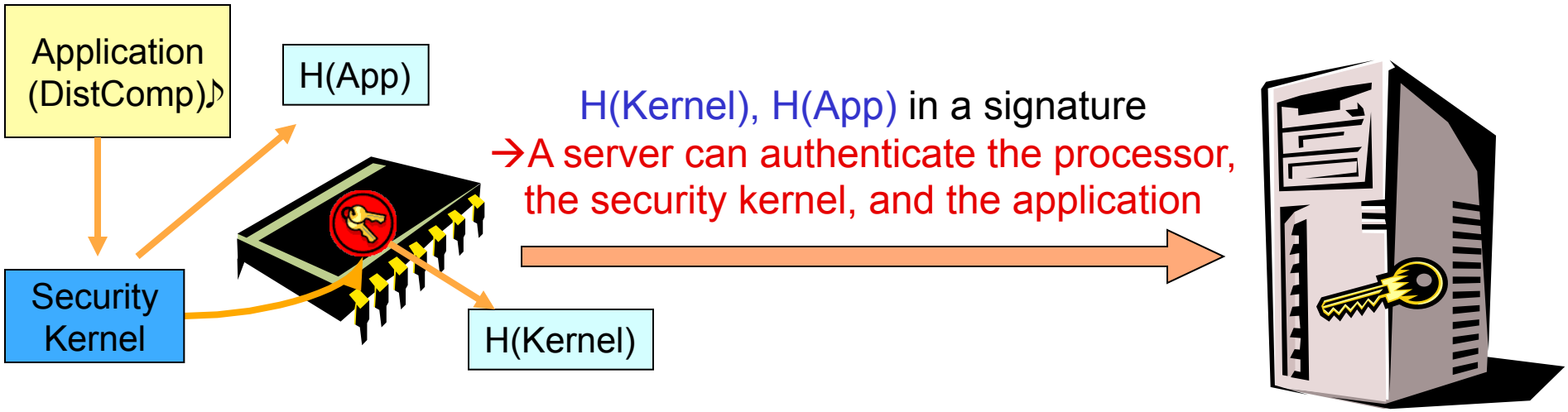
- Each processor should be unique
  - Contains a unique secret key SK

- Use public key cryptography
  - A key infrastructure (such as PKI) certifies the public key
  - PK can check if a message is signed with SK
  - If a message is encrypted with PK, then only SK can decrypt it properly

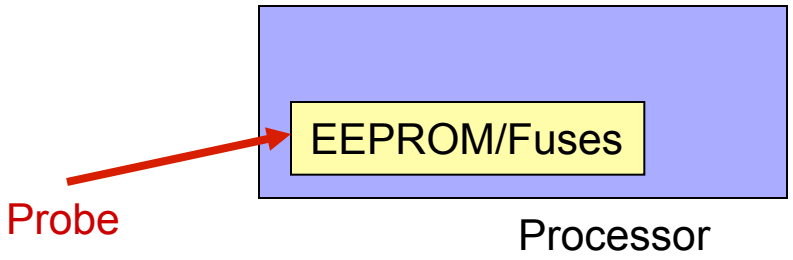


Authenticating Software

- The processor identifies security kernel by computing the kernel's hash (during bootup)
  - Cryptographic hash works as a unique fingerprint
  - Security kernel identifies application programs
- H(Kernel) is included in a signature by the processor
  - Security kernel includes H(App) in the signature



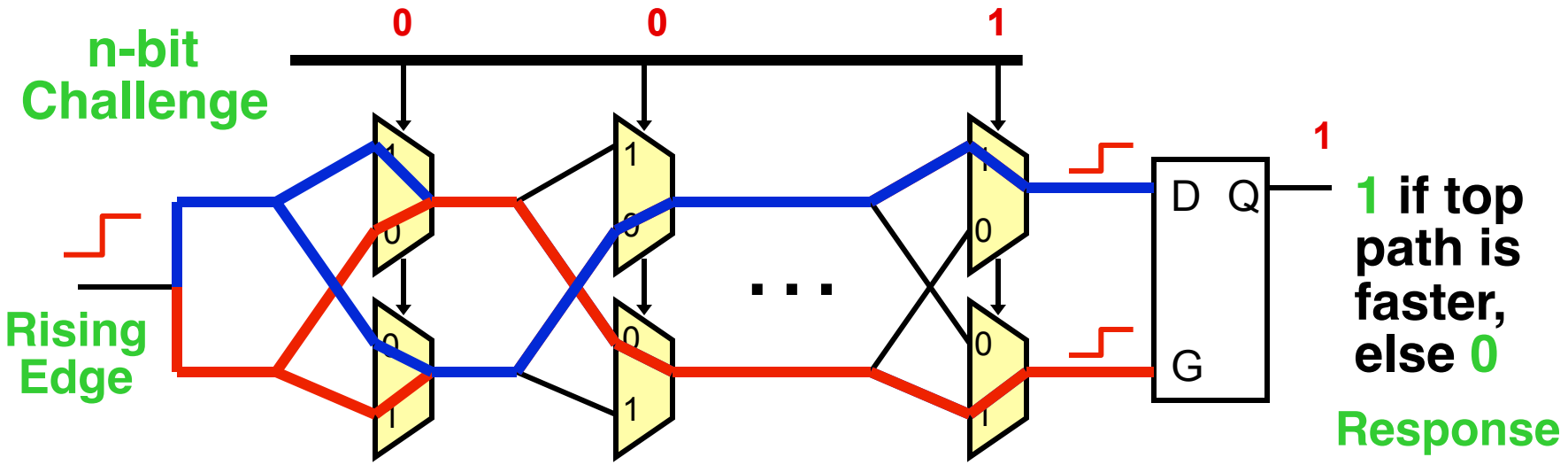
Is one concerned with physical attack?



- Storing digital information in a device in a way that is resistant to physical attacks is difficult and expensive
  - Adversaries can physically extract secret keys from EEPROM while processor is off
  - Trusted party must embed and test secret keys in a secure location
  - EEPROM adds additional complexity to manufacturing
- PUFs (see L16) generate volatile keys that are harder to extract – however, keys are not completely reliable!

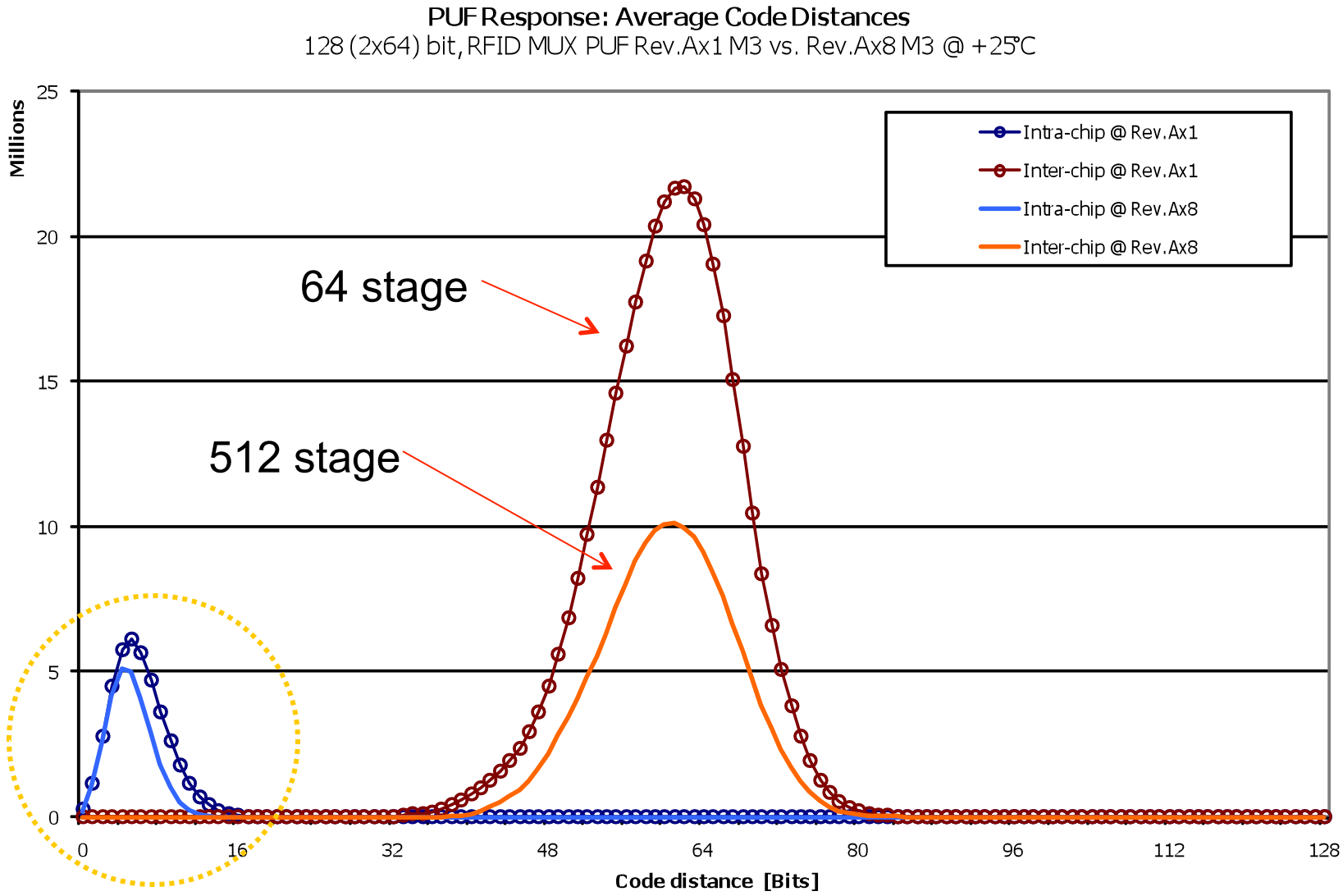
PUFs (Recap)
and
Reliable PUFs

An Arbiter-Based Silicon PUF



- Compare two paths with an identical delay in design
  - Random process variation determines which path is faster
  - An arbiter outputs 1-bit digital response

Arbiter PUF Experiments: 64 and 512 stages

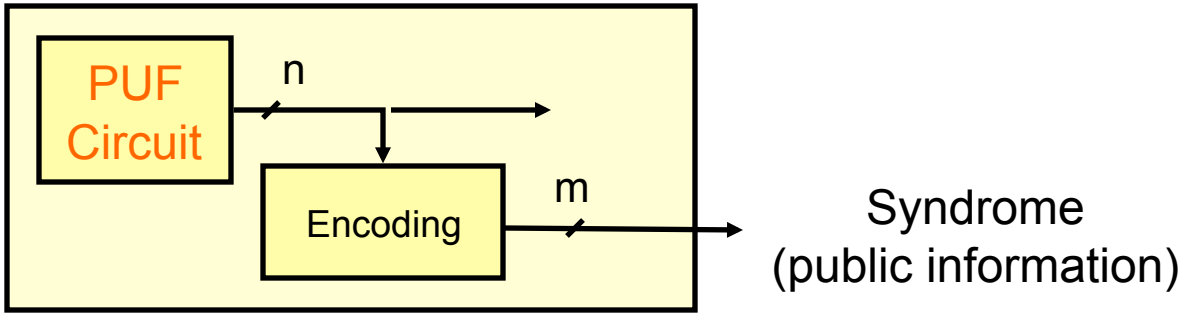


Using a PUF as a Key Generator

- Are only going to generate a fixed number of bits from a PUF
  - Assume small enough number of bits to preclude modeling attacks or that bits are kept secret
- Cannot afford any errors!
- Important question: How to correct errors guaranteeing limited leakage of information?
  - Need to quantify entropy of PUF
  - Need to analyze/quantify leakage due to redundant syndrome bits

Reliable Response Generation: Initialization

Before First Use: Initialization

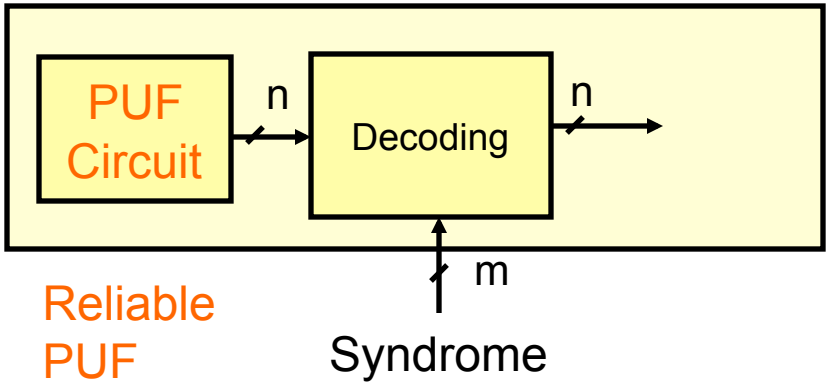


- To initialize the circuit, an error correcting syndrome is generated from the reference PUF circuit output
  - Syndrome is public information
  - Can be stored on-chip, off-chip, or on a remote server

- For example, BCH(127,36,31) code will correct up to 15 errors out of 127 bits to generate 36-bit secret
  - 91-bit syndrome gives away 91 bits of codeword
  - Failure probability will be dependent on PUF error rate

Reliable Response Generation: In the Field

In the Field:
Response
Generation

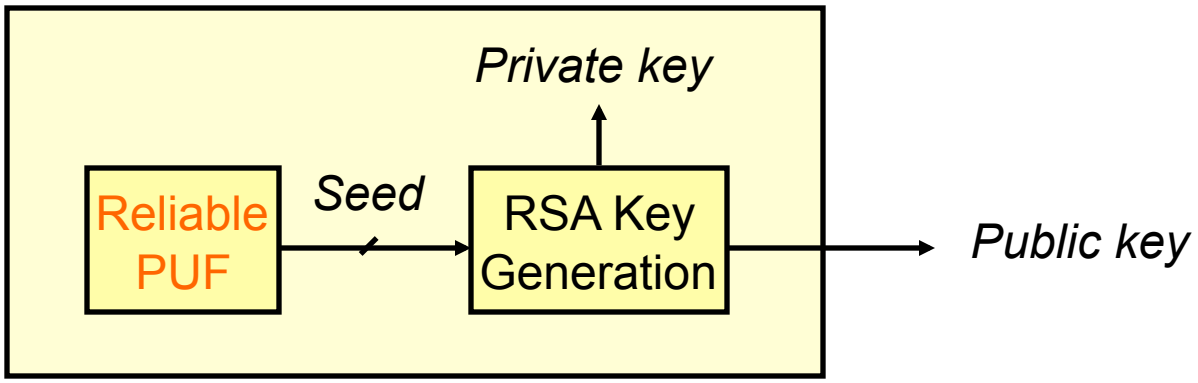


- In the field, the syndrome will be used to re-generate the same PUF reference output from the circuit

Error Correction Complexity

- Some examples of BCH codes that are necessary to correct “raw” PUF outputs
  - (127, 36, 31) gives 36 secret bits, corrects 15 errors; need to run 4 times to get 128-bit secret
  - (255, 63, 61) gives 63 secret bits, corrects 30 errors; need to run twice
- BCH engine complexity grows quadratically with code word size

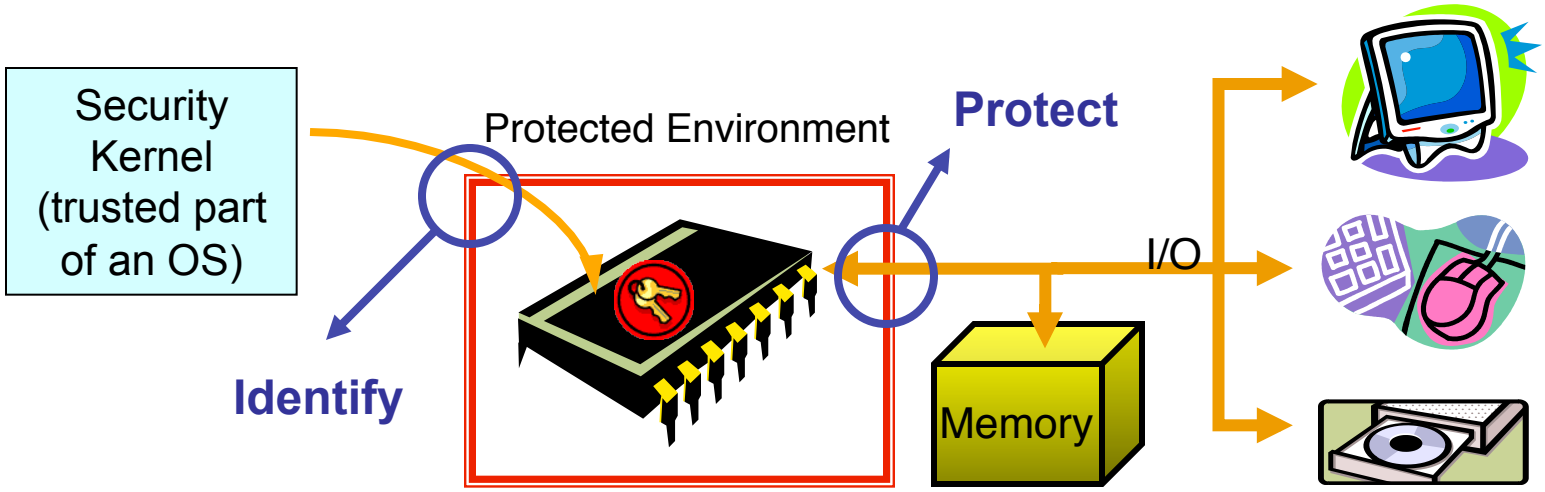
Private/Public Key Pair Generation



- PUF response is used as a random seed to a private/ public key generation algorithm
  - No secret needs to be handled by a manufacturer
- A device generates a key pair on-chip, and outputs a public key
  - The public key needs to be endorsed
  - No one needs to know private key

Single-Chip Secure Processor

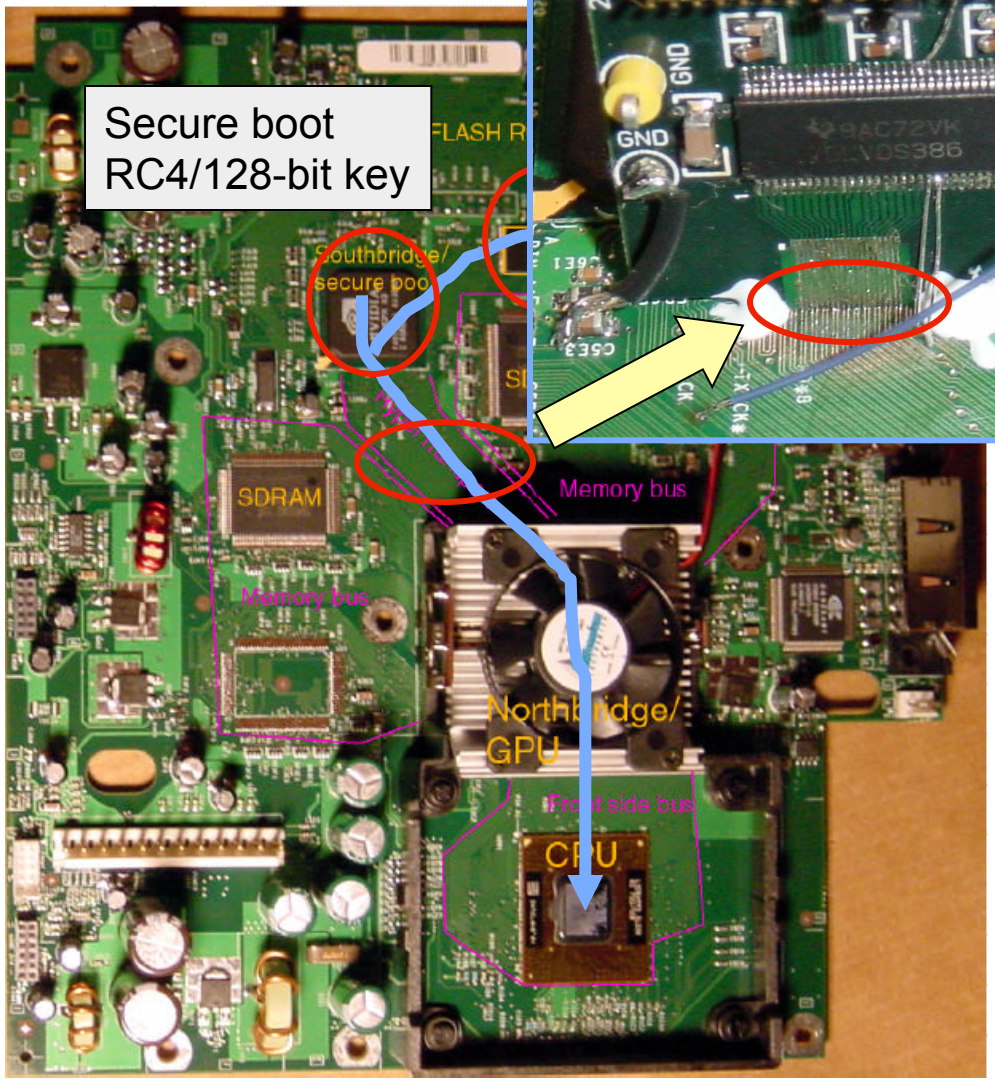
- Build a secure platform with a "single-chip" processor as the only trusted hardware component

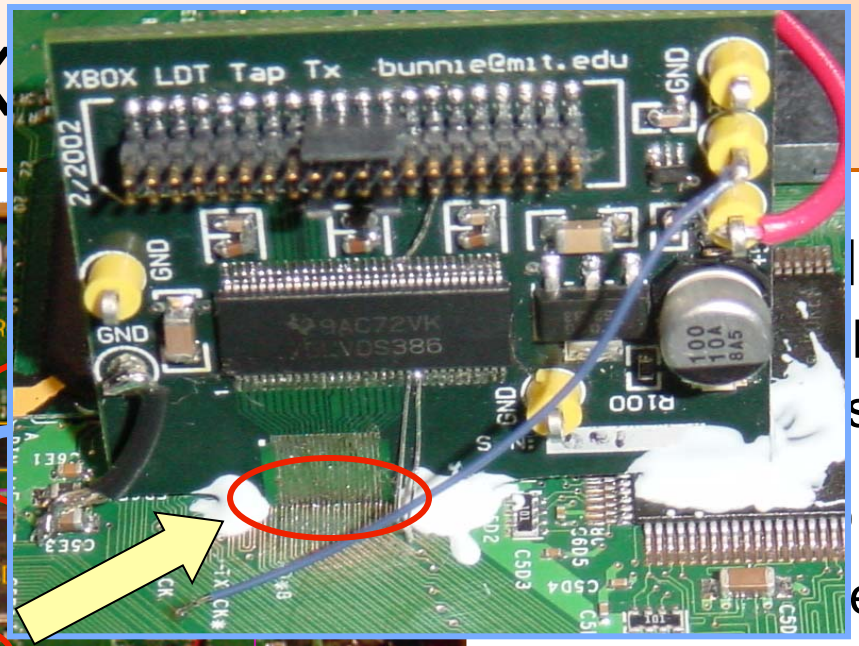


- The processor can be authenticated, identifies the security kernel, and protects program state in off-chip memory

Off-Chip Memory Protection

Microsoft X





lge ASIC
boot code
secret key

M
ed Bootloader

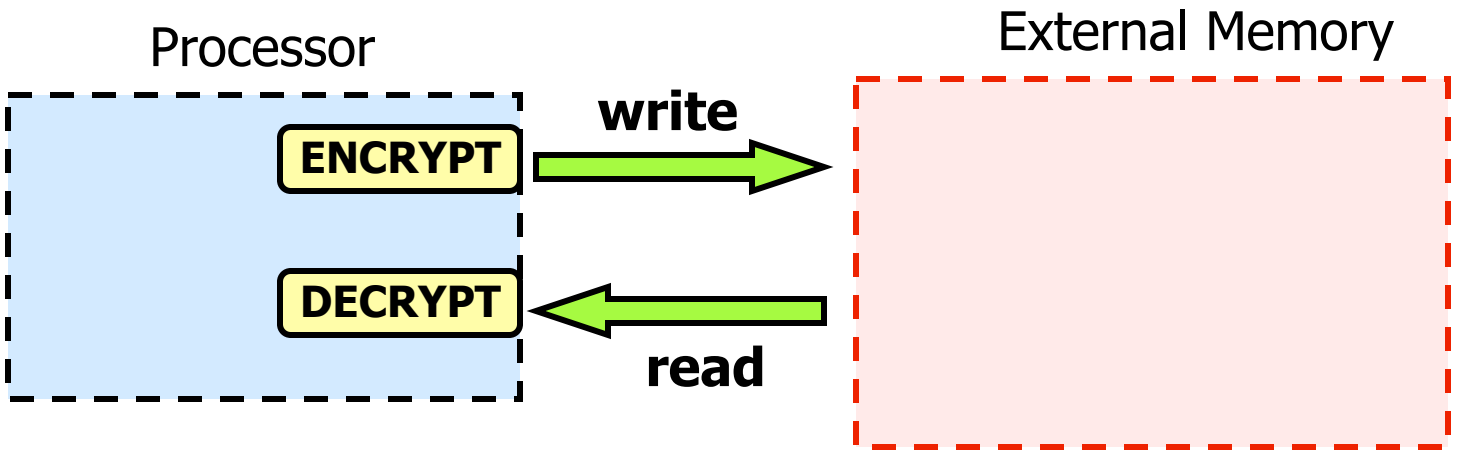
- Broken by tapping bus
  - Read the 128-bit key
  - Cost about $50

Observation

- Adversary can easily read anything on off-chip bus

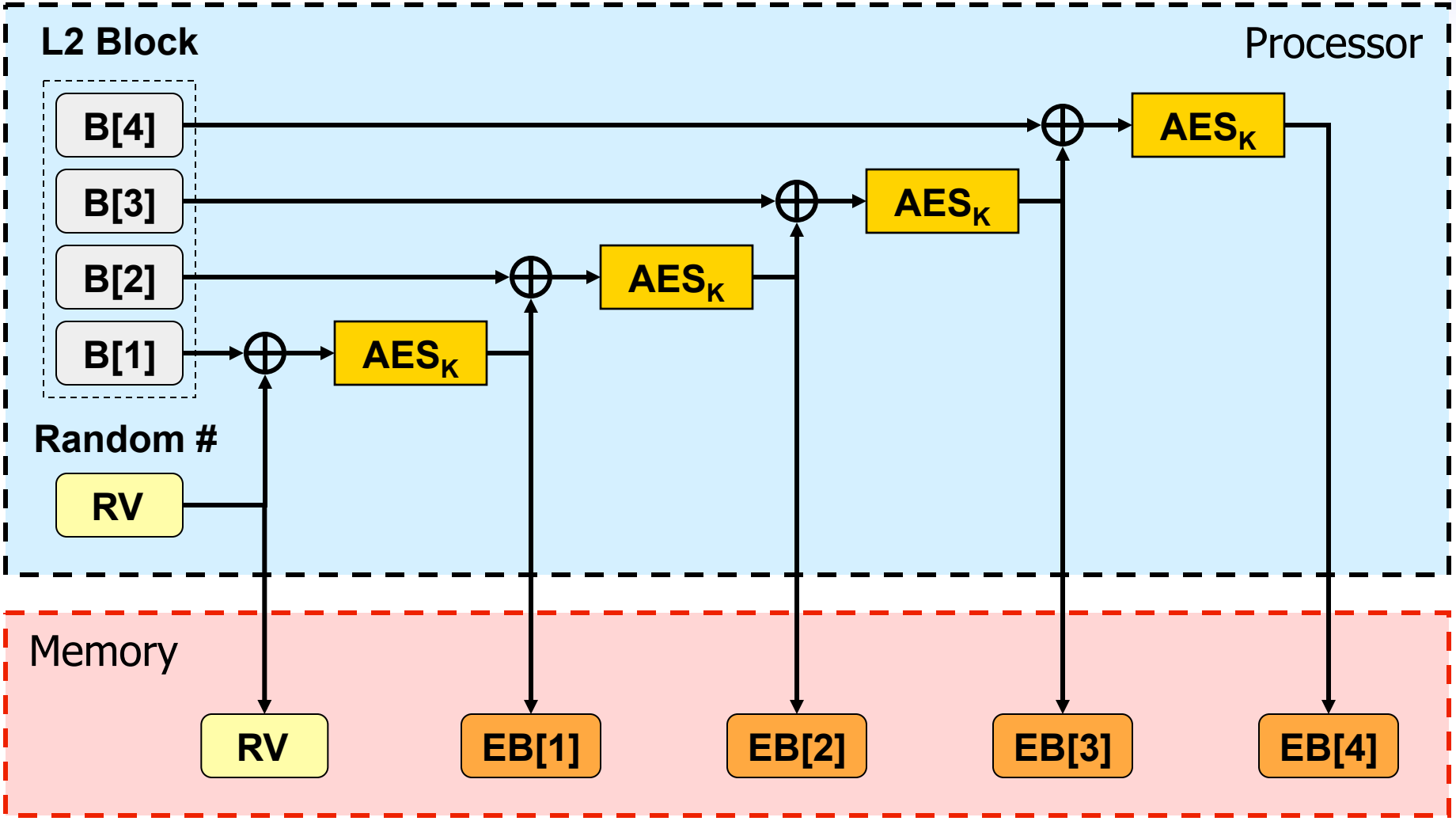
From Andrew “Bunnie” Huang’s Webpage

Memory Encryption

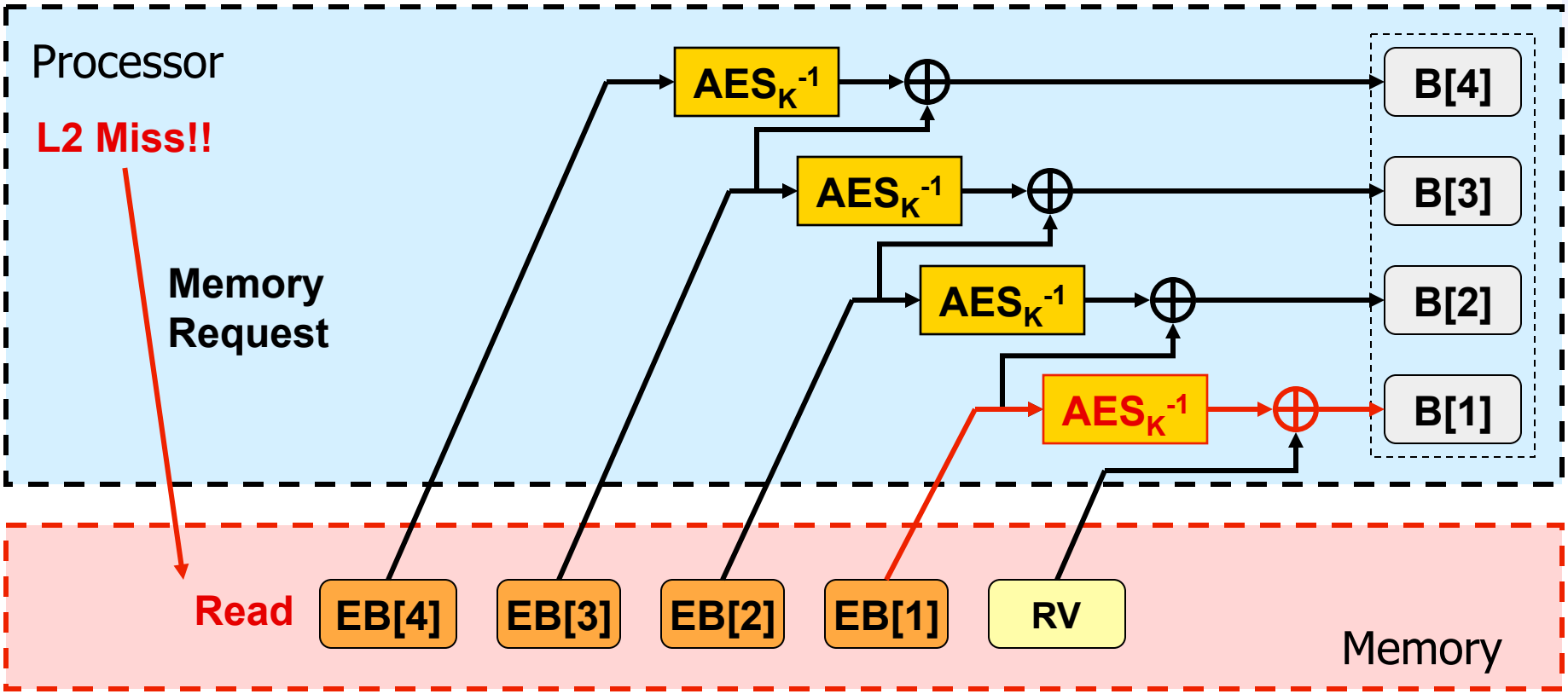


- Encrypt a cache block to protect privacy
  - Must be randomized to prevent comparing two blocks
- Use a fast symmetric key block cipher (3DES, AES)
  - The same processor encrypts and decrypts
  - 16 Byte input → 16 Byte encrypted output
- Decryption can add latency to each memory access

Direct Encryption: encrypt



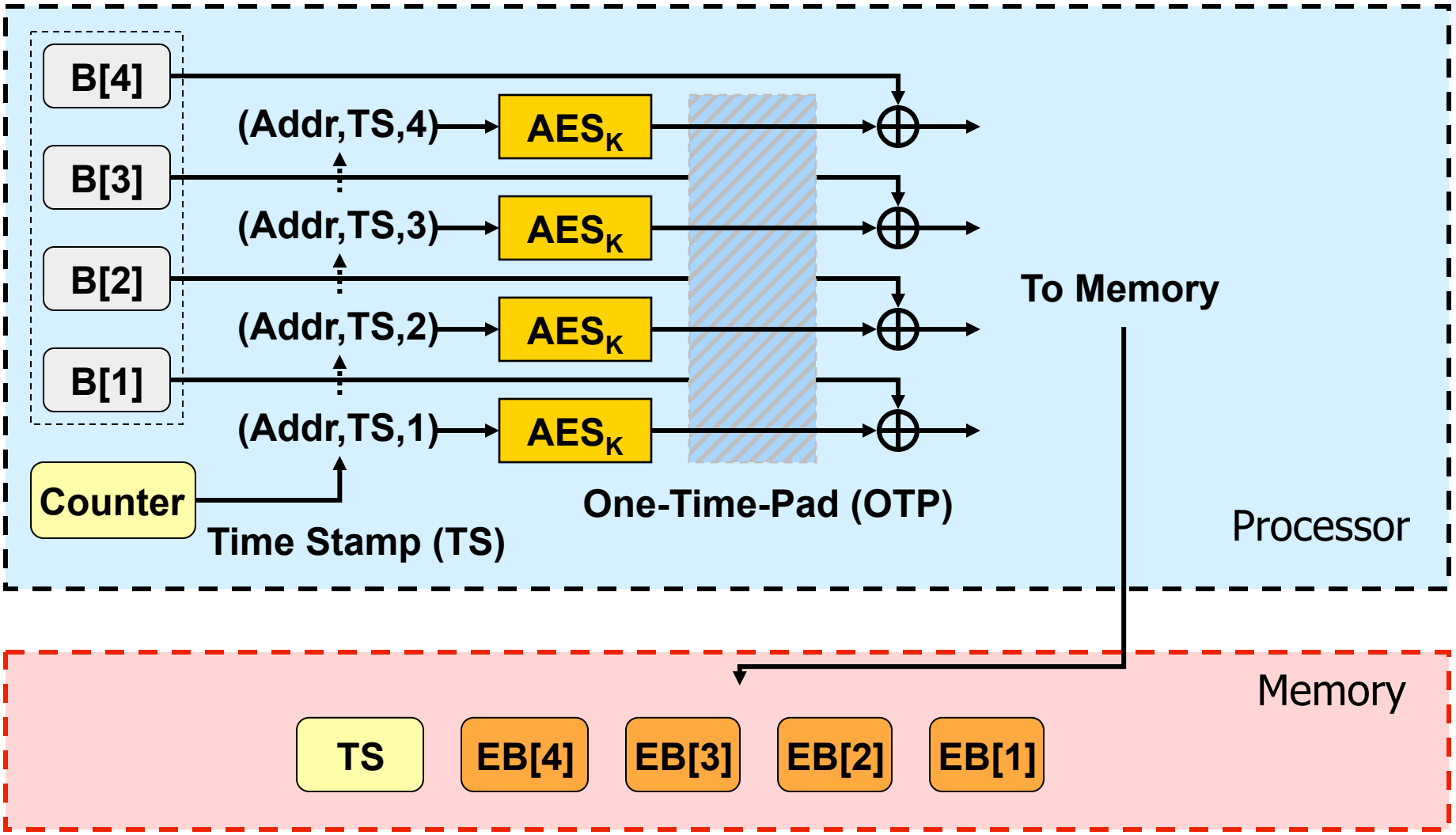
Direct Encryption: decrypt



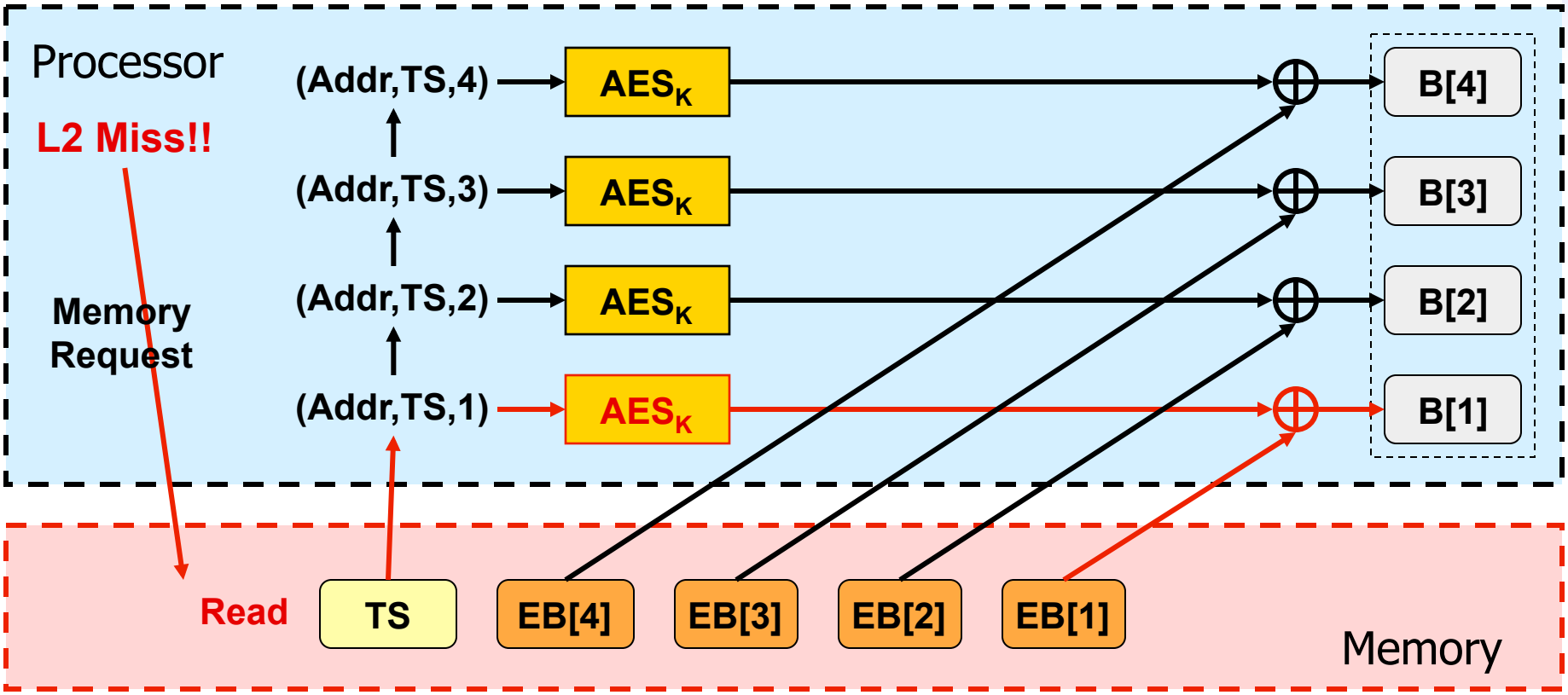
- AES operation can start only after encrypted blocks are read from memory

  →Decryption directly impacts off-chip latency

Counter-Mode Encryption: encrypt

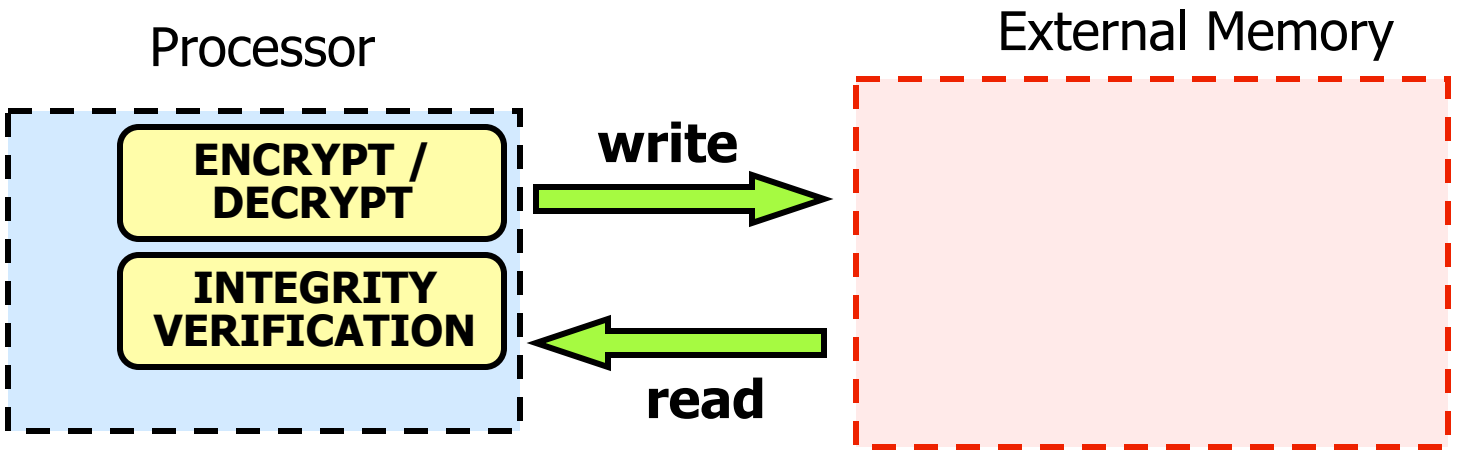


Counter-Mode Encryption: decrypt



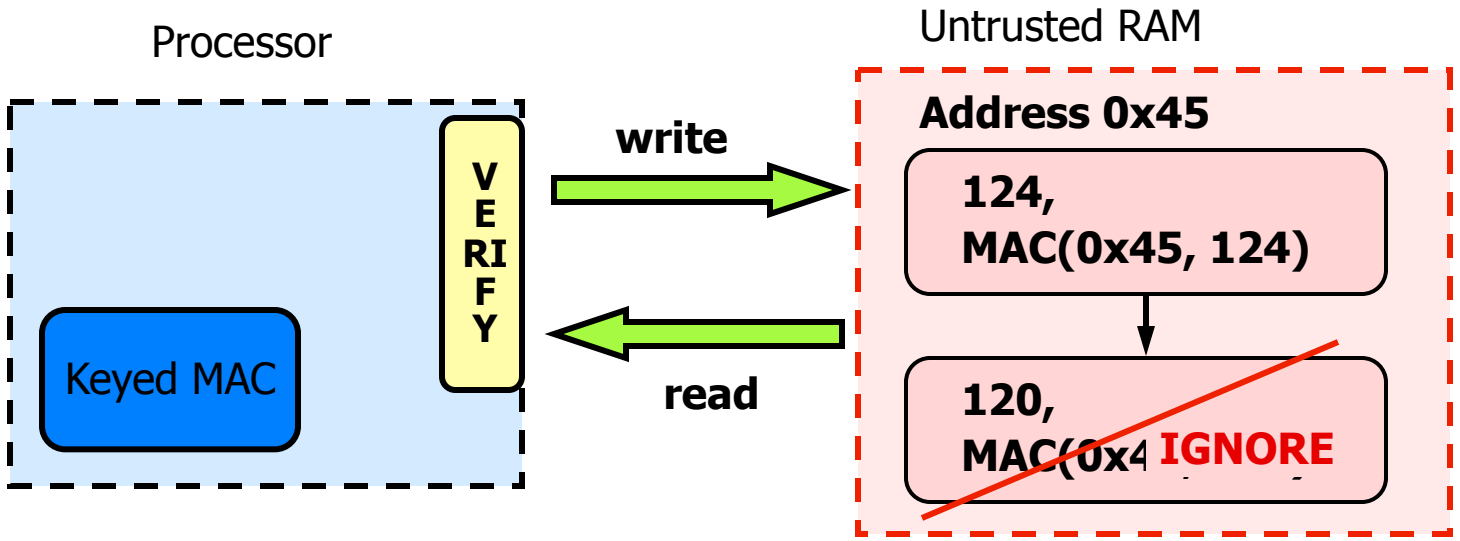
- AES can be performed in parallel to memory accesses
  →Reduces the overhead by 40% on average

Integrity Verification



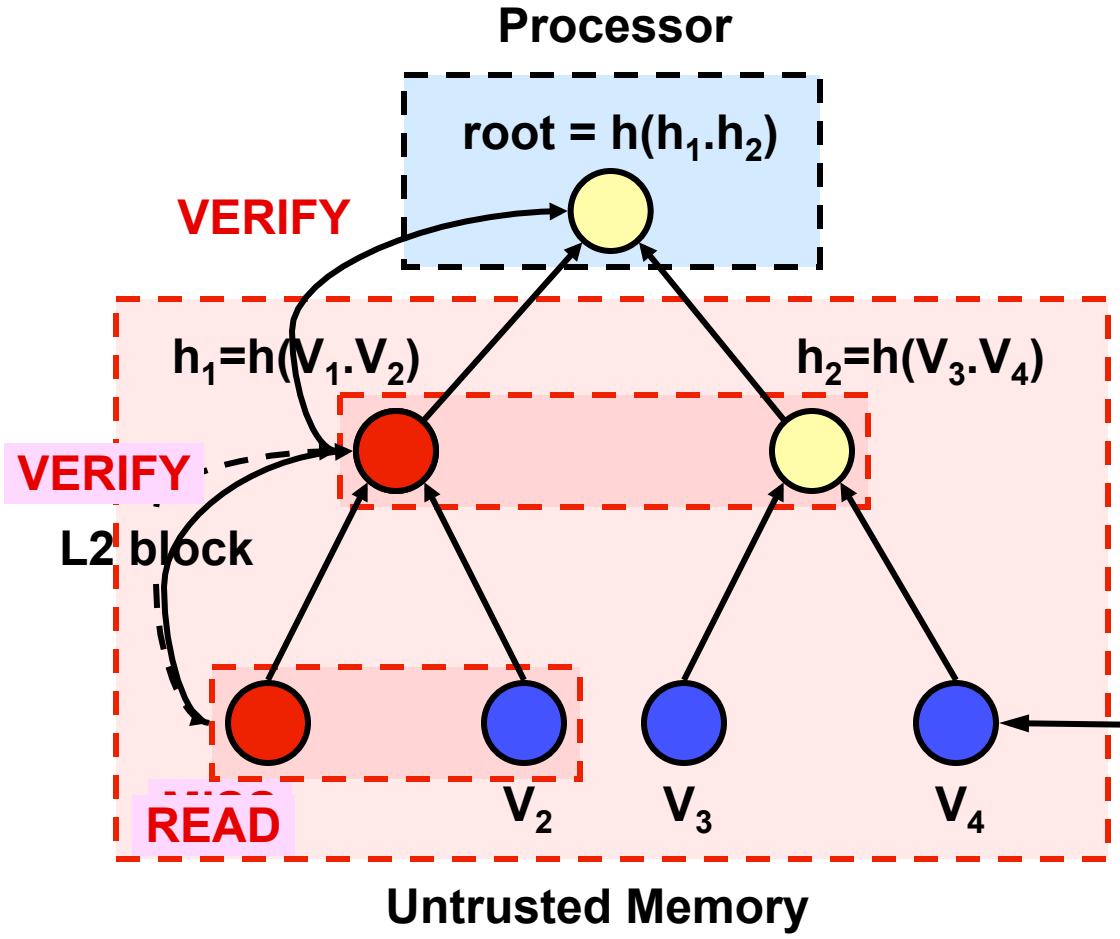
- Integrity Verification
  - Check if a value from external memory is the most recent value stored at the address by the processor

MAC-based Integrity Verification?



- Message Authentication Code (MAC) is often used to authenticate a network message
- Store MAC(address, value) on writes, and check the MAC on reads (used by XOM architecture from Stanford)
  - Does NOT work → Replay attacks
- Need to securely remember the off-chip memory state

Hash Trees (Merkle Trees)



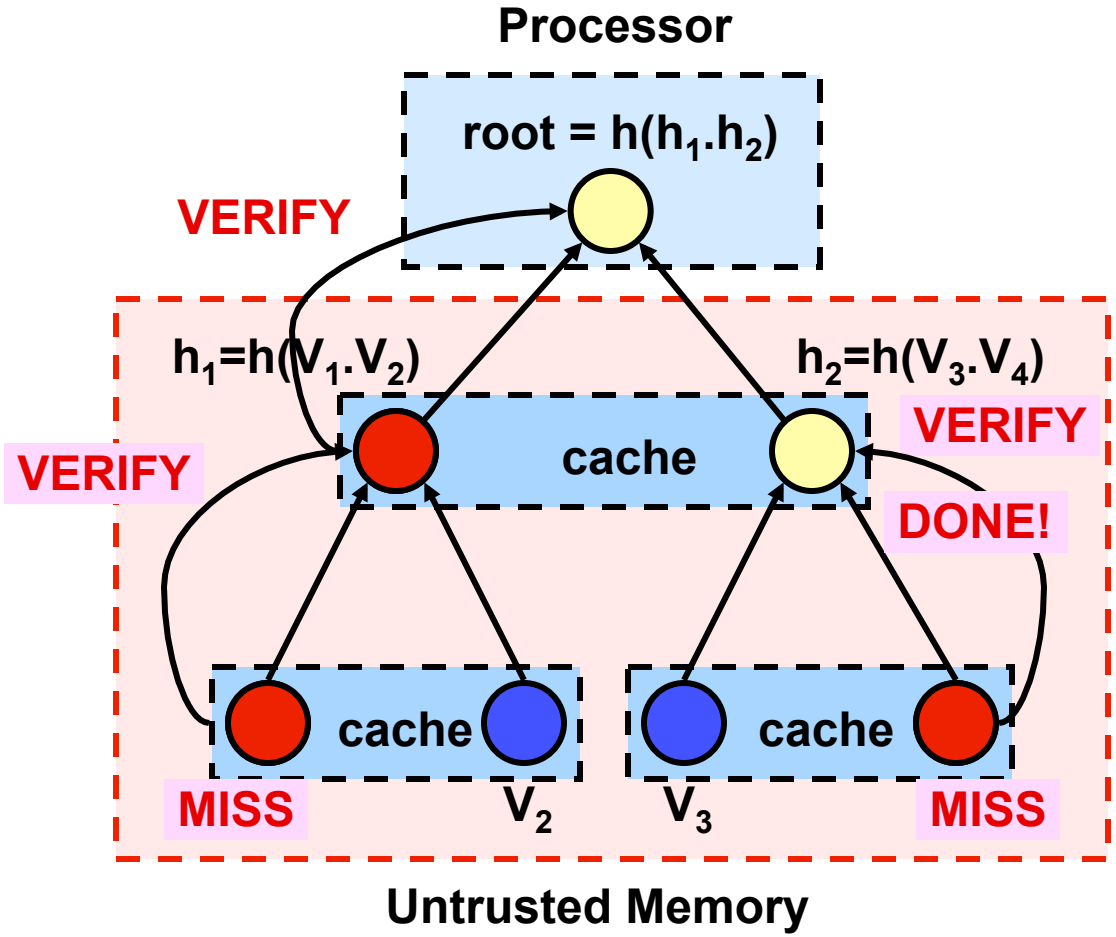
Construct a hash tree

- On-chip Storage: 16-20 Bytes
- Read/Write Cost: logarithmic

Data Values

→ 10x slowdown

Cached Hash Trees



Cache hashes on-chip

→ On-chip cache is trusted

→ Stop checking earlier

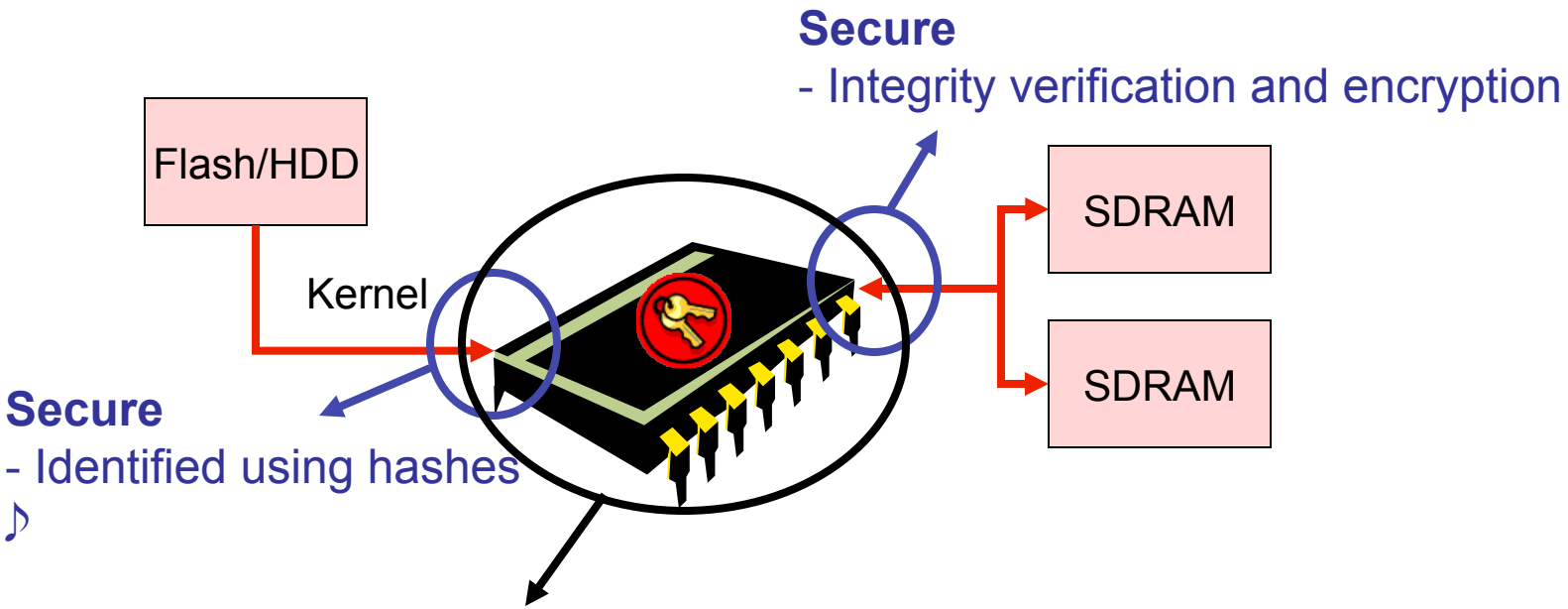
Hiding Verification Latency♪

- Integrity verification takes at least 1 hash computation
  - SHA-1 has 80 rounds → 80 cycles to compute
- Speculatively use the value and check in the background
  - Not a security problem for most instructions
  - No need for precise exception; simply abort
- Except for instructions that can compromise security
  - Example: signing with a private (secret) key


```
load r0,(r1)
addi r0,r0,#3
store (r1),r0
```

Security Review

- Have we built a secure computing system?



For greater physical security use PUFs and techniques from smart cards and physical intrusion detection circuitry (IBM 4758)