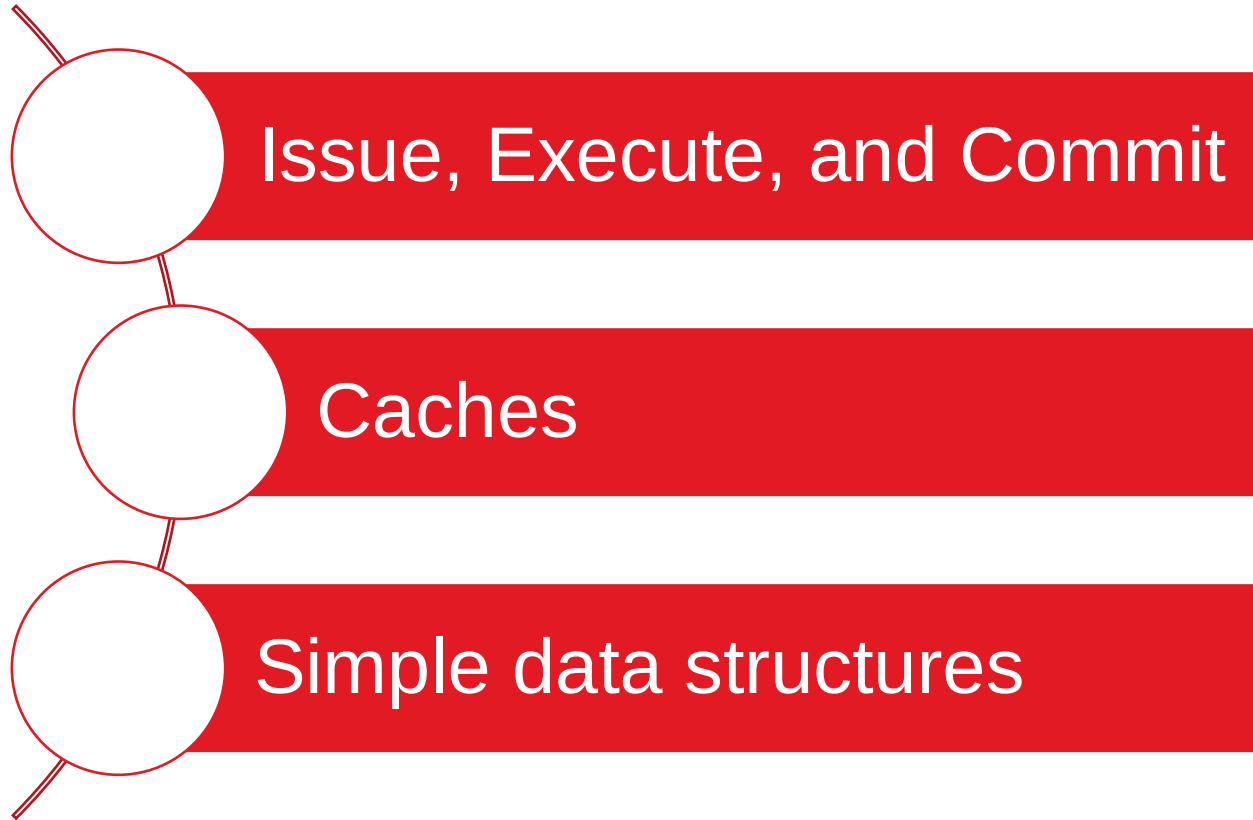




Chapter 13: Secure Processors

Background Required to Understand this Chapter



Chapters 4
and 7

Outline



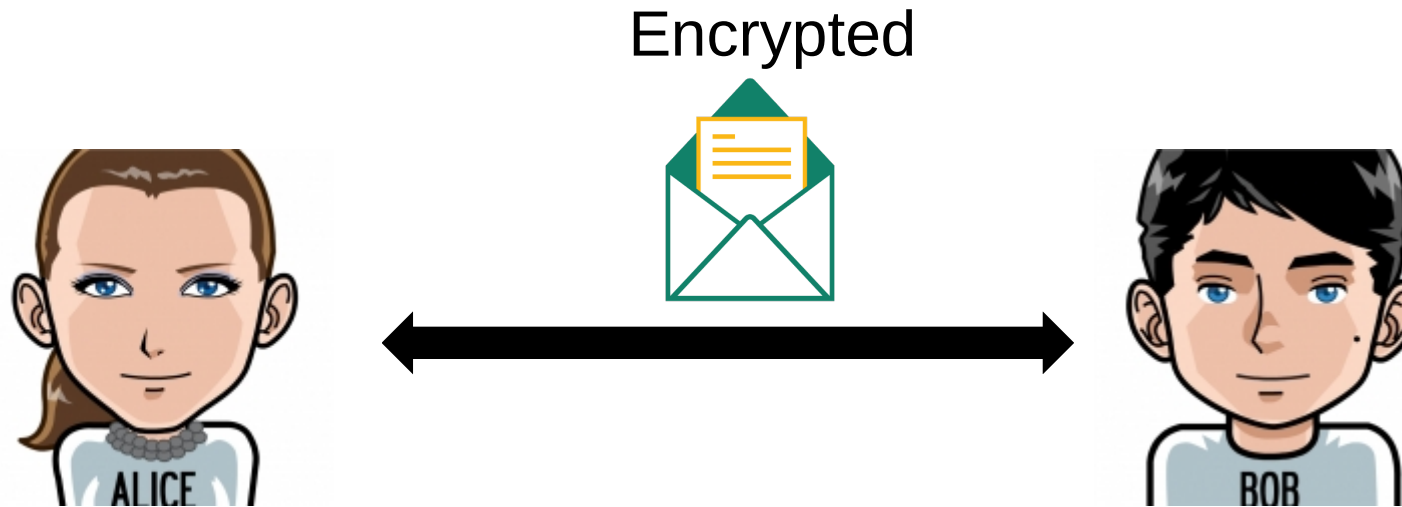
1. Data Encryption

2. Hashing and Data Integrity

3. Secure Architectures

4. Side-Channel Attacks

How to keep a secret?



Confidentiality

Data is **encrypted**, and the original data cannot be **inferred** from the encrypted data.

Integrity

The data has not been **tampered** with

Plaintext



Encryption



Ciphertext

Data Encryption



The two **key** concepts in data encryption.

Confusion

If we change a single bit of the key, most or all of the ciphertext bits will be affected. This ensures that the key and ciphertext are not correlated (in a statistical sense), and thus given the ciphertext, it is hard to guess the key.

Diffusion

This property states that if we change a single bit in the plaintext, then statistically half the bits in the ciphertext should change, and likewise if we change one bit in the ciphertext then statistically half the bits in the plain text should change. This reduces the correlation between the plaintext and the ciphertext.

Two Kinds of Ciphers

- Block ciphers \Rightarrow **Divide** data into blocks of 128-256 bits. **Encrypt** the data block by block.
- Stream ciphers \Rightarrow Generate an infinite **sequence** of pseudorandom numbers. The **ciphertext** is the XOR of the n^{th} plaintext byte and n^{th} **random number**.



Block cipher

- **Key sizes**: 128, 192, or 256 bits
- Divide plaintext into sizes equal to the <key size>
- The algorithm runs in **rounds**

Key size	Rounds
128	10
192	12
256	14

Steps in AES

- **Running example:** Key with 128 bits
- Consider a piece of **text** with the same **length**
- **Divide** it into 16 blocks: 1 byte each

$$\begin{bmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{bmatrix}$$

State of the
algorithm

In each round

Operation 1: SubBytes

Each byte B_i is **replaced** by a byte C_i
Use a lookup table: **S-box**

Operation 2: ShiftRows

Left-rotate the i^{th} row by i positions \Rightarrow
Left-shift and then **move** the byte that
was **shifted** out to the rightmost position

AES Operations – II



Operation 3: MixColumns

Take the four bytes in each **column** and (modular) **multiply** it with a matrix. Last two operations **diffuse** bits.

Operation 4: AddRoundKey

Compute a bitwise XOR with the **round key** (128-bit block)

Generating Round Keys

Basic Operations

Operation 1: RotWord (R)

$$B_0 B_1 B_2 B_3 \rightarrow B_1 B_2 B_3 B_0$$

Operation 2: SubWord (S)

Substitute each byte in a word using the SubBytes function.

Operation 3: XORWord (X)

For a word of the form:

RC \Leftarrow array of round constants (1 per round)

Generating Round Keys – II

$$\begin{bmatrix} K_0 & K_4 & K_8 & K_{12} \\ K_1 & K_5 & K_9 & K_{13} \\ K_2 & K_6 & K_{10} & K_{14} \\ K_3 & K_7 & K_{11} & K_{15} \end{bmatrix}$$

Key matrix

- Every algorithm needs a 128-bit **key** (16 bytes)
- Convert it to a key **matrix** (key matrix for round 0)
- Let K_j^i be the j^{th} **key** in the **matrix** for round i
- For every subsequent round, it **changes** according to the key **schedule**

$$K_j^i = \begin{cases} K_j^{i-1} + K_{j-4}^i & 4 \leq j \leq 15 \\ K_0^i K_1^i K_2^i K_3^i = X \left(S \left(R \left(K_{12}^{i-1} K_{13}^{i-1} K_{14}^{i-1} K_{15}^{i-1} \right) \right) \right) \oplus \\ K_0^{i-1} K_1^{i-1} K_2^{i-1} K_3^{i-1} \end{cases}$$

Full AES Algorithm (Assume N rounds)

0th round

Only the AddRoundKey operation is performed

N-1 rounds

All 4 operations are performed

Nth rounds

The *MixColumns* operation is skipped



The state of the algorithm is the final ciphertext



Encrypt a large piece of text

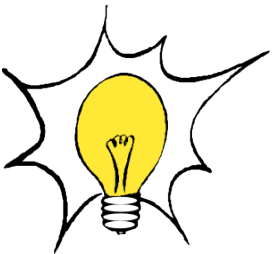
Encryption Modes

Electronic Codebook

Break text into 128-bit sized blocks and **encode** each one separately. High confusion, low diffusion.

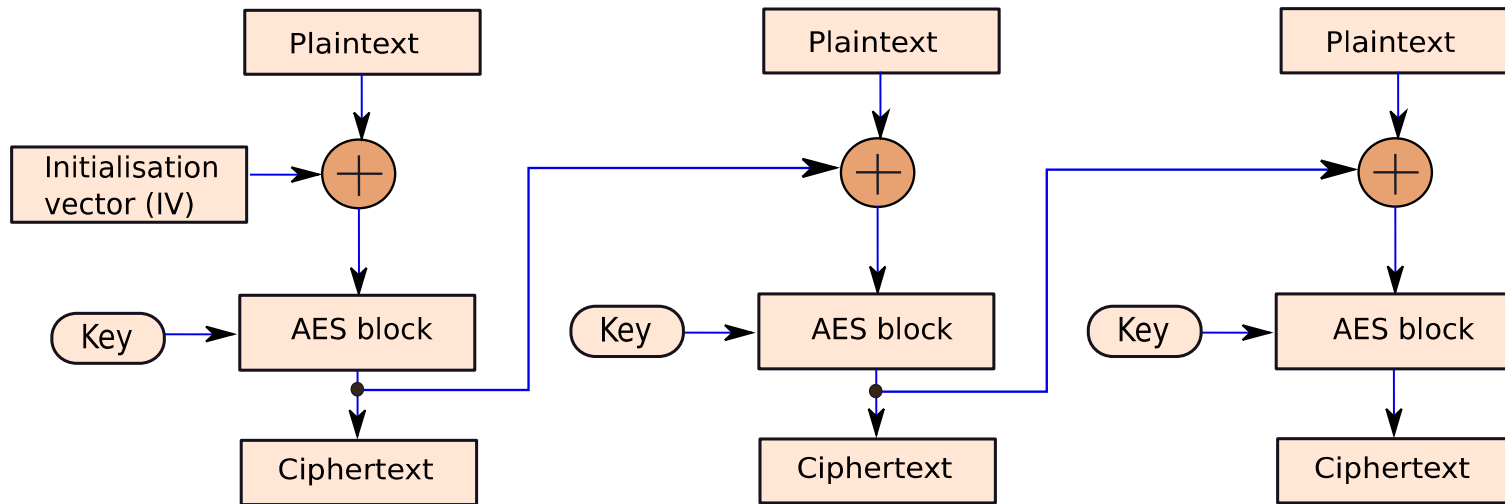


Increase diffusion



Link the encryption of one block to the next one. This will ensure that a large part of the ciphertext will change.

Cipher Block Chaining

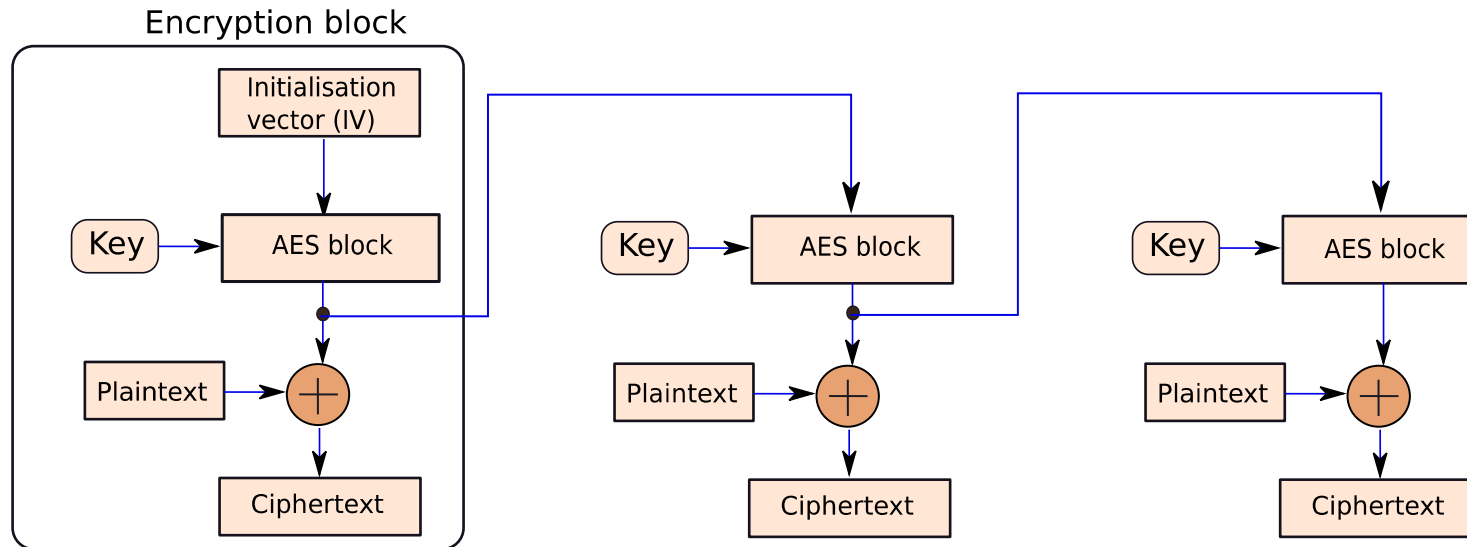


Chain the blocks. This increases the diffusion.



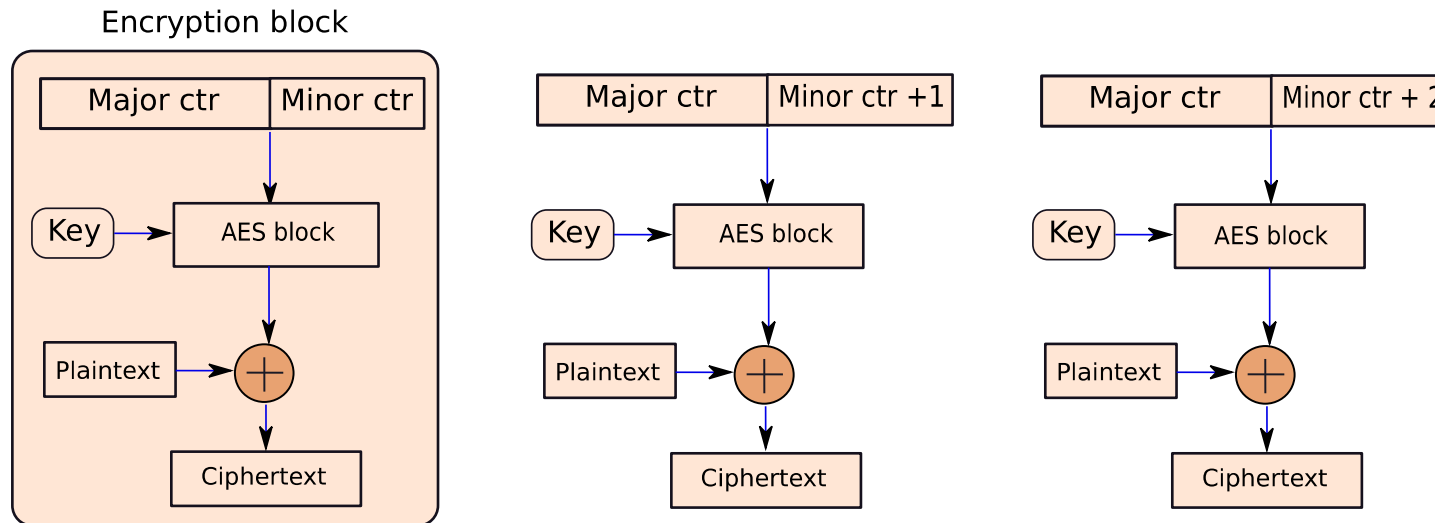
Encryption is sequential, decryption is parallel.
If there is a single 1-bit error in the plaintext, the entire ciphertext is damaged.

Output Feedback Mode



- Errors don't **propagate**
- Error checking of the plaintext and **encryption** can happen in **parallel**
- Both encryption and decryption are **sequential**

Counter Mode Encryption



- Major counter: Specific to an **encryption** task
- Minor counter: **Increases** by one for every block
- The AES block **generates** the OTP (one-time pad)
- This is **XORed** with the plaintext
- **Parallel** encryption/decryption and error tolerance
- **Good** confusion properties and error tolerance

RC4 Stream Cipher

- Generate an **infinite sequence** of pseudorandom numbers based on the input **seed** (similar to the **OTP**)
- Encryption: XOR an **input byte** with the randomly generated number
- **Decryption**: Same as encryption (use the ciphertext byte instead)



- The **S** array is the **key** data structure
- It is **initialized** with a known IV (initialization vector)
- The **length** of the key is between 5 and 16 bytes

The Phases of the RC4 Algorithm

S array



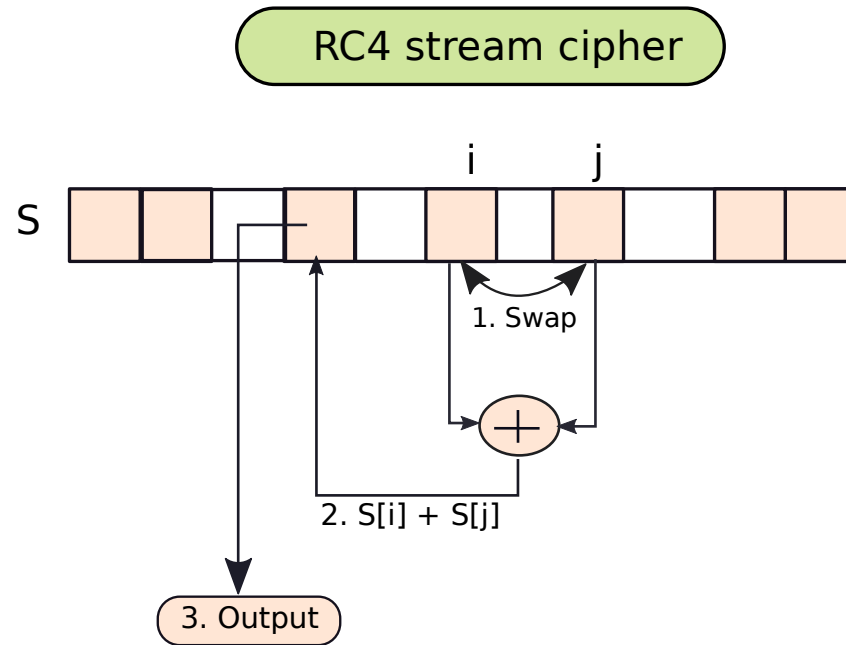
Initialization

```
j=0;  
for (i=0; i <=255; i++) {  
    j = (j + S[i] + key [i% key_length ])  
    %256;  
    swap (S[i],S[j]);  
}
```

Pseudo-random number generation

```
i=0; j=0;  
while (1) {  
    i = (i+1) % 256;  
    j = (j+S[i]) % 256;  
    swap (S[i],S[j]);  
    output (S[(S[i] + S[j]) % 256]) ;  
}
```

Illustration

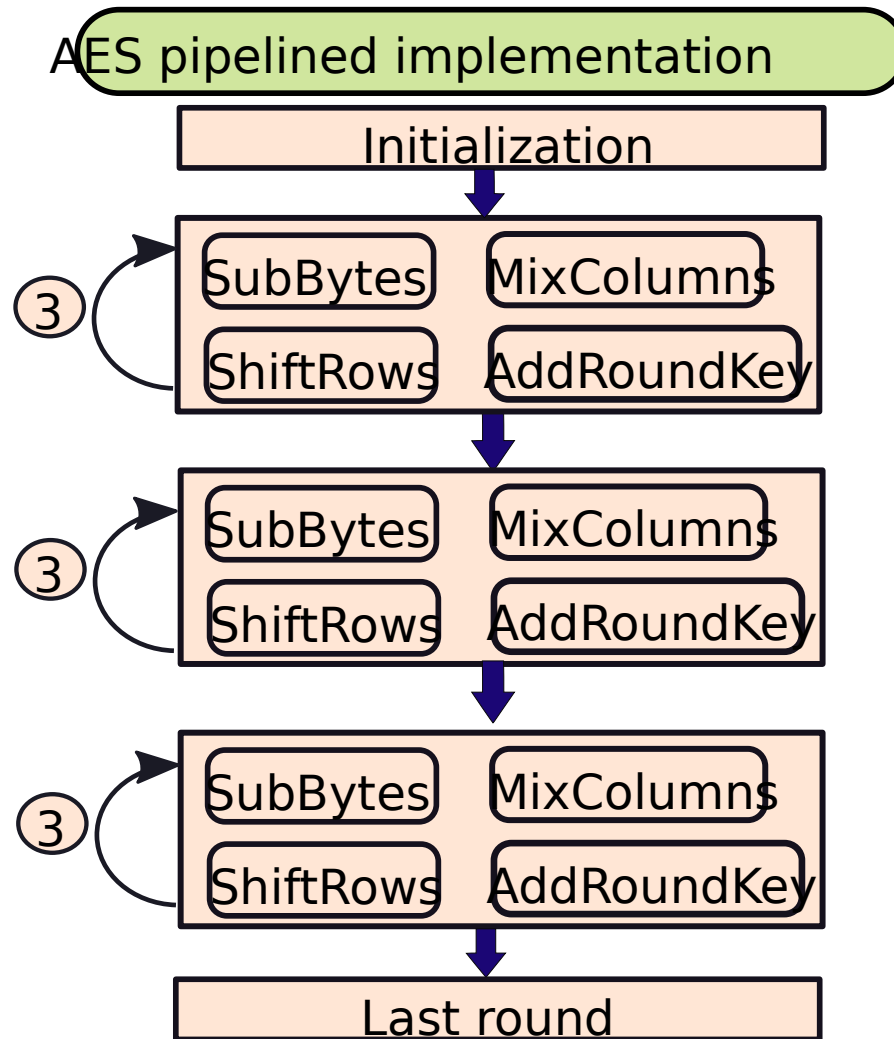


Steps

- Swap
- Add
- Access and output

Hardware Implementation

- It is easy to **pipeline** AES.
- 1st stage – **Initialization**
- 3 **iterations** in one stage
- The **final** stage is the last round
- The RC4 algorithm has **limited** benefits with pipelining



Asymmetric Encryption: RSA

- **Encrypt** with the encryption key (public key)
- **Decrypt** with the decryption key (private key)
- **Convert** the message to a number: m

Example

- $p = 59, q = 67. n = pq = 3953$
- $\lambda(n) = \text{lcm}(p-1, q-1) = \text{lcm}(58, 66) = 1914$
- Choose e such that e and $\lambda(n)$ are **coprime**. Say, $e = 31$.
- **Compute** d such that $(d \cdot e - 1) \% 1914 = 0. d = 247$
- **Public key** = $(n=3953, e=31)$, **private key** = $(n=3953, d=247)$
- **Key** = (n, e, d)

Encryption

$$E(m) = m^e \bmod n$$

Decryption

$$D(m) = m^d \bmod n$$

Properties of RSA

Encryption and decryption are commutative

$$E(D(m)) = m = D(E(m))$$



Encrypt and then decrypt same as decrypt and then encrypt

Digital signatures

Alice encrypts a message with her private key. Bob can use Alice's public key to verify that the message has indeed come from Alice if he already knows the contents of the message.



This message is Alice's digital signature. Nobody else could have sent it.

Digital Signatures



What is the aim of a **signature**?

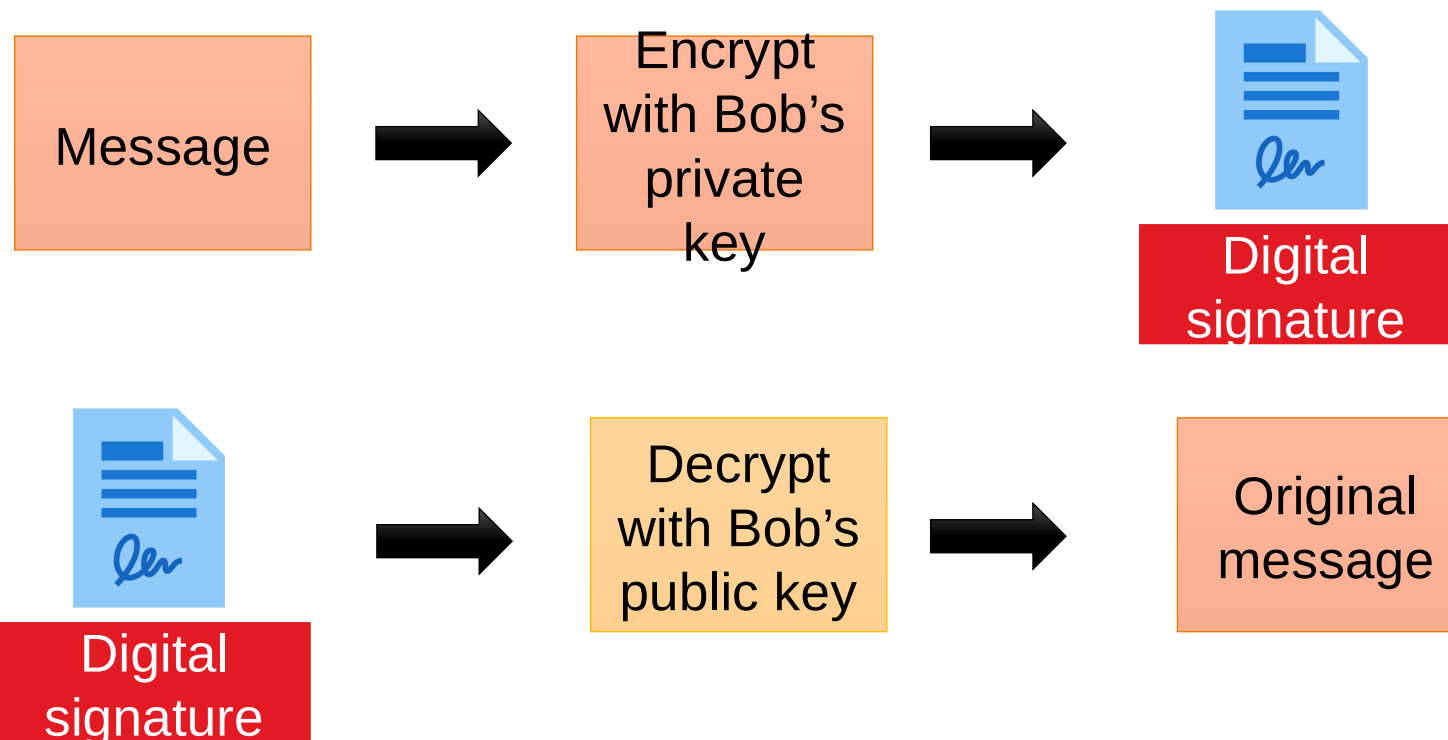
To prove to the world that the **signer** was aware of the contents of the document, and wholeheartedly endorses it.



What are the properties of a **digital** signature?

1. It should be **verifiable**. If Bob has **signed** it, the same can be verified.
2. The signature should be able to cover/protect arbitrarily large pieces of text. It should be clear that Bob has read (processed) the **entire** text (whatever is to be signed).

Digital Signatures – II



The fact that **nobody** other than Bob can have his private key establishes the authenticity and the validity of the **signature**. Note that the contents of the message that is being signed have to be **publicly** known.

Session Keys

Use the slow RSA-based algorithm to decide on a **session key**. Then we use the **faster** AES algorithm (using the **session key**) for encryption/decryption.

Diffie-Hellman Key Exchange

1. **Alice** and **Bob** pre-decide and share two numbers p and q .
2. Alice generates a secret number a . Bob generates a secret number b .
3. Alice computes A and sends A to Bob.
4. Bob computes B and sends B to Alice.
5. Alice computes s .
6. Bob computes s .
7. Both **compute** a key without sending secrets **over** the channel



Outline



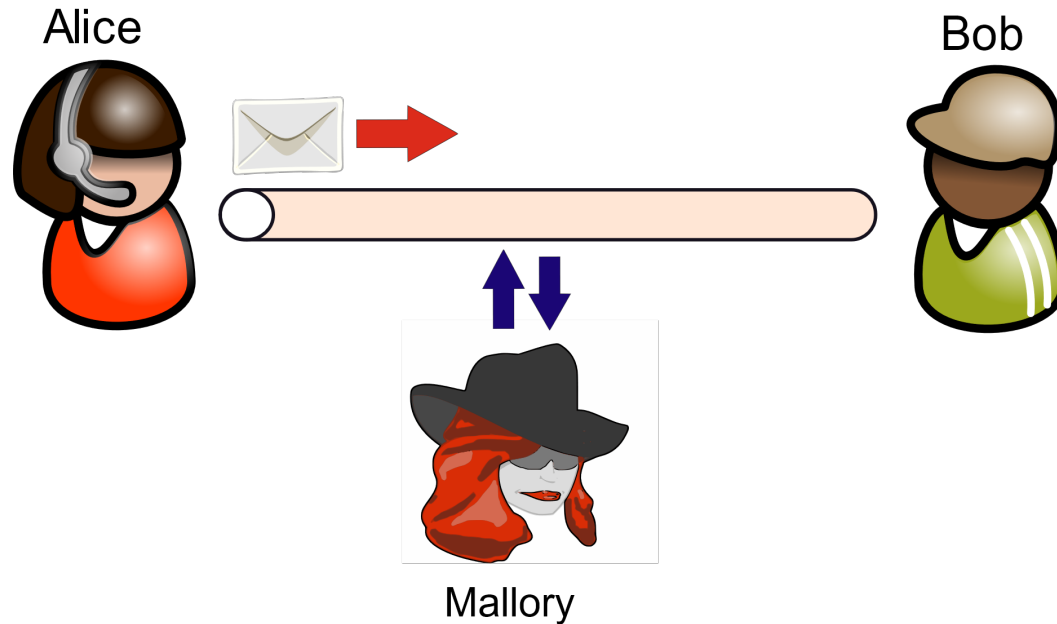
1. Data Encryption

2. Hashing and Data Integrity

3. Secure Architectures

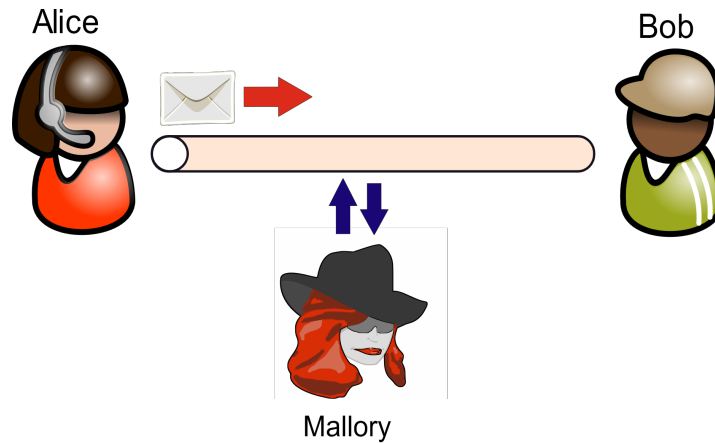
4. Side-Channel Attacks

Common Cryptographic Attacks



- Alice **sends** a **message** to Bob
- Mallory, the hacker, can mount several kinds of **attacks**
- If the message is not **encrypted**, it can simply eavesdrop on the channel and **read** the message.
 - *Solution*: Use encryption

Common Cryptographic Attacks – II



- Man-in-the-middle attack: Mallory **pretends** to be Bob to Alice and **pretends** to be Alice to Bob. Example of a **spoofing** attack.
 - *Solution*: Use digital signatures. They guarantee **authenticity**.
- Mallory can take some part of a **legitimate** message sent from Alice to Carlos and use that in a message sent to Bob.
 - *Solution*: Use separate keys for separate recipients
- Mallory can **capture** and **tamper** with the message (integrity violation)
 - **Use** hashes (discussed next)

SHA-based Hashing

Arbitrarily large
message

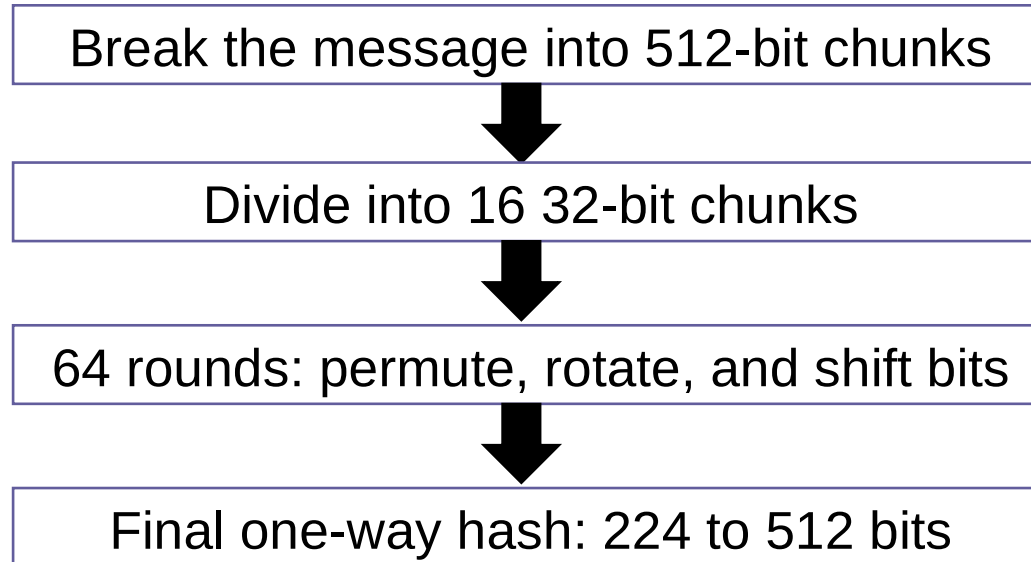


Hash



Unique
512-bit
number

- SHA (Secure Hash Algorithms): SHA1, SHA2, and SHA3



Definition

A MAC (message authentication code) is an **encrypted** hash.
Solves the problem of **integrity** and **authenticity**.

Replay Attacks are still Possible

- Mallory simply **records** a <message,MAC> pair sent from Alice to Bob and **replays** it later.
- Bob will not be able to make out the **difference**.
- Methods to avoid such **replay** attacks
 - **One-time session keys**: Use a **separate** key for every single communication. Counter-mode encryption is ideally suited for this. Keep **incrementing** the minor counter.
 - **Timestamps**: Use timestamps. The sender includes its local time. Reject old messages.
 - **Nonces**: Embed an integer (nonce) in every message. Treat it as a monotonic counter. This can be used to **identify** old messages.

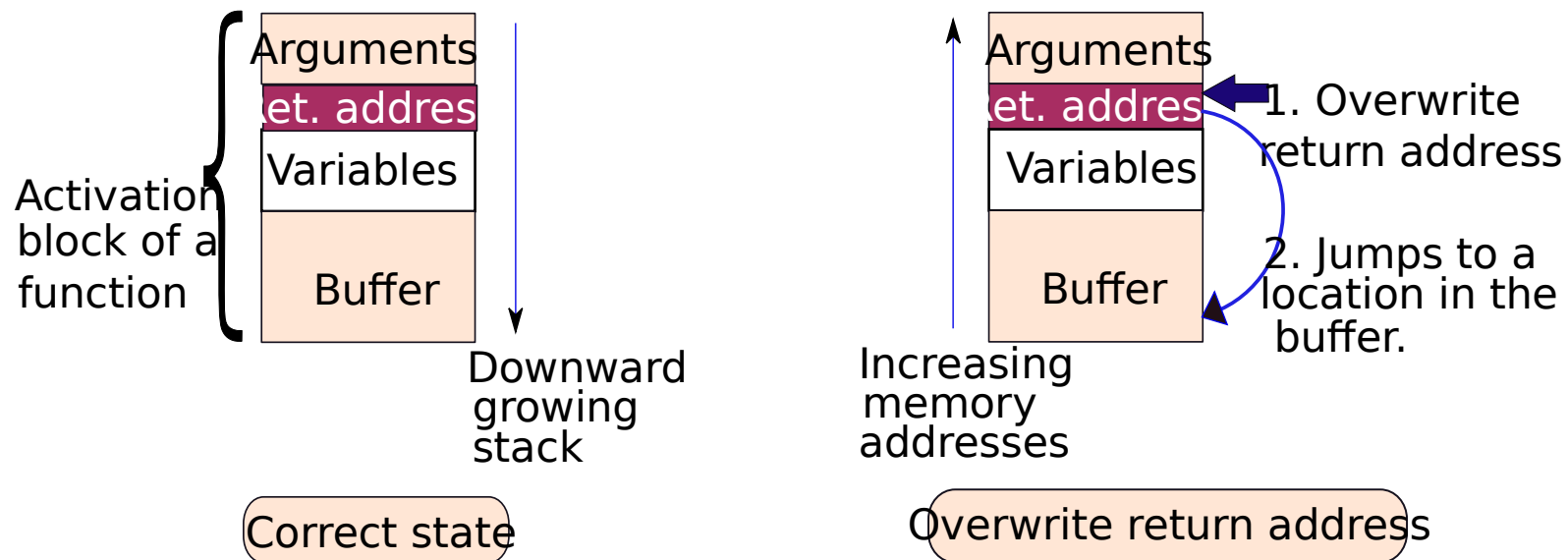
ACIF Properties

Authenticity	{	Establish that the message has come from the sender □ Use digital signatures
Confidentiality	{	Do not allow any intermediary to read the message □ Encrypt the message
Integrity	{	Do not allow message tampering □ Use a MAC
Freshness	{	Do not allow replay attacks □ Use a nonce or counter-mode encryption

Outline

- 
1. Data Encryption
 2. Hashing and Data Integrity
 3. Secure Architectures
 4. Side-Channel Attacks

Traditional Attacks in Software

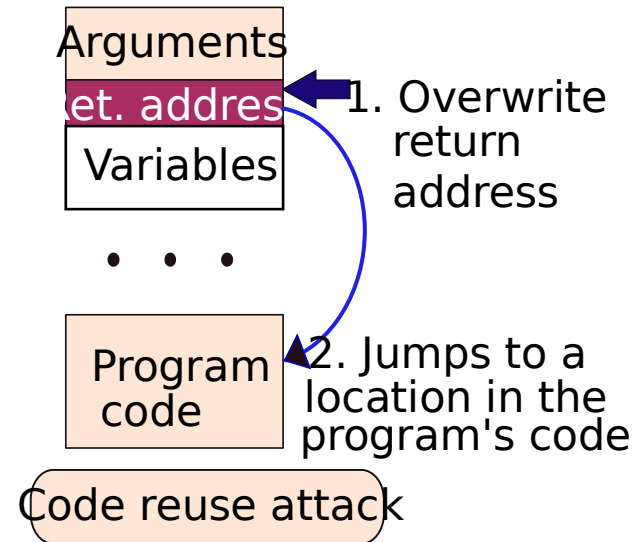


- The return address is typically **stored** on the stack
- It is possible to **access** an array beyond its **bounds** and **overwrite** the return address

```
int buffer[10];  
...  
buffer[20] = <address to be returned to>
```

Code
injection
attack

Code Reuse Attack



- We can make the **control** jump to an **arbitrary** point in the program.
- We can **reuse** existing code.
- To implement **malicious** logic.

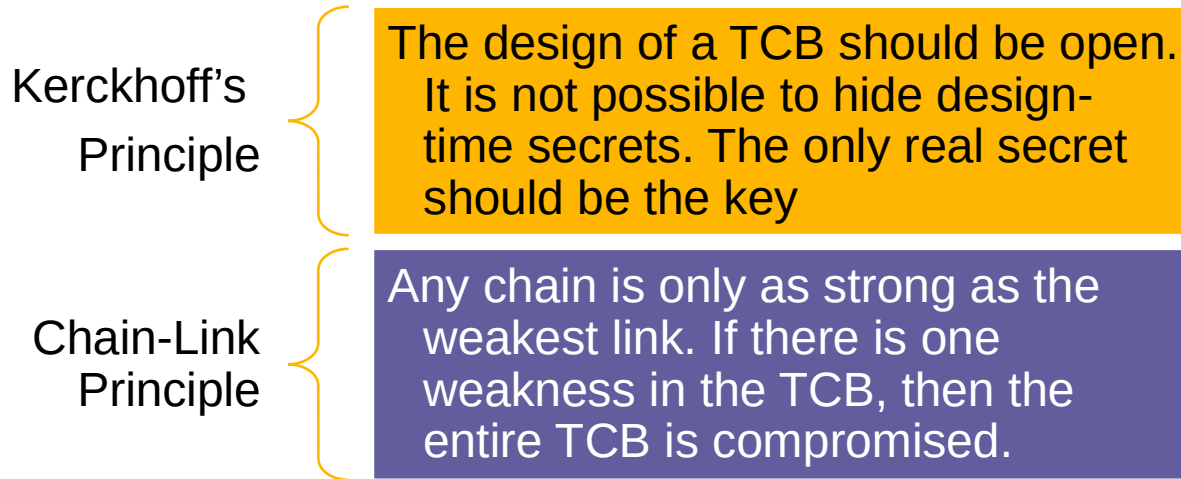
Buffer Overflow Attacks

- They are, by far, the most **common** kind of software-based attacks
- Most **sophisticated attacks** build on this. This is a very effective method of **diverting** the control flow to specific **points** within the program's code and data segments
- Code reuse attacks are quite **powerful**
- They can be **stopped** by **randomizing** the address space ASLR (address space layout randomization)
- **Accesses** to the code, data, DLLs, memory-mapped files, shared memory, etc., are **relative** to a region-specific offset
- This is specified at the time of **program** loading
- Makes it **harder** for the attacker to guess the exact addresses



Hardware Security: Key Concepts

Trusted Computing Base (TCB) \sqsubset Set of hardware/software components that are assumed to be secure.



Attack Surface \sqsubset Set of all the **attacks** that can be mounted on a TCB.

Threat Model \sqsubset **Subset** of the attack surface. This is the set of all the attacks that the designer **expects** to be mounted.

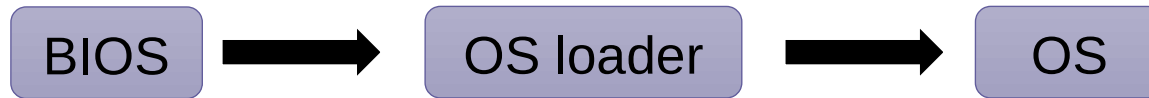
Some More Definitions

Trusted Execution Environment (TEE) \sqsubseteq It is a hardware environment (primarily) that allows the user to create and run a **secure process** known as an **enclave**.

Root of Trust (RoT) \sqsubseteq This is some module that is completely trusted (cannot be compromised). It typically provides the following services:

- A dedicated piece of hardware known as the **Trusted Platform Manager** (TPM) verifies the **BIOS** and initiates a **secure** boot process.
- Provides **cryptographic services** \sqsubseteq encryption, digital signatures, attestation

Notion of Measurements



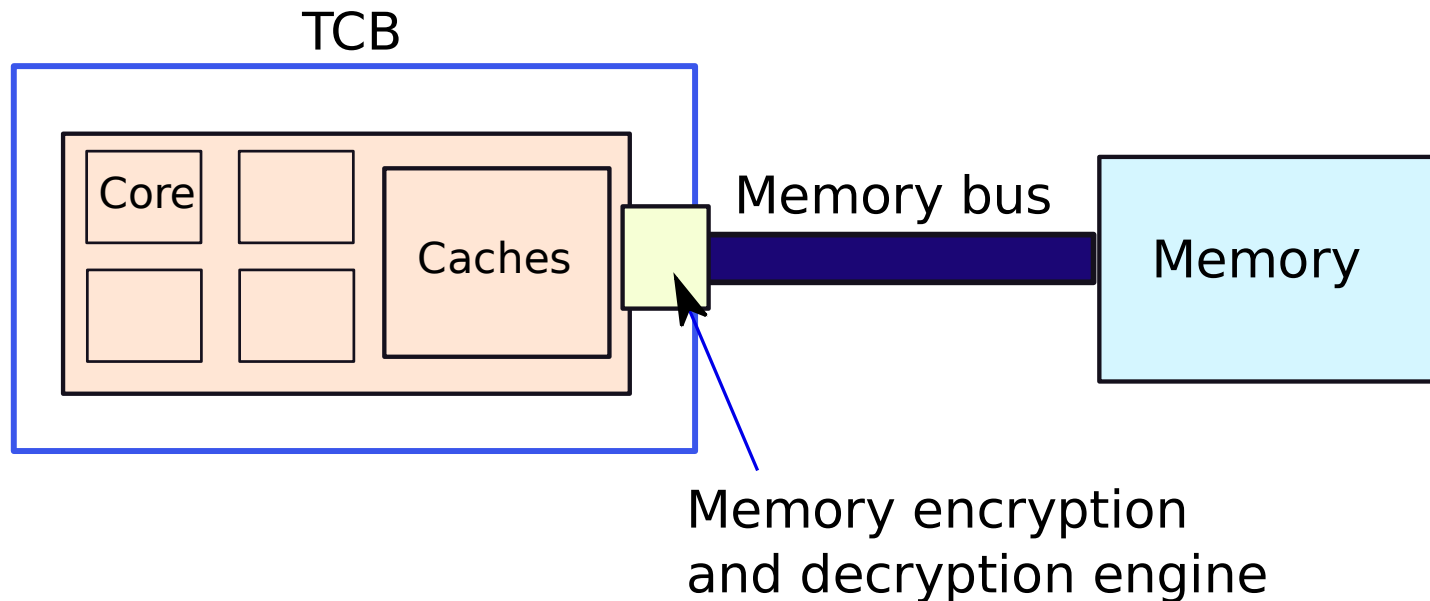
- Compute a **hash** of the BIOS code, OS loader, and the OS code. This is done by the **RoT** and is known as a **measurement**.
- The measurement can be **digitally signed** and sent to a **remote** server. It can **verify** the signature and assess the **validity** of the **measurement**. This is known as **attestation**.
- Sealing \Rightarrow Tie the code with the data. **Encrypt** the data with a key derived from the **measurement**.



Use all these concepts to create a secure processor that provides the ACIF guarantees.



Design of a Secure Processor (inspired by Intel SGX)



- Assume that the **chip** is **secure** – cache banks and cores
- However, the main memory is **vulnerable** – all of its contents can be **read** and **modified** at will. The memory can simply be **physically removed** and its contents can be **accessed/modified**.

Outline of a Solution

- Confidentiality \Rightarrow Encrypt all the data
- Integrity \Rightarrow Compute a hash of a cache line and store it somewhere
- Authenticity \Rightarrow Encrypt the hash with a secret key
- Freshness \Rightarrow Encrypt the same block with a different key every time.

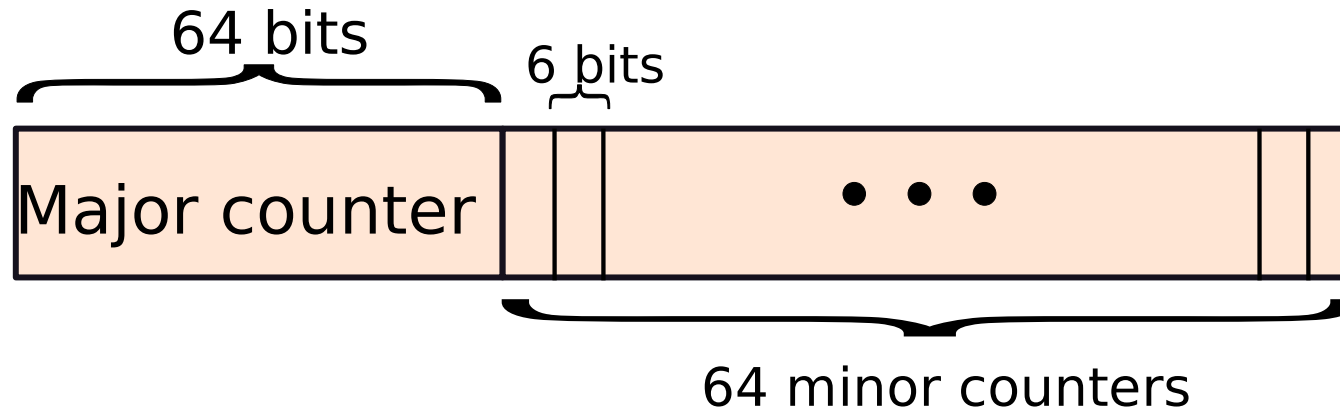


One more reason for encrypting the same data with a different key every time is \Rightarrow an attacker can guess that the same data is being sent if the key is the same. This reveals some important information. We have to stop this.



The main focus is on LLC (last level cache) misses. These cross the secure – nonsecure boundary.

Use Counter-Mode Encryption

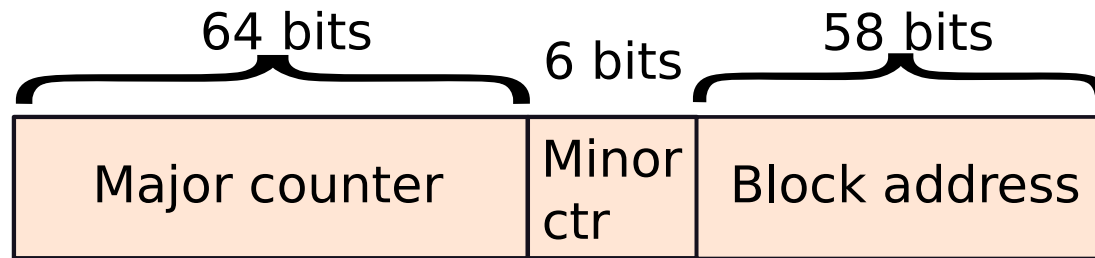


- Consider a 4 KB page. It has 64 **cache blocks** (each **block** is 64 bytes)
- We associate a 64-bit major counter with a **page**
- Each of the **blocks** is associated with a 6-bit minor counter
- Size of each **cache line**: 64 bytes (512 bits)

$$64 \text{ bits} + 64 * 6 \text{ bits} = 448 \text{ bits}$$

64 bits left

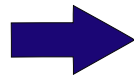
Process of Encryption/Decryption



Processor specific
unique number

PUF
Rnd. number
Enclave id

Generated at boot
time



Generate MAC

Simultaneously
compute the 128-
bit (16 byte) MAC

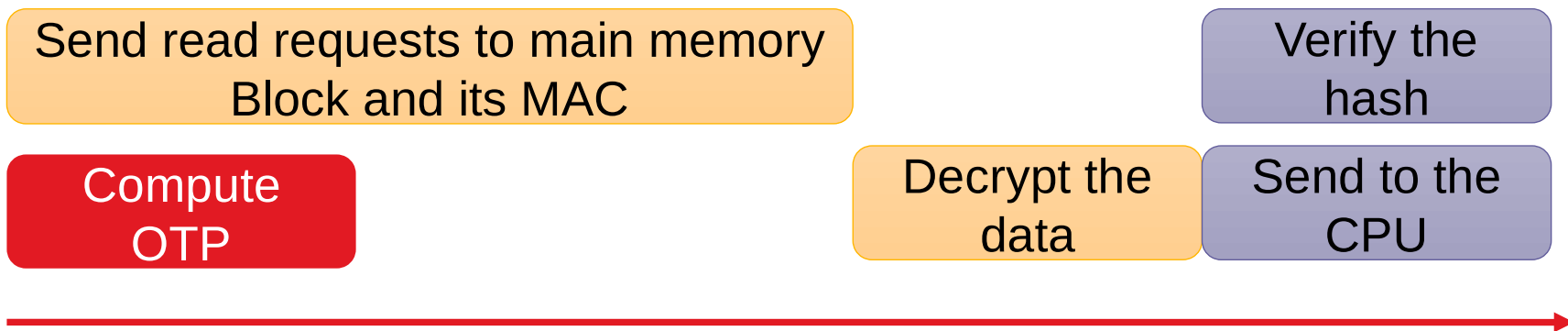


Increment the minor counter after every encryption

The **encryption** is **unique** to the processor, boot time, enclave id, block address and values of the counters (no **chance** of repeats)

Read Operation

If we find a **block** in the caches, we simply **read** it. If there is a **cache miss** in the **LLC** (last level cache), the following needs to be done.



- **Computing** the OTP is not on the **critical path** (shadow of a **miss**)
- **Verifying** the hash is off the critical path. If the hash does not **match**, it is a catastrophic **failure** – the security has been **breached**.

Writes and Evicts

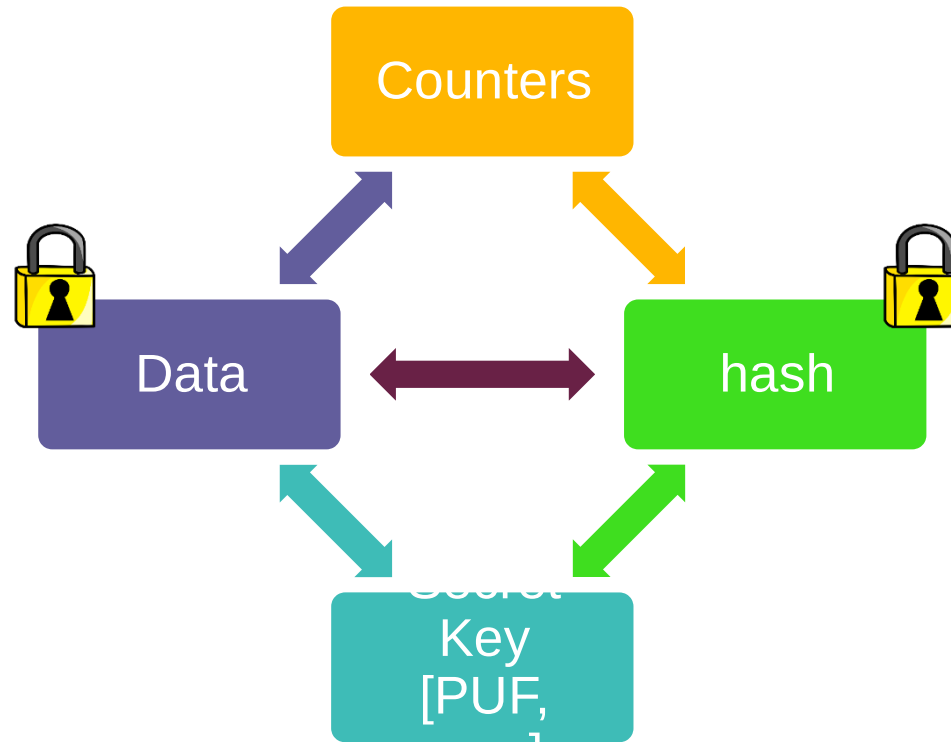
1. Whenever we **evict** a modified line from the **LLC**, we **write** it to secure memory.
2. **Increment** the minor counter, compute the OTP, encrypt the block.
3. Also, **compute** a MAC (hash + encryption). Use the same OTP.
4. Write the **encrypted** data and the MAC to secure memory.


If a minor counter overflows ...



1. **Increment** the major counter.
2. **Reset** all the minor counters for the page
3. **Re-encrypt** all the blocks in the page that are in main memory. (The counters have changed)

ACIF Properties



1. We cannot **change** all four. The **<PUF, rnd num>** combination is not **known** and cannot be **changed**.
2. We cannot **change** just the data and the hash \Rightarrow we will not be able to correctly **encrypt** them
3. We can of course **replace** the **<counter,data,MAC>** triplet with a valid triplet. The system should somehow **prohibit** this 

Integrity of Data



Data



MAC

Hash

C • Encryption ensures confidentiality

I • It is not possible to tamper with the data or MAC and remain undetected. This guarantees integrity. Can we tamper both?

- Can we replace the encrypted $\langle \text{data}, \text{hash} \rangle$ pair with another valid pair? The question is from the modified data can we figure out the new MAC. We need to correctly encrypt them, which is not possible. We don't know the PUF and random number (even if we can guess the counters).

Authenticity and Freshness

F

- ? How can we prevent replay attacks?
Replacing <counter,data,hash> with a **valid** triplet (seen in the past)

Answer: Protect the counters. Do not allow them to be maliciously **modified**. Just the **<encrypted data, MAC>** cannot be replayed. The counters would have changed, and **integrity** check will **fail**. We cannot fetch data for another **address** either.

A

- ? How does the processor know whether it is reading back the same data that it wrote?

Answer: The values of the **counters** for the block are the same. It just **decrypts** the block contents and MAC with the **same** counters. If the hash **matches**, we are sure that the processor is reading back what it had **written** earlier. Implicitly: the PUF, enclave id, and rnd num are the same.

Moral of the Story

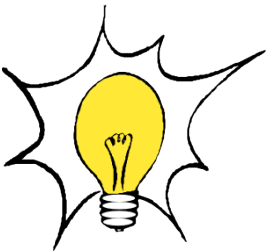
- Just **protect** the counters. The data is **secure**.
- All ACIF **guarantees** are provided.



The counters are much **smaller** as compared to the data.
They are easy to **store** and **protect**.

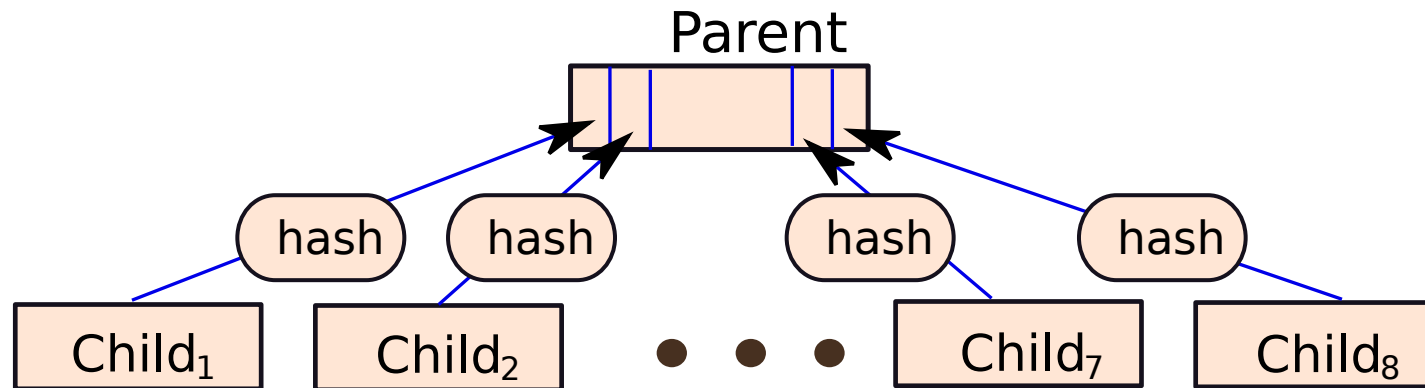
Where do we store the counters?

- There is a **dedicated** counter cache that is **stored** in the regular caches (software managed) or we have special **hardware** for it on chip
- If a **counter** is found, we take it from the **counter cache**.
- However, there may be **evictions**. Counters may need to be written to **main memory**.
- A mechanism is required to **protect** them in main memory. In this case, counters need to be themselves **encrypted**.
- We need counters for **encrypting** counters !!!
- What about those counters, we need another set of counters



We need a **recursive data structure** to store counters and provide **ACIF** guarantees.

Simple Tree-based Method for Storing Counters



Solution: Use a Merkle tree \Rightarrow Parent stores the hashes of its children.

- The leaf is the **set** of all the counters for a **page** (448 bits)
- An internal node contains the **hashes** of its children.
- For a **read** (check the hash) at the **parent**; for a **write**, cascade updates towards the root
- The **root** of the Merkle tree represents the **state** of the entire tree
- If a counter can be maliciously **modified**, we can always check its hash
- **One optimization:** The process stops at a cached internal node. No need to cascade updates till the root all the time.

Criticism of this Approach



Why does the **process** stop at a cached internal node?

Answer:

All cached data (within the processor) is deemed to be correct (it is within the TCB).



What are the problems with this approach?

Answer:

An **internal node** can have only 8 **children**. Assume it has a **size** of 64 bytes, and each **hash** is 64 bits (8 bytes). There will be too many levels.

Increase the arity
of the tree

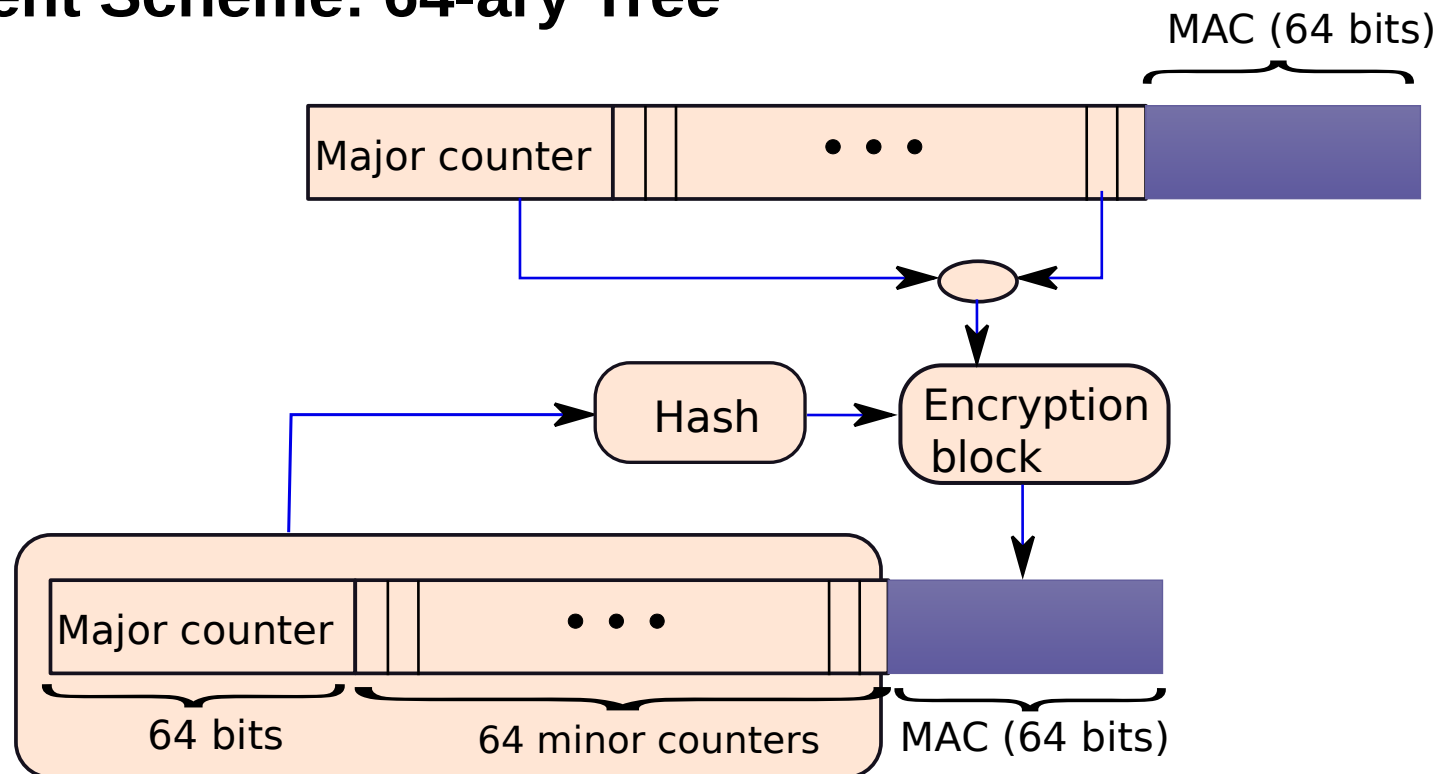


Reduce the
number of levels



Improve
performance

Efficient Scheme: 64-ary Tree



- Every **node** has 64 children.
- Contains one **major** counter and 64 **minor** counters
- The MAC is a **part** of the node (encrypted hash of its 448-bit contents)
- The node is conceptually connected to its **parent** — the counters to **create** the MAC come from the **parent**.
- Every **eviction increments** the minor counter at its parent.

Continued ...

- All the **nodes** have a similar **structure**
- 64-bit major counter + $64 * 6$ -bit minor counters + 64-bit MAC
- The parent and child are **connected** via the MAC
- This means that every **leaf** (counters used to encrypt data) is connected to the **root** of the tree
- The **root** is always stored in the TCB (assumed to be correct)
- A 64-ary tree **reduces** the number of **cascaded** levels, amount of **data** that needs to be stored on-chip, and **read/write** times
- We can have a **dedicated** counter cache within the chip or just use the **regular** caches

512 bits



Creating and Managing Enclaves

Boot time

Processor Reserved Memory (PRM)

Partition the physical memory space.




We can further **partition** the reserved memory: metadata and **secure** data.
Intel SGX v1: Total **size** of the PRM (128 MB), **usable** space (92 MB)



New instructions added to the ISA

Setting up an Enclave

- ECREATE **instruction**:
 - Create the **data structures** for an enclave
 - **Partition** the physical and virtual address space between the secure and nonsecure regions
-  The OS **manages** the page tables and TLBs, even though it is **not** trusted.
- EADD **instruction**:
 - Adds code and data to the secure enclave
 - **Copies** data into secure pages (within the enclave)
- EINIT instruction:
 - Compute a measurement and digitally sign it
 - May use an additional **quoting** enclave to **sign** the enclave and send it for **remote** verification
 - **Launch** the enclave

Memory Management

- The OS **manages** the page tables and TLB entry
- It can deliberately **induce** TLB misses and page faults.
 - This can **provide** the page address **sequence**
 - The hardware **zeros** out the bits within the **page offset** (address that caused the TLB **miss** or page fault)
 - Nonetheless, some information **leaks**



❓ How do we **stop** the OS from **redirecting** writes to the nonsecure region? Can be easily achieved by creating a malicious TLB mapping.

Answer: In secure memory (**metadata** region), we maintain an **inverted page table (IPT)**. Physical frame num \Leftarrow virtual page num

Managing Memory – II

- After an EADD **instruction** an entry in the IPT is created
- Before the enclave is **launched**, check whether all the mappings in the IPT are **valid**
 - A secure **page** should never be pointing to a **nonsecure** memory region or vice-versa
- All the **updates** to the TLB are **monitored**
 - No **illegal** mapping should be created
 - The IPT is always deemed to be **correct**
- The OS **maintains** the page table
 - This is **fine** because all TLB updates are **monitored**
 - It can **swap** out pages to the nonsecure memory or the disk

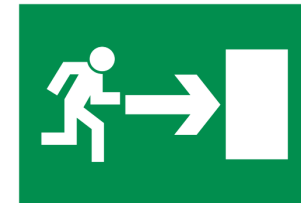


Provide
ACIF



Zero out page offset bits for secure pages

Interrupts and Enclave Entry/Exit

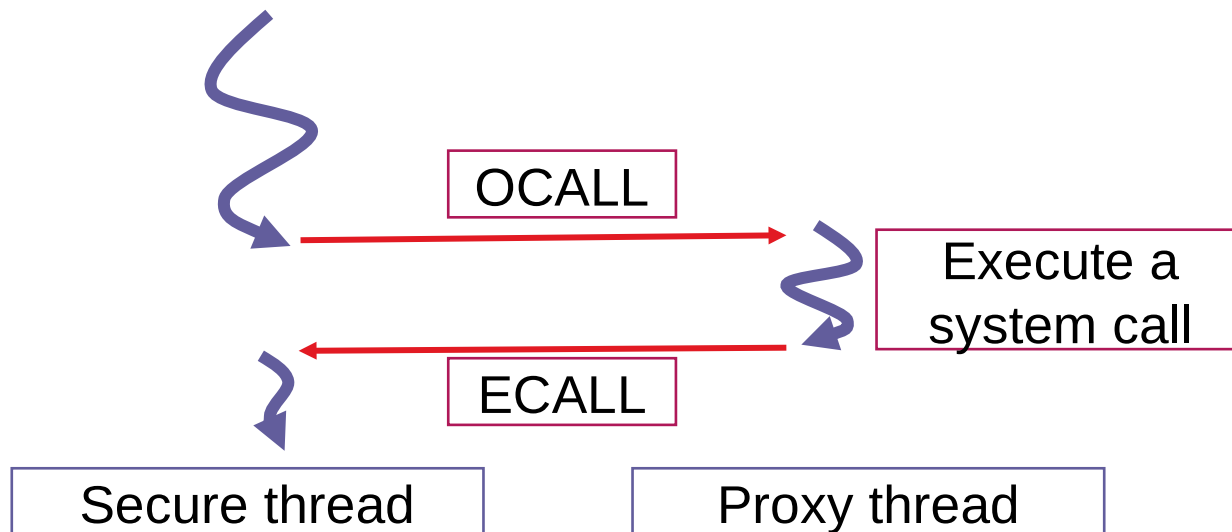


- Asynchronous Enclave Exit
 - An **interrupt** maybe delivered to a core that is **running** an enclave
 - The **secure context** needs to be **stored** in the secure memory region
 - **Flush** the TLB \Rightarrow the **secure** area should henceforth not be **accessible**
- Enclave Enter (EENTER)
 - An **application** must call a **secure** function (ECALL)
 - The OS **treats** this as a **regular** context switch
 - The **secure** mode gets turned on
- Enclave Exit (EEXIT)
 - The secure code calls a **nonsecure** function (OCALL)
 - All arguments are **copied**. The TLB is **flushed**.

Enclave Removal and System Calls

- The EREMOVE **instruction** is invoked to tear down an enclave.
- **Clear** all the state and pages in the memory and disk

❗ We **cannot** invoke system calls from secure mode. The OS will be able to **see** the state of the secure program, which is not **allowed**.



Oblivious RAM



We can see the **stream** of **addresses**. This can **leak** secrets.



The **attacker** should not be able to see the sequence of addresses.

Two strategies

1

Keep **migrating memory** words to new **locations**. Make it harder to guess by such **random permutations**.

2

Access more **locations** than needed. Give the attacker **redundant** information.

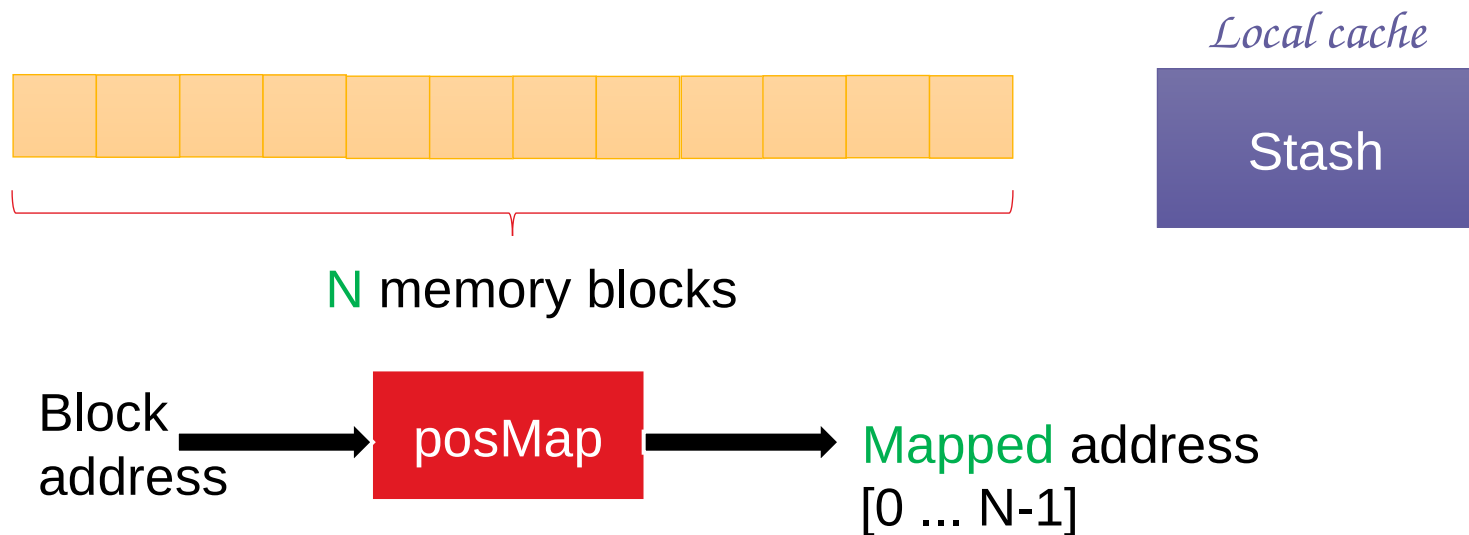
Oblivious RAM – II



Naive solution: For every **memory** access, just **access** all the main memory **locations**. Choose the **correct** value (for a **read** within the CPU).

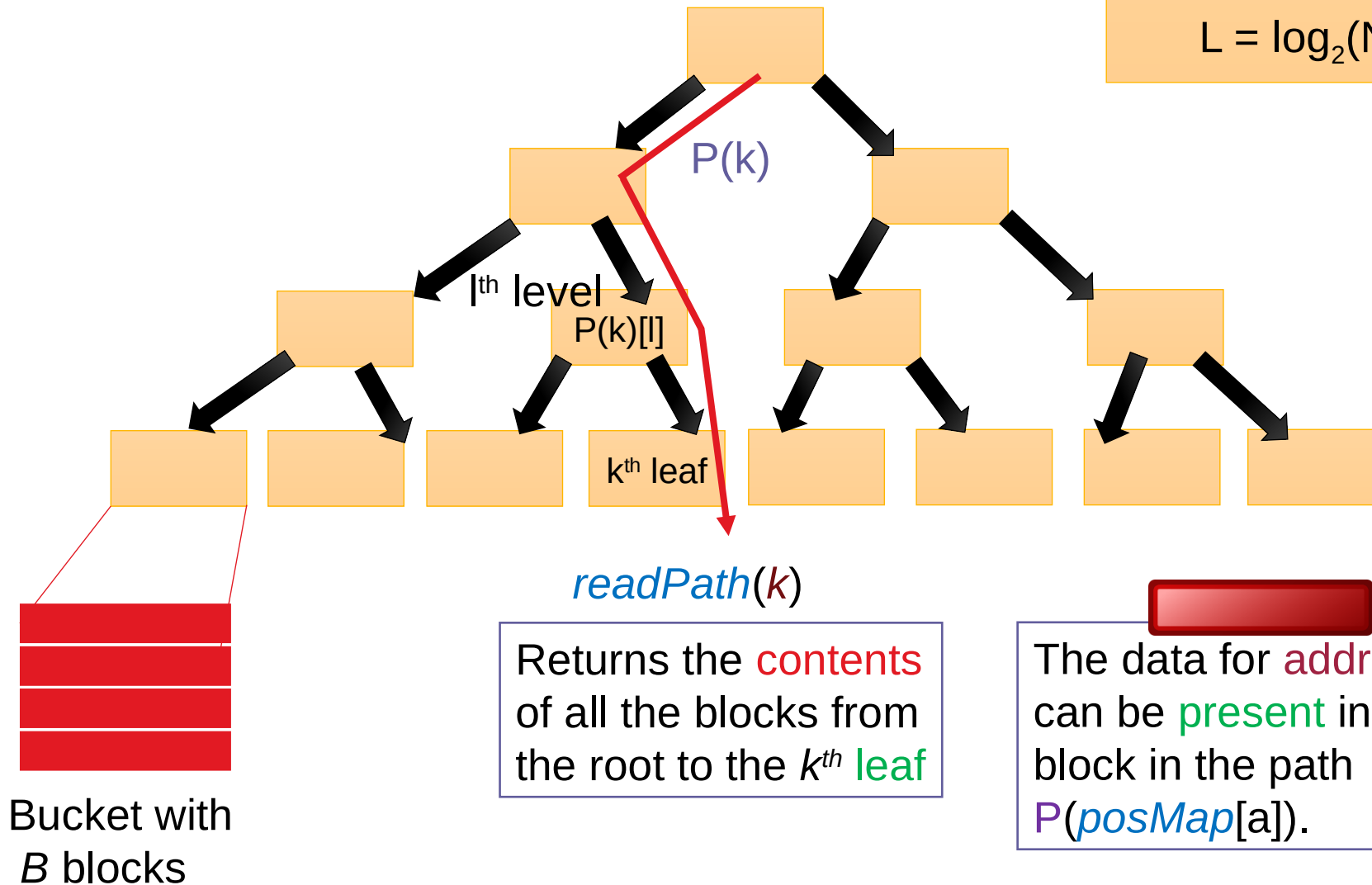
Oblivious RAM: Use some **degree** of **permutation** and **redundancy**.

Path ORAM: Much more **efficient** version of it.



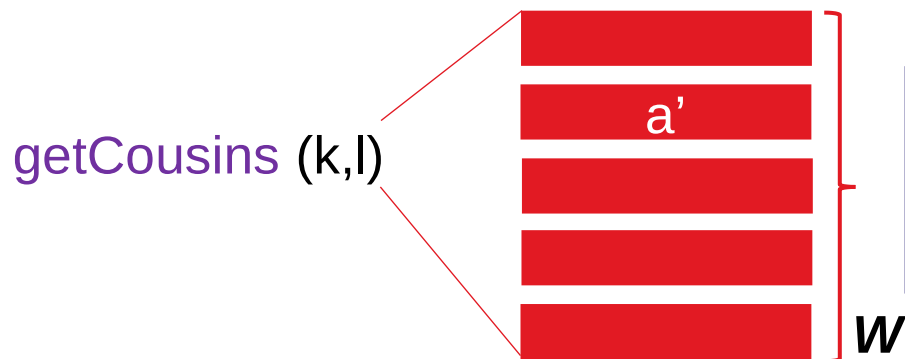
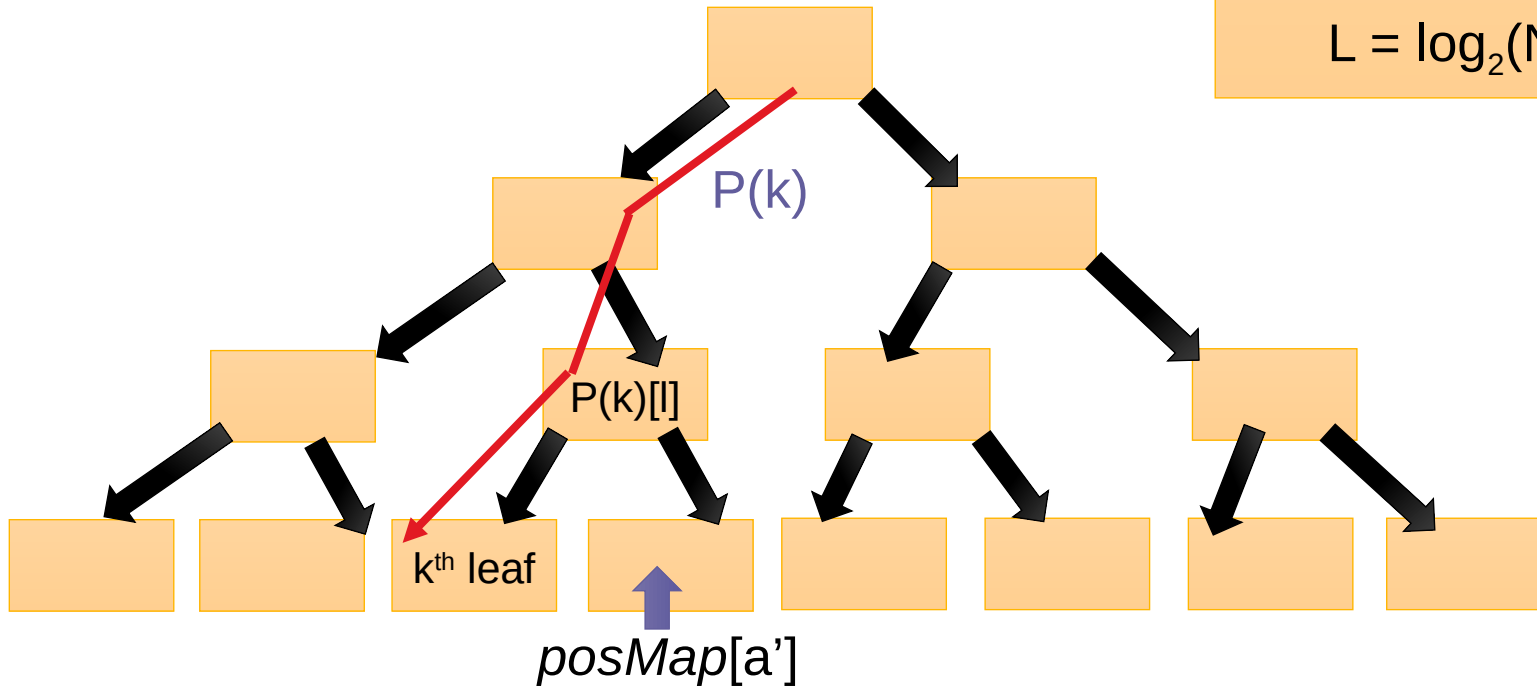
Tree Stored in Main Memory

Tree with 2^L leaves and $L+1$ levels
 $L = \log_2(N)$



Few More Definitions

Tree with 2^L leaves and $L+1$ levels
 $L = \log_2(N)$



Nodes share a common node
 and present in the stash

$$\forall a' \in W, P(k)[l] = P(\text{posMap}[a'])[l]$$

$$S[a'] \neq \phi$$

Path ORAM Algorithm

```
/* op is the operation , a is the address , new_data is the  
data to be written (if op == write ) */
```

```
function access (op, a, new_data):
```

```
/* Read the mapping of the address from posMap and compute  
the new random position */
```

```
pos = posMap [a]
```

```
posMap [a] = random (0 ... (N -1))
```

```
/* Add all the blocks on the path (pos to root ) to the stash S */
```

```
S = S readPath (pos)
```

```
/* Implement the read or write . For a write add the  
<address ,data >  
to the stash . */
```

```
old_data = S[a]
```

```
/* output of the read */
```

```
if (op == write ) S[a] = new_data
```

Path ORAM Algorithm – II

Permute Operation

```
for {  
    W = getCousins (pos , l) /* get all cousin nodes that  
                               are in the stash */  
    W = trim (W)             /* |W| = B (block size) */  
    S = S - W                /* remove W from S */  
  
    /* Add all the addresses in W to the bucket at level  
    l in P(pos), along with their  
    data */  
    P(pos)[l].add (a', S[a'])  
}  
    Add all cousins to the bucket  
    at P(pos)[l]
```

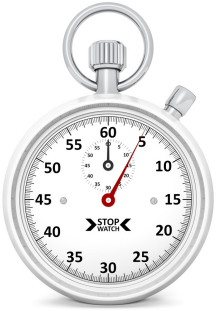


Read the path \Rightarrow Remove the cousins from the stash \Rightarrow Add them to the tree at possibly a new location

Outline

- 
1. Data Encryption
 2. Hashing and Data Integrity
 3. Secure Architectures
 4. Side-Channel Attacks

Side Channel Attack



Modern **processors** have a **nanosecond-level high-resolution** timer.

An L2 **miss** **makes** the processor **stall** for hundreds of cycles. An L1 or i-cache **miss** induces a **smaller** penalty, which may be **detectable**.

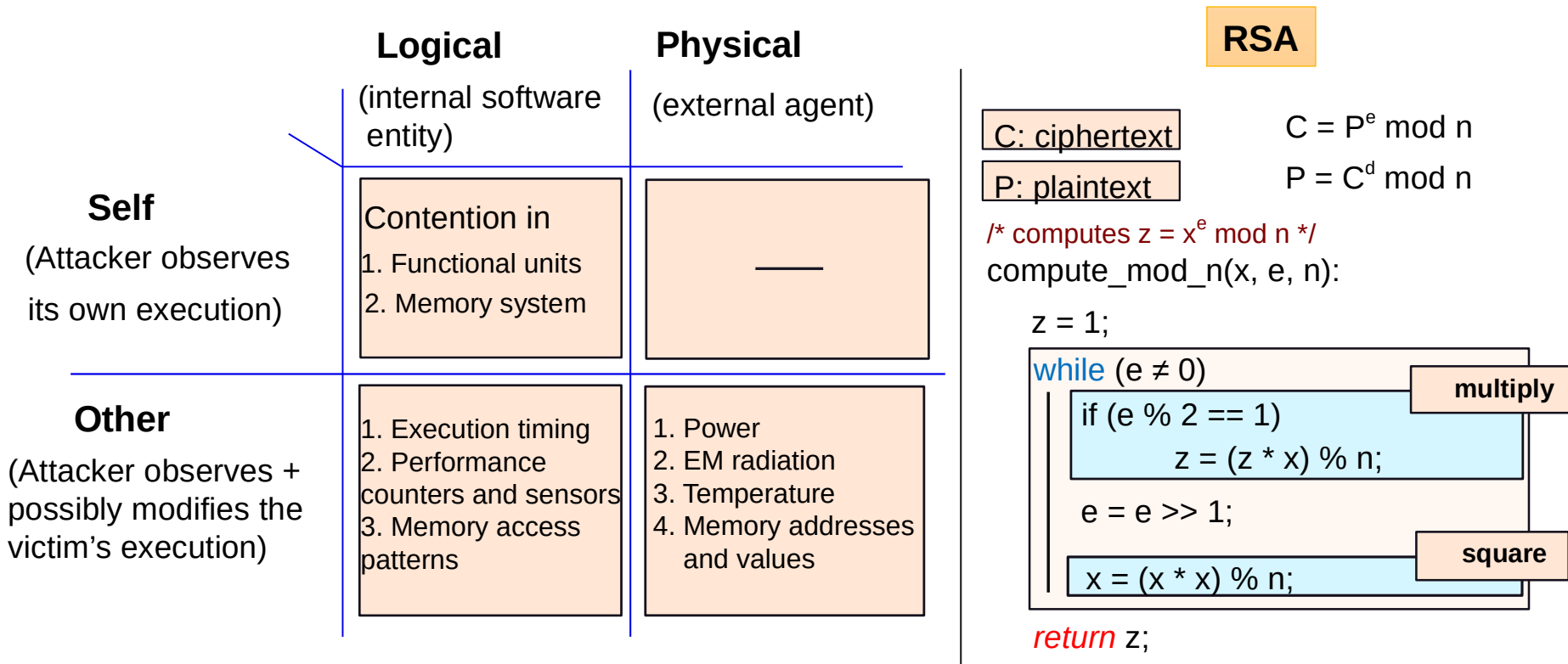


Is it **possible** for a **process** to look at its **cache** hit/miss behaviour and figure out the **memory address** sequence of another **processor**?

Answer: **Yes**, by measuring the nature of destructive interference.

There is information leakage through a side channel

Space of Side-Channel Attacks



- Whenever some **artefact** of program **execution** is measurable, we have a **side channel**. Some information can **leak** out.
- See the **case** of RSA, where we have a **data-dependent** i-cache access.

Prime+Probe Attack

1. The **attacker** thread **accesses** all the lines in the cache (Prime phase)
2. **Yield** the processor to the **victim** process
3. The victim thread **executes**. If it needs **access** to a cache block (**code** or **data**), it will have a **miss** and fetch it from the **lower level**
4. The attacker **resumes** control. It **measures** the time required to **access** every cache line. Based on the measured times, it gets an idea of the memory access behavior of the **victim** process.

Example

RSA has a **data-dependent** i-cache access (**modular multiply function**). We can **track** if this instruction was **fetched** or not. This will **tell** us about the i^{th} bit in the **key** !!!

Flush+Reload Attack

Assume code pages are shared between the attacker and victim threads/processes

1. The **attacker** first **flushes** a given set of **cache** lines.
2. Allow the **victim** to run for some **time**.
3. The **attacker** accesses the **same** set of lines again.
4. Check if they are **back**.



Same idea as the Prime+Probe **attack**. Get an **idea** of the **memory** addresses **accessed** by the **victim** in critical code **regions**.

Other Side Channels



Basically, whenever there is **contention** and the same is **measurable**, we have a side channel

- Functional units
- Architectural structures
- Even **DRAM** rows



- Read disturbance \Rightarrow Reading a row repeatedly can **disturb** the contents of the adjoining rows
 - The **attacker** can deliberately **perturb** a nearby row.
 - After the victim **accesses** its row a **sufficient** number of times, the attacker's **row** will have a bit flip
 - This can be **detected** with an HRT
 - The **address** accessed by the attacker can be **detected**

Evict + Time Attack

1. The **attacker** deliberately **evicts** some cache lines (carefully crafted accesses)
 2. It **runs** the victim
 3. It carefully **measures** the time the victim task took
 4. If there have been cache **misses**, the **victim** will take more time.
 5. The **address** sequence will thus be **visible**.
-
-  The **attacker** can also take a look at **performance** counter and **page fault** data (in the case of an **OS**).
 -  Many other **sources** of information **leakage** are possible: **power** consumption, EM **radiation**, **temperature**, etc.

Transient Execution Attacks



● Consider this piece of code

```
if ( val < threshold )  
    v = array1 [array2  
[x]];
```

1. Consider an OOO processor with branch prediction
2. It **speculatively** accesses the address:
 $\text{array1_base} + \text{array2}[x] * 4;$
3. Later on, the **pipeline** is flushed.
4. This **address** can be detected using a Prime+Probe attack
5. We will get to know **array2[x]** if we know **array1_base**
6. If we can control **x**, we have **access** to the entire **memory** !!!

Countermeasures

- Strictly **partition** the **resources** such as cache sets between **threads**. This will eliminate interference.
 - Can lead to a **suboptimal** use of resources
- Deliberately add **noise**
 - Again, can cause **slowdowns**
- Turn **off** or **decrease** the resolution of the **timer**
 - Some network and gaming applications may **cease** to work

Conclusion

We can encrypt data using either block ciphers or stream ciphers. AES is the most common block cipher.

For encrypting a block of data, counter mode encryption is a very useful method.

Any secure architecture has to provide the ACIF guarantees.
ACIF \equiv Authenticity, Confidentiality, Integrity, and Freshness

A secure architecture just needs to store the counters securely. A Merkle tree of counters is typically used.

Modern processors are prone to side-channel attacks. Whenever there is contention, some information leaks.



The End