# 13 Secure Processor Architectures

Up till now we have assumed that all the components that we use are trustworthy. Of course, they can develop faults and they can fail; however, they will not deliberately introduce faults, or leak sensitive data to unauthorised outsiders. Sadly, these assumptions are not valid in today's complex world. Modern computing systems have to be designed to be immune to various kinds of attacks. Otherwise, it is possible that sensitive data such as passwords and credit card numbers can be stolen and used by malicious adversaries. Hence, gradually security is becoming an important criteria in designing processors and computing systems in general.

In this chapter, we shall primarily look at two important aspects of security namely *confidentiality* and *integrity*. The former ensures that data is encrypted and is not visible to outsiders. The latter ensures that it is not possible to maliciously change the contents of data or instructions, and remain undetected. First, we shall discuss the cryptographic primitives that are necessary to enforce these properties. Then, we shall design architectures that use basic cryptographic hardware and protect different aspects of the processor and the memory system.

**Definition 100**

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

**Integrity** *The data has not been tampered with.*

## 13.1 Data Encryption

Given a piece of text known as the *plaintext*, the aim is to encrypt it to an encrypted form. The resulting text is known as the *ciphertext*, which typically has the same size as the original plaintext. Any algorithm that encrypts data is known as an *encryption* algorithm. In general, the algorithm is not kept hidden, instead we protect the key, which is a small 128-256 bit number that determines the result of the encryption. All encryption algorithms rely on such a *secret key*, and may also use multiple keys for encryption and decryption respectively.

We can divide the space of encryption algorithms into two types: block ciphers and stream ciphers. Block ciphers encrypt data block by block, where a block is between 128 bits and 1024 bits, and stream ciphers encrypt data at smaller granularities. It is typically possible to parallelise the implementation of block ciphers, however, stream ciphers have a predominantly sequential character.

There are two properties that such ciphers should normally satisfy namely *confusion* and *diffusion* (see Definition 101).

**Definition 101**

**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.*

Many of the algorithms have interesting confusion-diffusion trade-offs, which we shall study in this section.

### 13.1.1 AES Block Cipher

The most popular block cipher algorithm as of 2020 is the AES algorithm (Advanced Encryption Standard). The AES algorithm typically has a block size of 128 bits, and it uses three different key sizes (depending upon the degree of security that is required): 128, 192, or 256 bits. The AES algorithm is divided into a sequence of rounds, where in each round we perform simple substitution and permutation operations. The number of rounds depends on the key size as shown in Table 13.1.

| Key size | Rounds |
|---|---|
| 128 | 10 |
| 192 | 12 |
| 256 | 14 |

Table 13.1: AES key sizes and rounds

Consider a sequence of 128 bits or 16 bytes of plaintext. Let us number the bytes as $B_0, B_1 \ldots B_{15}$. We can represent the 16 bytes as a $4 \times 4$ matrix of byte-sized blocks as shown below.

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

This matrix is known as the *state* of the algorithm. The state is initially set equal to a block of plaintext. Subsequently, in each round we perform a subset of the following operations. They modify this matrix. After all the rounds, this matrix contains the bytes of the ciphertext.

### AES Operations

`SubBytes` In this stage we replace (or substitute) each byte $B_i$ with another byte $C_i$. The relationship between the two is determined algebraically; however, we shall not discuss the math in this section because it is out of the scope of this book. The basic idea is that we represent each byte by a polynomial, and then perform arithmetic on polynomials. It is typically not required to do this at runtime. For each combination of 8 bits, we can store the result in a lookup table known as the S-box. The aim of this step is to increase the distance between the plaintext and ciphertext as much as possible.

`ShiftRows` In this stage we shift the contents in each row. This is also known as left rotation because the byte that is shifted out enters the row at the rightmost end. The rows are numbered 0, 1, 2 and 3. We left shift the $i^{th}$ row by $i$ positions.

`MixColumns` In this step we take the four bytes of each column, and multiply it with a fixed matrix. The multiplication however uses modular arithmetic (defined over a Galois field [Howie, 2007]). The aim of the `ShiftRows` and `MixColumns` step is to *diffuse* the bits of the plaintext.

`AddRoundKey` This is the last stage. In AES, we derive a key for each round from the original encryption key. The size of the round key is the same as the size of the state (128 bits in this case). We compute a byte-by-byte XOR with the round key. This helps us mix the key with the data.

### Key Schedule

The algorithm for generating multiple round keys in different AES rounds is known as the key schedule. The basic operations for generating each round key are as follows:

`RotWord` We perform a left rotation: $B_0B_1B_2B_3 \rightarrow B_1B_2B_3B_0$.

`SubWord` We substitute the value of each byte using the `SubBytes` operation defined in the previous section.

`XORWord` For a word of the form $B_0B_1B_2B_3$, we replace $B_0$ with $B_0 \oplus RC[i]$. The array $RC$ is known as the round-constant array. We have $RC[1] = 1$ and for $i > 1$, $RC[i] = 2 * RC[i-1]$.

Let us now create a *key matrix* (assuming a 128-bit key) where each element is a single byte. It is constructed in exactly the same way as we constructed the *state* of the algorithm out of plaintext bytes. In this case, we initially use the original key to create the key matrix. It is shown below. This matrix is updated in each round to generate a new round key.

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

Let us use the superscript to specify the round number, for example, $K^i_j$ is the $j^{th}$ key in the matrix for round $i$. Let us use the mnemonics $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{X}$ for the functions `RotWord`, `SubWord`, and `XorWord` respectively. The key matrix is initialised with the AES key (round 0), and for every subsequent round it changes according to the key schedule, which is as follows (reference: [Padhye et al., 2018]).

$$K^i_j = \begin{cases} K^{i-1}_j \oplus K^i_{j-4} & 4 \leq j \leq 15 \\ K^i_0 K^i_1 K^i_2 K^i_3 = \mathcal{X}(\mathcal{S}(\mathcal{R}(K^{i-1}_{12} K^{i-1}_{13} K^{i-1}_{14} K^{i-1}_{15}))) \oplus K^{i-1}_0 K^{i-1}_1 K^{i-1}_2 K^{i-1}_3 \end{cases} \tag{13.1}$$

### Complete Algorithm

The complete algorithm is as follows. Assume that the protocol consists of $N$ rounds. The first step also known as the $0^{th}$ round is called *pre-whitening*. In this stage, only the `AddRoundKey` operation is performed. For the rest of the $N-1$ rounds, all the four operations are performed, and in the last round ($N^{th}$ round), the `MixColumns` operation is skipped. Finally, the state of the algorithm becomes the ciphertext. We can convert the matrix back into a string of 16 bytes.

### Decryption

All the operations in AES are invertible. We start from the final state and perform a reverse order of operations. During this process, we will need the round keys at each stage. To aid this process, the round keys can be generated first, and then stored in a lookup table.

### AES Modes

An AES block can only encrypt 128 to 256 bits (16 to 32 bytes) at a time. For longer plaintext messages we need to use one of the following *encryption modes*.

The key idea here is to divide the entire plaintext message into a set of 128-256 bit blocks, and encrypt each plaintext block separately using a piece of hardware called an *encryption block*. Each such encryption block incorporates one AES block and some additional logic. The encryption blocks can either operate serially or in parallel. Depending upon the encryption mode an encryption block can provide some information to the subsequent encryption block. The overall goal is to increase the parallelism of encryption/decryption as well as ensure good confusion/diffusion properties.

#### Electronic Codebook

This is a naive encryption method, where we divide the plaintext into blocks, and encrypt them with the same key. The main problem with this method is that identical blocks of plaintext produce the same ciphertext. This can reveal significant details about the structure of the plaintext, and thus this is not preferable. Even though we can ensure high levels of confusion, the levels of diffusion will be low.

#### Cipher Block Chaining

Let us solve the problem with the electronic codebook method (see Figure 13.1).

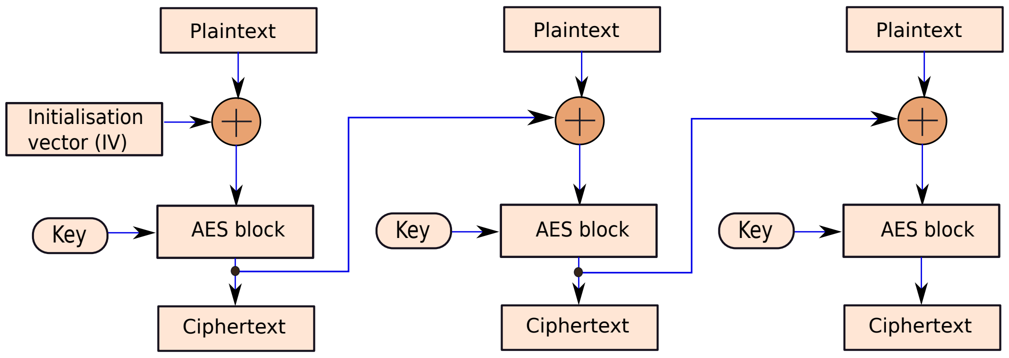


Figure 13.1: Cipher block chaining

In the first block on the left we compute a XOR between the plain text and a pre-decided initialisation vector. We encrypt this block using the secret key. The important feature of this algorithm is that the encrypted cipher is used as the initialisation vector for the second block, which means that the plaintext for the first block determines the ciphertext for the second block. The encryption proceeds in a similar fashion. If we flip a single bit of the plaintext, then the ciphertexts for all the subsequent blocks change. This is considered a default implementation of AES as of 2020. The other advantage of this algorithm is that the decryption can be parallelised even though encryption is a strictly sequential operation. The reason for this is left as an exercise for the reader.

### Output Feedback Mode

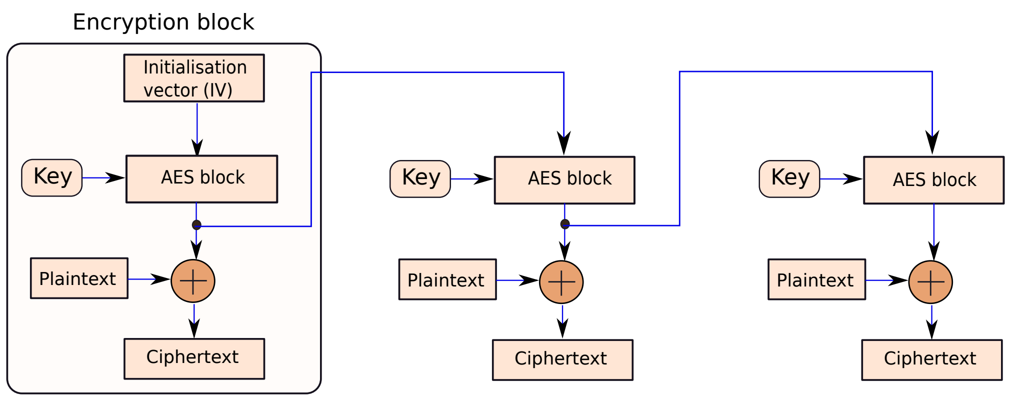


Figure 13.2: Output feedback mode

Having a high diffusion rate has its problems as well. It limits parallelism and introduces a degree of sequentiality in the computation. In addition, if there are errors in the plaintext, then the entire ciphertext is damaged. One approach that fixes the second problem (not the first w.r.t. parallelism) is the output feedback mode (see Figure 13.2). Here, the first encryption block takes the initialisation vector (IV) as input, the second encryption block takes the output of the first AES block as input, the third encryption block takes the output of the second AES block as input, and so on. This approach sequentialises the encryption and decryption processes. The output of each AES block is known as an one-time pad (OTP).

For each encryption block, the ciphertext is a XOR of the output of the AES encryption block (OTP) and the plaintext, which is a very fast operation. This means that if there is an error in a single bit of the plaintext, then there is an error only in a single bit of the ciphertext. The other interesting thing is that given a key we can compute the output of all the AES blocks that are chained together. In parallel, we can perform error correction on the plaintext (if there is a need). Once these parallel tasks complete, we can then quickly compute the ciphertext with a XOR operation.

Off late this approach has been superseded by counter mode encryption because it provides more parallelism.

### Counter Mode Encryption

The main problem with output feedback mode encryption is the lack of parallelism. Counter mode (CTR) encryption fixes this (see Figure 13.3).

The key innovation here is that the encryption blocks are not chained. Instead, their constituent AES blocks use different inputs. Each input is a combination of two numbers: major counter and a minor counter. The major counter remains the same for all the encryption blocks. This means that if we are encrypting a long message we can have a single major counter for all of its constituent plaintext blocks. The value of the major counter should remain confidential. It can be thought of as an encryption key for the entire message.

Now, if we have $k$ blocks in the plaintext, then we need $k$ minor counters. They can be of the form $0, 1, 2, 3, \dots (k-1)$. Each AES block takes in a major+minor counter pair, encrypts it with a secret key that is hard-coded, and produces a one time pad (OTP). This is similar to the OTP that we had generated in the output feedback mode. Subsequently, we compute a XOR between the OTP and a block of plaintext to produce a block of ciphertext.

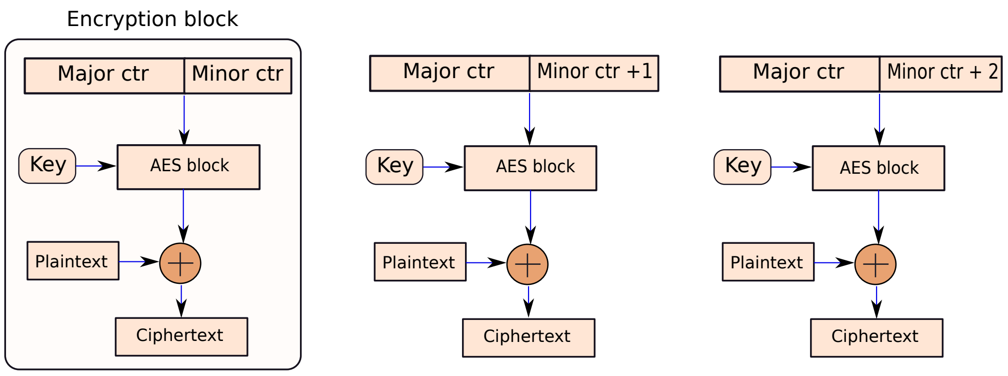


Figure 13.3: Counter mode encryption

The key advantage of having this major-minor counter mechanism is that we just have to store the major counter, the minor counters can simply be generated at runtime. Second, as compared to the output feedback mode, the encryption blocks are not chained together. In this case, both encryption and decryption can be done in parallel across the encryption blocks. We can also simultaneously correct errors in the plaintext.

Finally, note that because the AES algorithm itself has good confusion/diffusion properties, the OTPs are random in nature and extremely hard to predict if the major counter is kept confidential. Even if two blocks of plaintext are the same, they will produce different blocks of ciphertext because their minor counters will be different. Hence, we are not significantly sacrificing on any cryptographic guarantees by using counter mode encryption.

We shall see in later sections that counter mode encryption is the basis of modern secure architectures.

### 13.1.2 RC4 Stream Cipher

RC4 (Rivest Cipher 4) is the most popular stream cipher algorithm as of 2020. As compared to a block cipher, a stream cipher has a different philosophy; it works at the level of individual bytes or small blocks of bytes. It is useful in situations where the length of the plaintext message is not known beforehand such as many wireless communication scenarios.

The key idea of the RC4 cipher is that it generates a sequence of pseudo-random bits known as the *keystream*. We can treat each byte in the keystream as a one-time pad that can be used to compute a XOR between itself and a plaintext byte. The main features of the algorithm are as follows.

The most important structure is the state vector, $S$, which has 256 entries. Here also we have a *key* that is an array of 5 to 16 bytes. The constant *key_length* is the length of the key in bytes. We first have an initialisation phase, and then we have a pseudo-random number generation phase.

#### Initialisation Phase

We first set $S[i] = i$ $(0 \leq i \leq 255)$. Then we run the following code (either in software or in hardware) .

```
1 j=0;
2 for (i=0; i<=255; i++){
3         j = (j + S[i] + key[i%key_length])%256;
4         swap(S[i],S[j]);
5 }
```

Line 3 helps increase the confusion and Line 4 helps increase the diffusion within the vector $S$.

**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]);
}
```

In this case, we are basically computing complex permutations of the bytes in the state vector, and then returning a randomly generated byte that can be used as an OTP. Recall that since each location in $S$ is just one byte, we have 256 possible combinations, which have been stored in $S$ in the initialisation phase. In this phase, we just permute the values and produce one of the values in each iteration as the output. This is a pseudo-random number and can be used as the OTP. We need to compute a XOR between the OTP and a plaintext byte to compute the ciphertext. The process of decryption is very easy. We just need to compute the OTP and compute a XOR with the corresponding byte of the ciphertext to obtain the plaintext byte.

### 13.1.3 Hardware Implementation

Let us briefly discuss the hardware implementations of the AES and RC4 algorithms. For encrypting 128 bits, AES uses 10 rounds with added overheads for initialisation and computing the key schedule. However, the advantage is that all the 128 bits (16 bytes) are encrypted at once. Since all the rounds (other than the last one) do the same computation, we can in principle unroll the loop and have multiple units where each unit computes the result for several rounds as shown in Figure 13.4(a).

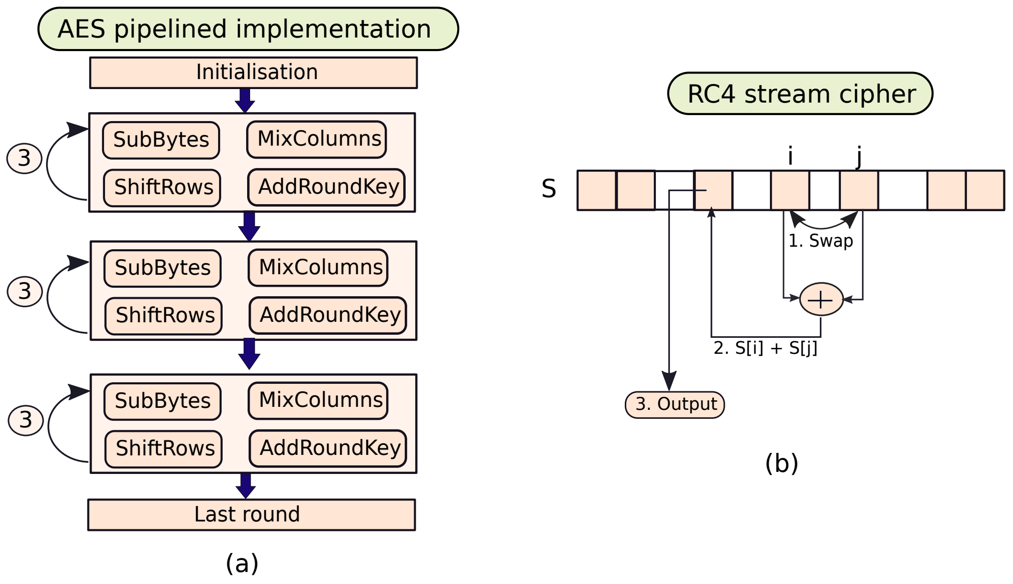


Figure 13.4: (a) Pipelined AES implementation, (b) RC4 stream cipher

If we look closely at Figure 13.4(a) we can discern that we have divided the process of computing the encrypted ciphertext using the AES algorithm into five stages. Each of the three intermediate stages processes three rounds of the algorithm. The advantage is that this structure can be pipelined because there are no dependences across the stages. This can help us tremendously increase the performance of the AES algorithm.

In comparison, the implementation of the stream cipher as shown in Figure 13.4(b) is sequential. We need to read two locations $i$ and $j$, access the state vector $S$, perform a swap (step 1), add the values in $S[i]$ and $S[j]$ (step 2), access that location and read the output (step 3). Even though this process is fast it is nevertheless still sequential and not pipelined – we need to process the data byte by byte. However, it is still possible that RC4 is more power efficient (depends upon the implementation details though).

In modern power-constrained devices used in the internet of things, researchers have developed fast and power efficient block and stream ciphers. This is an active area of research as of 2020.

### 13.1.4 Symmetric and Asymmetric Ciphers

RC4 and AES are examples of symmetric ciphers because we use the same key for encryption and decryption. This means that we need to share the key between the sender and the receiver. This might not be desirable in some cases.

Hence, in this case we opt for asymmetric ciphers that have two keys – a public key and a private key. The RSA (Rivest Shamir Adleman) algorithm is the most prominent in this space. It is typically not used in CPUs because it is at least 1-2 orders of magnitude slower than AES. We shall explain the basic principles in this section. The users should keep in mind that even though RSA is seldom used in secure hardware; however, it is most often the target of attackers because RSA is used extensively in the internet to securely access web pages (HTTPS, SSL, and TLS technologies). Nevertheless, there are exceptions. For example, many technologies built on RSA are used to initialise the secure state of the processor.

#### Basics of the RSA Protocol

Let us first define the modular congruence relation between two integers $a$ and $b$, $a \equiv b \ (mod\, n)$, which means that the positive integer $n$ divides $(a - b)$.

Instead of describing all the math, let us explain the protocol with an example.

1. Choose two distinct prime numbers $p$ and $q$. Let us choose $p = 59$ and $q = 67$. Compute $n = pq$. $n = 3953$.

2. Compute $\lambda(n) = lcm(p-1, q-1) = lcm(58, 66)$, where $lcm$ is the least common multiple. We have $\lambda(n) = 1914$.

3. Let us choose a number $e$ $(< \lambda(n))$ such that $e$ and $\lambda(n)$ are coprime. Let us choose $e = 31$.

4. Compute a number $d$ such that $d \times e \equiv 1 \ (mod\ 1914)$. We can choose $d = 247$. The readers needs to manually verify that $247 \times 31 \equiv 1 \ (mod\ 1914)$.

5. The public key is $(n = 3953, e = 31)$. The encryption function is $\mathcal{E}(m) = m^e \, mod\, n$.

6. The private key is $(n = 3953, e = 247)$. The decryption function is $\mathcal{D}(m) = m^d \, mod\, n$.

Here, is the magic! Let us consider a plain text message and convert it to a number. Say the number is 54. We can easily compute:

$$54^{31} \ \ mod \ 3953 = 3574$$
$$3574^{247} \ \ mod \ 3953 = 54$$

What is even more interesting is that the following relationship also holds!

$$54^{247} \ \ mod \ 3953 = 1011$$
$$1011^{31} \ \ mod \ 3953 = 54$$

Given a key $K = (n, e, d)$ the operation $\mathcal{E}_K = m^e \, mod \, n$ is known as public key encryption and the operation $\mathcal{D}_K = m^d \, mod \, n$ is known as private key encryption. What is intriguing is that the following relationships also hold.

$$\mathcal{D}_K(\mathcal{E}_K(m)) = m \tag{13.2}$$
$$\mathcal{E}_K(\mathcal{D}_K(m)) = m \tag{13.3}$$

**Encryption and Decryption**

Using RSA for encryption is very easy. We simply compute the triplet $K = (n, e, d)$. There are very efficient algorithms to do this. Then we compute the function $\mathcal{E}_K$, which involves modular exponentiation. We can simply use the following property to speedup up our algorithm if $a \equiv b \, (mod \ n)$ and $c \equiv d \, (mod \ n)$, then $ac \equiv (bd \ mod \ n) \, (mod \ n)$. We invite the reader to prove this.

Decryption is a similar process. Note that the numbers $e$ and $d$ are not shared. The only number that is shared is $n$. Furthermore, given $n$ and $e$, it is computationally very hard to compute $d$ if these numbers are large enough. Even though these are commutative operations, the convention is that one pair of numbers $(n, e)$ is known to all – it is referred to as the public key. The other pair $(n, d)$ is private and is only known to one entity (it is known as the private key). We can have implementations where both the pairs of numbers are private (only known to their respective owners).

It is important to appreciate that in RSA, encryption and decryption are essentially the *same operations*. They just use different pairs of numbers (or different keys).

**Digital Signatures**

How do we ensure that a message sent from Alice to Bob is actually sent by Alice? Again this is very easy. We simply encrypt the message with Alice's private key. Bob can then use Alice's public key to retrieve the original message (recall that $\mathcal{E}_K$ and $\mathcal{D}_K$ are commutative operations). If Bob knows some part of the original message, and that part matches, then the message has indeed been sent by Alice. This can be formalised as follows. Alice publishes her public key and a known message. Then she encrypts the known message with her private key – this is known as her *digital signature*. Anybody (including Bob) can validate the digital signature by decrypting it with Alice's public key. They should get the known message back. Whenever a message contains some information that allows us to establish the identity of the sender, the message is said to be *authentic* – this property is known as *authenticity*.

A modern approach to ensure the authenticity of a given piece of hardware is to use a physically unclonable function (PUF). Because of process variations (see Chapter 12), every fabricated chip has a unique signature in terms of the properties of its transistors. It is possible to place sensors at different locations within a chip, measure the values of parameters related to process variation such as the leakage current or temperature, and use them to generate a unique hardware fingerprint (32 to 128-bit number) called the PUF. This serves as a unique id of the device, and as long as the parameters used to generate the PUF remain stable, the PUF uniquely identifies the device. There are several algorithms that use PUFs for authentication, software license management, and secure communication.

### 13.1.5 Session Keys

The main difference between communication on the internet and within a processor is that the latter does not allow us to use algorithms like RSA that are very slow. We need to be able to encrypt and decrypt messages quickly. For that purpose the AES algorithm and faster variants are used. However, the main problem that arises here is how to distribute a common key to the sender and the receiver. During the process of key distribution an attacker can try to read it, particularly, in modern systems-on-chip where different hardware modules are made by different vendors. In such cases, malicious behaviour cannot be ruled out.

We can think of slow solutions at boot time, where some trusted hardware circuit generates and distributes the key using a separate network. It is the job of the system integrator to ensure that this process works correctly in spite of malicious hardware or software. Another slow solution is to use RSA based public-private key communication at boot time to establish a secure connection with a trusted entity and securely obtain an AES key. Then a sender-receiver pair can use such a dedicated key to establish a fast AES-based secure communication channel. Such keys are typically valid for a single session (from power up to power down); they are known as session keys.

There is a faster method to compute a session key. It is known as the Diffie-Hellman key exchange protocol, which is as follows.

1. Alice and Bob decide on two prime numbers $p$ and $q$. These numbers need to be exchanged between them only once. If they are hardware entities, then this can be done at the time of fabrication.
2. Alice generates a secret number $a$, computes $A = q^a \ mod \ p$, and sends $A$ to Bob.
3. Bob does the same, generates a secret number $b$, computes $B = q^b \ mod \ p$, and sends $B$ to Alice.
4. Alice computes $K = B^a \ mod \ p = q^{ab} \ mod \ p$.
5. Bob computes $K = A^b \ mod \ p = q^{ab} \ mod \ p$.
6. Once the reader has verified the math, she will quickly realise that now both have computed the same value of the key $K$ unbeknownst to any entity snooping the channel!

**Definition 102**

**RSA encryption and decryption** *Encryption and decryption in the RSA algorithm are commutative operations. They use the same algorithm albeit with different keys.*

**Digital signature** *A message encrypted with the private key of the sender can be used to establish its* authenticity*. Any receiver can decrypt the message with the sender's public key and if it gets a piece of known plaintext then the message is* authentic*.*

**Session key** *Slow algorithms such as RSA are often used to establish a session key between a pair of communicating nodes. If the nodes have exchanged some information at an earlier point of time, then we can use the faster Diffie-Hellman key exchange algorithm. The nodes can then use this session key and the AES algorithm to exchange encrypted messages.*

## 13.2 Hashing and Data Integrity

### 13.2.1 Common Cryptographic Attacks

Sadly, encrypting data is not enough. We also need to ensure *data integrity* – messages are not tampered. Consider Alice and Bob again. Alice sends a message to Bob (see Figure 13.5).

If Mallory is a passive attacker who can just eavesdrop on the messages being sent on the channel, then there is no issue. This is because we are assuming that Alice has already encrypted the message. Thus, even

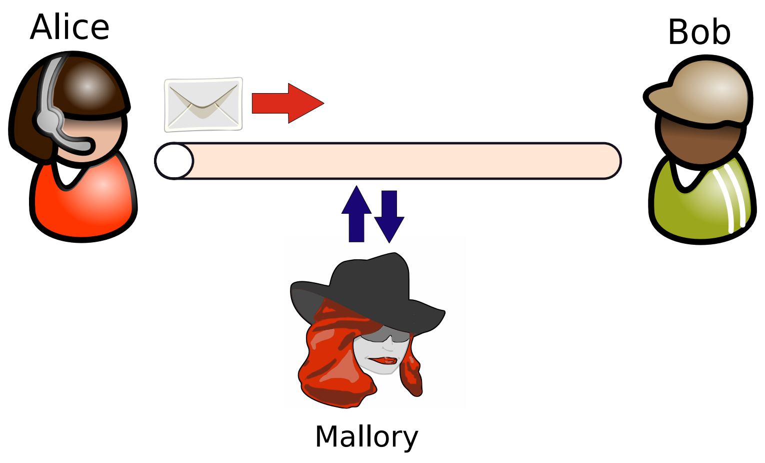


Figure 13.5: Alice trying to send a secure message to Bob. Mallory is the attacker.

if Mallory gets copies of the ciphertext, she will not be able to get any useful information. However, the problems begin if she is an active attacker. This means that she can insert and remove messages from the channel.

One of the classic attacks in this space is called the man-in-the-middle (MIM) attack. In this case, she establishes a secure channel with Alice and masquerades as Bob. She gets the message from Alice, then decrypts and reads the message. Then, she maliciously modifies the message and sends the modified message to Bob. Bob again thinks that the message is coming from Alice. Unbeknownst to both of them the messages are being modified by Mallory. This is not acceptable. Such MIM attacks can be easily thwarted with digital signatures as described in Section 13.1.4. This will establish the authenticity of the sender. It is not possible for Mallory to mount such attacks, which are also known as *spoofing attacks*, where Mallory pretends to be someone else such as Alice or Bob.

Alternatively, Mallory can mount a *splicing attack*, where she captures legitimate encrypted commands and responses from another communication between Bob and another user Carlos. She can then join (or splice) the relevant messages that she has captured with plaintext message headers that Alice uses to communicate with Bob. The spliced message can be sent to Bob. If Alice and Carlos use the same key for creating digital signatures then Bob will get tricked. To avoid such splicing attacks, Alice and Carlos need to use separate keys. This seems reasonably obvious in this case. However, the implication here is that if Alice and Carlos are different memory locations, then we need to encrypt them with different keys.

Let us look at the problem of digital signatures. They need to be sent along with every message. This increases the size of the message. If the size of the signature is $k$ bits, then the probability of mounting a successful spoofing attack can be shown to be $2^{-k}$. To ensure that this probability is vanishingly small, $k$ should be more than 64 bits at least – this increases the message size quite a bit. Second, it takes a lot of time to compute these signatures because we need to perform RSA encryption. We thus need a much faster solution that does not involve sending a digital signature.

### 13.2.2 SHA-based Hashing

Secure Hash Algorithms (SHA) are the de facto standards as of 2020 for hashing a block of data. The main idea of hashing is as follows. We take a piece of text (the message), thoroughly permute and mutate the bits, and finally arrive at a string that is 160 to 512 bits long. This is a one-way hash function, which means that it is not possible to recover the original text from the hash. This hash serves as a signature of the message. Given a message and its hash, the receiver can recompute the hash. If the computed hash matches the hash that the sender has sent, then the message is correct with a very high probability. If the length of the hash is 160 bits, then the probability of two messages having the same hash tends to $2^{-160}$, which is vanishingly small.

Let us now compare this approach with digital signatures, where the key idea is to embed a piece of ciphertext in the message, which is a known piece of text encrypted with the sender's private key. The entire message itself might be encrypted with a session key. The idea is that even if a single bit is tampered, the embedded digital signature will not match with a very high likelihood. However, if we have some other way of enforcing authenticity, and we just want to ensure message integrity, we simply send the hash after the message. The size of the hash will be much smaller than the size of the message, and computing it also will be very fast.

Most secure processors use hashes from the SHA family that as of 2020 has three versions: SHA1, SHA2, and SHA3. SHA2 produces hashes that are between 224 bits and 512 bits. The broad approach is as follows. We break the message into 512-bit chunks. Each chunk is divided into 16 words that are 32 bits each. The processing of each chunk is divided into 64 rounds (same philosophy as AES). In each round we compute a complicated function of the bits in the message that involves permutation, shifting, rotation, and Boolean operations involving round constants. After processing all the chunks, we finally arrive at a hash that is anywhere from 224 to 512 bits (depending upon the variant of the algorithm). The probability of two pieces of text having the same hash is roughly $2^{-l}$, where $l$ is the length of the hash.

### 13.2.3 Message Authentication Code (MAC)

We have seen two solutions: digital signatures and hashes. Both have their flaws. Digital signatures take too long to compute yet provide authenticity; in comparison, hashes can be computed quickly yet do not ensure authenticity. We thus use a message authentication code (MAC) that provides both.

There are many ways to compute a MAC. One of the common approaches used in architectures is to first compute the hash of a message, and then use an encryption algorithm based on a session key to encrypt the hash. The hash of the message needs to be computed anyway, and thus the consequent delays incurred are unavoidable. Since computing hashes is a much faster process than encrypting the entire message this is a tolerable overhead. In the same time, we can use a shared session key to generate a one-time pad (see Section 13.1.1). This will be relatively fast because we are not encrypting the entire message, we are typically encrypting a 256 or 512-bit vector. The MAC is a XOR of the hash and the OTP.

Once the receiver gets the message and the MAC, it proceeds as follows. It proceeds to decrypt the message and the MAC by recomputing the OTP using the shared session key. It then computes the hash of the message, and compares it with the decrypted MAC. If they match, then the message has not been tampered with. Furthermore, it is not possible for a third party to maliciously modify the message or the MAC and still go undetected, because it does not have the shared session key. The reader needs to convince herself of this fact.

There is still a problem unfortunately. Mallory can read a set of (message, MAC) pairs and store them. Later on it can simply *replay* the sequence of messages. If the sequence of messages were to withdraw money from Alice's bank account (Bob being the banker), Mallory can ensure that Alice goes bankrupt! This is a *replay* attack where legitimate commands are recorded and replayed at a later point in time.

### 13.2.4 Preventing Replay Attacks

The sad news is that in spite of ensuring confidentiality and integrity we have the possibility of replay attacks. Thankfully, there are a set of known techniques to prevent replay attacks that are as follows. All of them try to ensure the *freshness* of data, which means that the data that is received is not old data.

**One-time Session Keys** Change the session key for every communication round. This is possible to do with counter mode encryption (Section 13.1.1). This means that the sender and receiver have a pre-decided arrangement where the keys change according to a particular set of rules. Thus replaying messages encrypted with an older key will not work.

**Timestamps** The sender can include its local time in the message. If there is clock synchronisation between the sender and receiver, the receiver can reject messages that were sent before a certain time.

**Nonces** The problem with timestamps is that we need to have large timestamps and also ensure that they do not overflow. We can achieve the same result with *nonces*, which are integers embedded in the message. Every time the sender sends a message it can increment the nonce. The receiver needs to maintain a state variable that stores the last value of the sender's nonce. Here, the assumption is that messages are delivered in FIFO (first-in first-out) order.

**Definition 103**

**Security properties** *We are primarily interested in ensuring four properties in a system for guaranteeing secure communication: authenticity, confidentiality, integrity, and freshness. Let us refer to these properties as the ACIF properties or ACIF guarantees. Many texts also mention* availability *as another property, where attackers can simply try to make a system unavailable by keeping it busy. However, this is not very relevant for processor architectures, and thus we shall not consider it.*

**Eavesdropping** *Eavesdropping is a passive attack where the* eavesdropper *can read the contents of the message. If the message is encrypted then the eavesdropper cannot extract any meaningful information.*

**Man-in-the-middle Attack** *It is possible for an intermediary like Mallory to masquerade as Alice or Bob, and establish a secure communication channel with both of them. She can then read and modify all the messages. This is prevented by using digital signatures that guarantee* authenticity *(establish the identify of the sender).*

**Hashing** *Hashing using the SHA family of algorithms generates short 224 to 512-bit hashes for arbitrarily large pieces of plaintext. These are one-way functions. The message cannot be regenerated from the hash.*

**MAC** *A MAC is an encrypted hash that additionally authenticates the sender.*

**Replay attack** *It is possible for an intermediary to replay both the original message and its MAC. To prevent this, we can either embed timestamps or nonces in the message, or use time-varying session keys. For the latter, counter mode encryption is a very effective technique.*

## 13.3 Secure Architectures

### 13.3.1 Security in Traditional Processors

There are several things that we need to keep in mind before we design a secure architecture. The first is that systems are already designed to be reasonably secure. This means that starting from wearable computers to servers, it is very hard to get unauthorised access to data or maliciously modify a program's execution. The most important security enforcing mechanisms in modern computer systems are the virtual memory sub-system and the file system.

Virtual memory and the paging mechanism ensure that one process cannot read or modify the data of another process in the caches or main memory. Unless intended otherwise, the memory spaces of two processes are separate. It is thus not possible for one process to access (read or write) the memory space of another process. Similarly, for data stored on the disk there are strict file access policies. They ensure that users without the right permissions are not able to access a given file. For critical hardware resources including I/O devices, all the accesses need to go through the operating system, which decides the legitimacy of the requests. This means that for most normal users, most security requirements are taken care of. However, this is not enough for commercial users such as banks and cloud computing providers. Moreover, defence installations and military hardware also have an increased need for security, which traditional mechanisms typically do not provide. Let us give an example of a simple attack that can be mounted on software by providing special input sequences.

### Traditional Attacks on Software

The most common approach in this space is the *buffer overflow* technique. Assume that a program requires a password, which is stored in an array of characters. Instead of entering the password, we enter a very large string. If the program does not check the length of the input, it will store all the characters in an array (known as the *buffer*). There will be an overflow in terms of the size allocated to the buffer. As a result, the data that we enter will get stored on the heap or stack and overwrite previous data. This is where we can play a little trick. In many systems including Intel x86 the function return addresses are stored on the stack. If we carefully craft our input, we can ensure that the overflowed bytes overwrite the return address and make it point to a location of our choosing, which can be the starting address of another section of the large string array. We can thus execute any code that we like including custom code that we inject into the process's virtual memory space using this approach. This would be an example of a *code injection attack*.

Let us look at a simple example in Figure 13.6. In this case, the expected behaviour is that the function returns seamlessly (Figure 13.6(a)). However, it is possible for us to mount a code injection attack, where we overwrite the return address. Figure 13.6(b) shows a situation where the return address points to a location in the buffer.

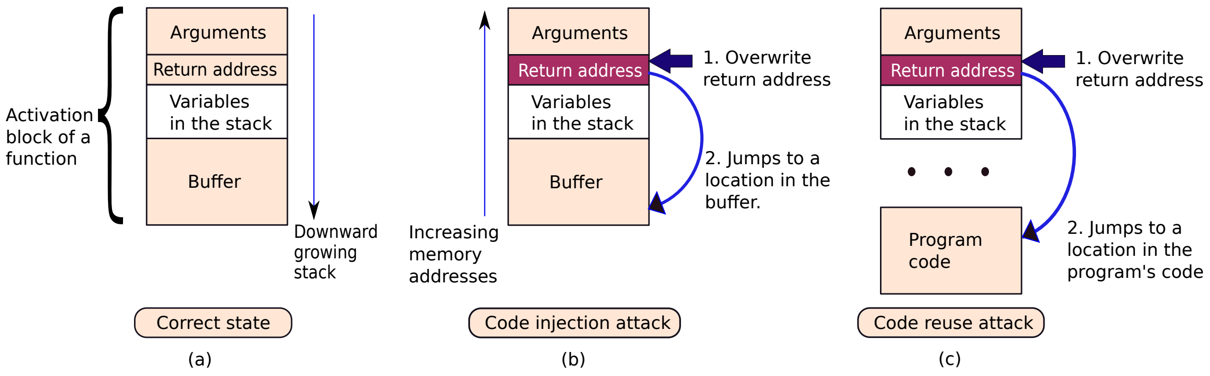


Figure 13.6: (a) Correct execution, (b) Code injection attack, (c) Code reuse attack

Stopping such a code injection attack based on a buffer overflow is easy. The simplest solution is to check every input and ensure that no buffer ever overflows. A more generic way that does not rely on programmers' skill is as follows. Recall that the new instructions are stored on the stack, which is essentially *data*. We just create a method to separately mark data and code pages. A data page is not allowed to contain code. Even if malicious users are able to modify the return address and direct the control flow to an address within a data page, they will not be able to execute the corresponding code.

Alternatively, we can direct the control flow to existing functions within the code of the running program (Figure 13.6(c)). It has been shown that it is possible to string together many such existing functions in the code (known as *gadgets*) to implement a custom logic. This is a *code reuse attack*, and marking data pages as non-executable will not fix this problem. We thus randomise the layout of the virtual address space on every run. This means that the starting addresses of the stack and heap, dynamic linked libraries, and memory-mapped files change with every run. To compute the actual memory address in virtual memory, the code relies on certain base addresses in each of the aforementioned regions. Specifically, for each memory access, we need to add an offset to the corresponding base address. These starting addresses are set by the operating system in a random fashion. The main advantage of this approach is that the attacker does not know the exact addresses of the return address or of functions in the code. This technique is a standard as of 2020. It is known as ASLR (address space layout randomisation).

### 13.3.2 Hardware Security: Key Concepts

Our aim in this section is to introduce some of the concepts that form the bedrock of hardware security: Trusted Computing Base (TCB), Attack Surface, threat model, Trusted Execution Environment (TEE), Trusted Platform Module (TPM), Root of Trust, measurement, remote attestation, and sealing.

#### Trusted Computing Base (TCB)

Whenever we consider a computer system inclusive of the software, CPU, off-chip memory, and storage modules, we need to define the set of entities that we want to secure. This is known as the *Trusted Computing Base*, abbreviated as TCB. It contains a set of hardware and software components that we assume to be secure. In other words, it means that the components of a TCB together guarantee certain security properties, which are often the ACIF properties (see Section 13.2.4).

There are two important principles that are used in designing the TCB.

**Kerckhoff's Principle** This states that the design of the TCB should be open and publicly known. The only secret should be a set of keys. They should ensure the security of the entire system. The reason is that sooner or later the secrets of the design leak out. Hence, there is no point in keeping the design a secret.

**A chain is only as strong as the weakest link** Security is typically thought of as a series-failure system – if one component in the TCB succumbs to an attack, the entire system is compromised. It is assumed that an adversary will always find the weakest part of the TCB and try to attack it.

#### Attack Surface

We can next define the term *attack surface*, which is defined as the set of all the attack mechanisms that can be mounted on the TCB. The larger is the TCB, the larger is the attack surface. Note that attacks can be mounted both by untrusted hardware and malicious software that are outside the TCB. Attacks can either be passive that are limited to eavesdropping, or they can be active where the attacker tries to change the state of the system such as spoofing, splicing and replay attacks as discussed in Section 13.2. Alternatively, the term "attack surface" broadly refers to all the vulnerabilities in the TCB that can be targeted by attacks.

#### Threat Model

A *threat model* is a precise set of attacks that the designers expect to be mounted on their system. They essentially consider a subset of the attack surface and then provide countermeasures against expected attack mechanisms (also known as *attack vectors*). Note that every architecture is designed with a specific threat model in mind. Most commercial architectures typically do not provide countermeasures against all possible threats because this increases the hardware overheads significantly. Designers and architects consider the most realistic scenarios and design an appropriate threat model to thwart most attacks. This is where a trade-off needs to be made between security guarantees and the overheads of providing security.

### Trusted Execution Environments (TEEs)

After the TCB and threat model are defined, it is the job of the hardware vendor to create an environment where secure code can run. This is known as the *Trusted Execution Environment* or *TEE*. The TEE allows users to create secure processes. The TCB guarantees the security of these processes in accordance with the threat model. The TEE creates an execution environment for a secure program that is known as an *enclave*. In most secure processors, we can have multiple enclaves running in parallel.

### Root of Trust

The key question that we need to ask is, "Who sets up the TEE?" If the operating system is malicious, then it can trick secure programs into thinking that they are actually executing within a TEE. It can then steal their secrets such as passwords or credit card numbers. In fact, there are a large number of events that happen before setting up a TEE: the BIOS boots the processor, it initialises the firmware and the I/O devices, the OS is loaded, the TEE is set up, and then the secure program is run in the TEE. If any of these steps is compromised, then the TEE may not be setup correctly. Consider the case of a fake TEE created by the OS. In this case, the secure application should first be able to verify if the TEE is genuine or not. This means that it needs to verify if the entire chain of actions that lead to the creation of the TEE are secure or not.

Let us first consider a *secure boot* process. In the beginning, we need to run a small program called the BIOS (Basic Input/Output System) to initialise and test all the hardware components. The BIOS typically runs in firmware (code stored in a ROM). There is a need to verify if the BIOS has been tampered with. A simple way to do this is to read the code of the BIOS, compute a hash, and verify it with a stored hash. Now, who does this? That module needs to be infallible. Every secure system needs to have such a module that is known as the *root of trust* (RoT), which is assumed to be immune to attacks and is fully trustworthy. It is typically a small secure coprocessor that runs along with the main processor. Such a processor is also known as a *Trusted Platform Module*. The root of trust can alternatively be a piece of software, firmware or even a remote device – depends on the threat model.

The job of the root of trust hardware at boot time is to verify the BIOS by computing its hash and comparing it with a known value. The RoT also offers other cryptographic services to the rest of the system such as encryption, secure storage, and the facility to create digital signatures. It typically maintains a public-private key pair that allows it to *attest* a given piece of data (digital signature). Remote machines that have access to the processor's public key can subsequently verify the digital signature. The key pair can optionally be generated from the processor's PUF.

Finally, note that a system can have multiple RoTs and TPMs.

### Measurement and Chain of Trust

The hash of a piece of code or data that is computed by the RoT is known as a *measurement*. To ensure that the TEE is setup correctly, the RoT can compute a measurement of the BIOS code, which can subsequently compute a measurement of the code of the OS loader. The OS loader can do the same for the OS, and the OS can finally compute the measurement of the secure application. Similar to cipher block chaining (Section 13.1.1), we can combine all the hashes to form a *chain of trust*. The final hash is the measurement of the entire TCB. The RoT can compare this with a stored value, and not allow the system to establish a TEE if the values do not match. If the hashes do not match, then it means that someone has tampered with the system.

The process of measurement starts from the BIOS and ends in the last module that is assumed to be within the TCB. If the OS is a part of the TCB, then the entire chain of trust will contain all the elements of the boot process including the OS. However, if the OS is not a part of the TCB, then the chain of trust will stop at at an earlier point such as the BIOS or the boot loader.

#### Remote attestation

The process of verifying the measurement can be done remotely as well. A secure software can request the processor to generate a digitally signed measurement of the TEE. This can then be sent to a remote machine, which can obtain the public key of the processor from a trusted third party. It can then validate the digital signature and verify if the measurement is correct or not. Such remote machines typically maintain a set of acceptable values of measurements for different configurations. If the measurement that was sent matches one of the stored values, then the remote machine can send data back to the secure software by encrypting it with the processor's public key. This can be decrypted by the RoT for the secure software.

#### Sealing

Secure programs use a lot of data that needs to be written to the hard disk. This data needs to be stored in an encrypted format and should be accessible to only the secure program at a later point in time. We thus need to somehow *tie* the key used to encrypt the data to a measurement. This process is known as *sealing*. Once the secure program is about to write to the disk or other forms of stable secondary storage, it asks the RoT to generate a measurement. This measurement is used to derive a key that is used to encrypt the data. The next time that we read this data back, we can derive the decryption key from the measurement. As long as the measurement is valid, we can read the data; otherwise we cannot read it.

### 13.3.3 Design of a Secure Processor

Let us create a reference design of a secure processor that is loosely inspired by the Intel® SGX (Software Guard Extensions) [Costan and Devadas, 2016] technology.

#### Design of the System

The first thing that needs to be decided while creating a secure processor is the TCB. This is shown in Figure 13.7. We assume that everything within the processor package inclusive of the cores, NoC, and the caches is secure. The processor-memory traffic is visible to an adversary. The threat model is that the adversary can read and write to all the memory locations, snoop and modify the data being sent on the memory bus. In fact this is very easy to do in practice, and is known as a *cold boot attack*. If we lower the temperature by dipping the motherboard in liquid nitrogen, then the DRAM can retain its data for a reasonably long time. This time is enough to translocate the DIMM chips to another motherboard. There we can boot an untrusted OS, read and modify all the data in main memory.

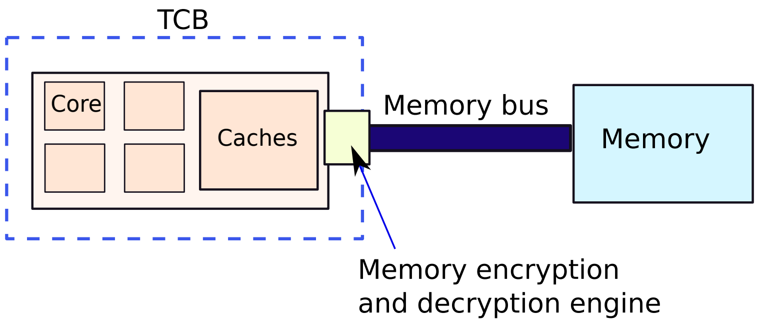


Figure 13.7: The TCB in a typical secure processor

Given the threat model, it is obvious that we need to encrypt the traffic on the memory bus, and store all data in the DRAM memory in an encrypted form. We also need to provide the ACIF guarantees (Section 13.2.4).

### Secure Memory Architecture

The aim is to secure the main memory and the memory bus by successfully avoiding snooping (eavesdropping), spoofing, splicing, and replay attacks. We assume that each memory controller has a dedicated unit called a Memory Encryption Engine (MEE) that implements all the security functionality.

We need to ensure the ACIF properties. Let us consider them one by one in a different order. First consider *confidentiality*, which can simply be ensured by encrypting all the data that is sent on the memory bus. If we always encrypt the same data with the same key, then this is leaking some information to the adversary and is thus unwise. Hence, we need to use a succession of different encryption keys. Next, consider *integrity*. We need to store a hash of every line that we read from memory. For ensuring *authentication*, we can encrypt the hash with a secret key. An encrypted hash is referred to as a Message Authentication Code (MAC). Once the MEE decrypts the hash and compares it with the hash of the data block, it also implicitly verifies the authenticity property. Finally for ensuring *freshness*, we need to avoid replay attacks. This can happen if we encrypt the same block every time with a different key, and the key is stored within the TCB. If any old data is sent by the attacker, the MEE will try to decrypt it with the stored secret key and then verify its hash. The verification process will fail. Note that this paragraph is loaded with concepts, and is fairly difficult to understand for beginners. Hence, we would like to request the reader to read this paragraph several times, understand the ACIF properties, and go through the entire section on cryptographic attacks (Section 13.2). It would be unwise to proceed to the rest of the text without meditating on this paragraph for quite some time.

### Counter Mode Encryption

The key idea of our discussion up till now has been that we cannot encrypt the same piece of plaintext repeatedly with the same key. Even though the attacker cannot figure out the plaintext from the ciphertext, she can at least figure out that the same data is being used repeatedly. This behaviour can be used to mount many successful attacks. Hence, every time we encrypt a block of plaintext, we need to use a different key. This makes our task very complicated because now we need to manage millions of keys.

Instead of using so many separate encryption keys, we can use counter mode encryption that uses different counters in a systematic fashion. The effect is the same as using different encryption keys (see Section 13.1.1). We use two counters: a 64-bit major counter, and a 6-bit minor counter. We store one major counter per physical page (frame), which is assumed to be 4 KB, and we store one minor counter per 64-byte block. A frame contains 64 blocks, and thus we need to store 64 minor counters for each major counter. We cache the *counters* in a dedicated on-chip cache called the *counter cache* that the MEE has access to. Each entry in this counter cache is indexed by the physical page id and the line size is 448 bits. These bits are divided as shown in Figure 13.8. We store a 64-bit major counter and 64 6-bit minor counters.

At this point it is important to recapitulate the concepts in counter mode encryption. Recall that we encrypt the counter-pair with a secret key. The result is the one-time pad (OTP). We compute a XOR of the OTP and a block of plaintext to compute a block of ciphertext.

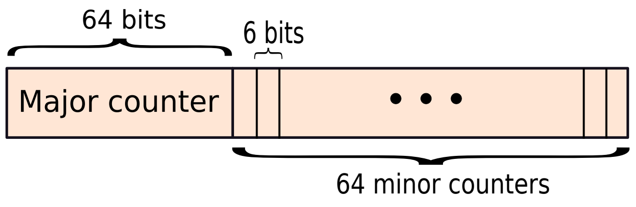


Figure 13.8: 64-bit major counter, and 64 6-bit minor counters (total: 448 bits)

The input to each counter mode encryption block for generating the one-time pad is a combination of the major counter (64 bits), minor counter (6 bits), and the block address (58 bits[1]). This is a total of 128 bits (or 16 bytes). The important point to note here is that we have concatenated the block address with

[1]Assume a 64-bit address space. With a 64-byte block size, the block address becomes 58 bits.

the major+minor counter-pair. The reason is that if two blocks have the same data, they will have the same encrypted contents for the same counter-pair. This can leak some information. To avoid this, we also include the block address along with the counter-pair.

The secret key used in each AES block is a combination of the PUF, a random number generated at boot time, and the id of the enclave. This ensures that other enclaves cannot read the data of the currently running enclave, and every run of the same program produces different encrypted data. This part of the counter mode encryption algorithm is shown in Figure 13.9. After generating the OTP (16 bytes), we need to compute a XOR with a 16-byte plaintext block. The next task is to use four such encryption blocks to encrypt an entire 64-byte line.

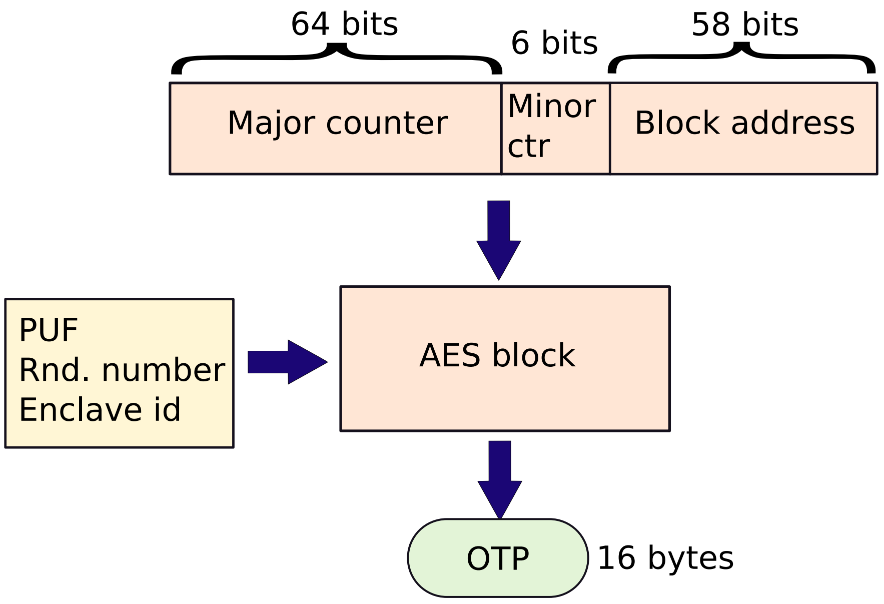


Figure 13.9: Encryption in a secure processor

Let us divide a 64-byte line into four 16-byte chunks, and encrypt them in parallel. The problem is that if two 16-byte chunks are the same, they will produce the same ciphertext. This problem can be avoided by computing a XOR between the 128-bit combination of the $\langle majorcounter, minorcounter, blockaddress\rangle$ and an enclave-specific randomly generated initialisation vector (IV) that is different for each 16-byte chunk. The IV is generated when the enclave is created. This will ensure that two chunks with the same contents are encrypted differently.

Finally, for computing the MAC we can use one of the known hashing algorithms, and the same AES hardware with counters for encrypting it. This method of computing the MAC is known as the Carter-Wegman construction [Wegman and Carter, 1981].

## Read Operation

Every time we have a miss in the last level cache (LLC) we need to send a request to main memory. We read the 64-byte data block, and its associated 64-bit MAC using two separate memory requests. This process takes a long time – typically 100-300 cycles. During this time we use the counter mode decryption algorithm to compute the OTP. The key innovation here is that the latency of this entire operation, computing the OTP, gets hidden in the shadow of the LLC miss. The decryption operation is thus not on the critical path. Once we get the encrypted data and its MAC, we decrypt them using the computed OTP. We can either wait to compute and verify the hash, or we can speculatively send the data to the LLC. Note that whenever we detect an ACIF violation, it is a catastrophic event, and we need to shut the processor down. Hence, there is no harm in sending the data to the LLC and verifying the hash at the same time. If there is no integrity violation (hashes match) then there is no problem, and if the hashes do not match then also sending the data early to the LLC does not matter because we shall lose all the erroneous volatile state by turning off the processor.

Since we verify the MAC, we can detect spoofing attacks – attacker cannot send arbitrary data. The block address is a part of the OTP; we can thus detect splicing attacks – attacker cannot replace the contents of memory locations $(A, B)$ (data, MAC) with the contents of memory locations $(C, D)$. Finally, since we use different counters each time we write a block to main memory, the attacker cannot mount a replay attack.

#### Evict Operation

Whenever, we evict a modified block from the LLC and write it to main memory, we need to encrypt it. This is done as follows. We first increment the minor counter in the counter cache; this creates a new encrypted version of the block. This will ensure that every time a block is written to memory it gets a new OTP (same effect as changing the encryption key). We can increase randomness further by initialising each major counter to a random value. The minor counters can still be initialised to 0.

#### Corner Cases

What happens if a minor counter overflows? This means it reaches $2^6 - 1$ (63). We would not like to reuse counter values – this enlarges the attack surface. Thus, we can reset all the minor counters to 0, and increment the major counter. The major counter is 64-bits long, and in practice will never overflow. This further implies that we need to read all the blocks in the page that are there in main memory, re-encrypt them with the new pair of counters, and write them back. This is an expensive operation. Fortunately, it is rather infrequent.

**Important Point 21**

*The important point to note here is that if all the counters are stored correctly, then the system provides all ACIF guarantees, or the system stops because an ACIF violation is detected. Using counters and MACs prevents eavesdropping, spoofing, splicing, and replay attacks. The reader needs to convince herself of this fact.*

### Integrity Verification: Merkle Tree

#### Classical Approach

Our entire architecture's security only depends on the integrity of the counters. As long as they are deemed to be correct, we can say that the entire execution is secure. If we find a counter in the counter cache, which is located within the chip, then the counter will always be correct because the chip is assumed to be secure. However, the counter cache has a finite capacity and it will be necessary to evict counters to main memory. This is where an attacker might try to change the values of the counters. Hence, we also need to compute and store a hash of the counter values. This will ensure their integrity.

It is unfortunately possible to mount a replay or splicing attack that can simultaneously replace a set of counters and their corresponding hashes or MACs. The classical way of dealing with this problem is to create a structure known as a *Merkle tree*. This is a k-ary tree ($k$ children per parent), where each parent contains the hash of each child's contents. The leaf nodes of this Merkle tree are the major and minor counters used to encrypt the data blocks in a single page (total: 448 bits).

We can compute the hash of these counters, and store the hash in the parent node. We can add one more level by again following the same procedure: compute the hash of the contents of each child, and store it in the parent. We can proceed in a similar manner to create a tree with $log_k(N)$ levels (see Figure 13.10). The most interesting property of the Merkle tree is that we can store the value of the root within the TCB: within a register in the MEE. As long as the root of the tree is stored correctly, the entire tree is correct.

This is the key property of a Merkle tree that we would like to use. The reader needs to convince herself of this fact before proceeding forward. Any tampering anywhere in the tree can be easily detected.

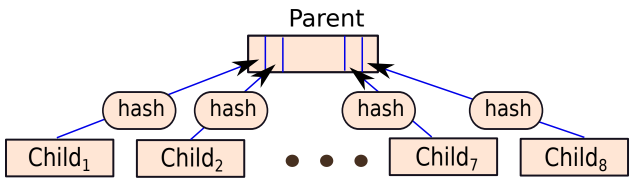


Figure 13.10: A k-ary Merkle tree

In a practical implementation, we can cache nodes of the Merkle tree in the processor's caches. Any cached value is deemed to be correct because it is within the TCB. Now assume that we need to verify the value of a certain counter, we first compute the hash of the major and all the minor counters, and check the hash at the parent node. If the parent node is in the CPU's caches, then it is deemed to be correct, and we need not proceed further. If this is not the case, then we need to verify the contents of the parent – this operation proceeds recursively until we reach a node that is either cached or is the root of the Merkle tree.

Likewise, for writing a group of major-minor counters to main memory, we need to traverse the Merkle tree towards the root, and update all the hashes on the way. This process can terminate when we reach a node that is cached in the CPU (again the reader needs to justify this). The entire process is reasonably inefficient because if a hash is 8 bytes (64-bits) then a 64-byte cache line can only contain 8 such hashes. Thus, $k$ is restricted to 8. For a 1-GB secure memory space we need 7 levels, which is a significant overhead [2].

### Efficient Approach

It turns out that we can use the same Carter-Wegman construction to compress the size of the Merkle tree.

First, recall that the size of the major and minor counters put together was 448 bits for a physical page. This means that we still have 64 bits left in a 64-byte (512-bit) line in main memory. Let us use these bits productively.

Should we use them to store the hash of the counters? This will unfortunately not work because the Merkle tree relies on a parent-child connection – parent stores the hash of the child's contents. This connection is not being established with this technique. Instead of storing the hash, let us store the encrypted hash – the MAC – in these 64 bits, and store the counters used to generate this MAC at the parent node. This solves all our problems – we have a parent-child connection, and the long 64-bit MAC is stored in the leaf node itself.

Let us elaborate. Consider the leaf nodes first. We first compute a 64-bit hash of the contents of a leaf node (major counter + all the minor counters). Then we use the Carter-Wegman approach to read the relevant counters from the parent, and encrypt the hash to produce the MAC. The assumption here is that the parent node is structured in a similar manner: it has one 64-bit major counter, 64 6-bit minor counters, and the space for a 64-bit MAC. Once we compute the MAC of the leaf node, it is stored in the leaf node itself (recall that we had kept 64 bits of space in a line for storing the MAC). The MAC is dependent on the rest of the contents of the leaf node (its counters) and the corresponding major-minor counter pair that is stored in the parent node. We can see the parent-child relationship here, and the way in which integrity is maintained.

We can now generalise this design. Given the fact that the structure of the leaf node and the parent node is the same in terms of the space apportioned for storing the major counter, the 64 minor counters, and the MAC, we can extend this design to all the levels of the tree. All of them are structured in this manner. A node at level $i$ has 64 bits to store its MAC, which is computed using the major-minor counter pair stored in its parent level (level $i-1$). Refer to Figure 13.11.

[2] A 1-GB memory has $2^{18}$ 4-KB pages. There are thus $log_8(2^{18})$ (=6) internal levels plus one level for the leaves. This makes a total of 7 levels.

Note that it is not possible to mount a splicing or replay attack because the parent stores the counters that keep changing with every update. The logic is similar to the reasoning we had used to design our system for storing regular data blocks.

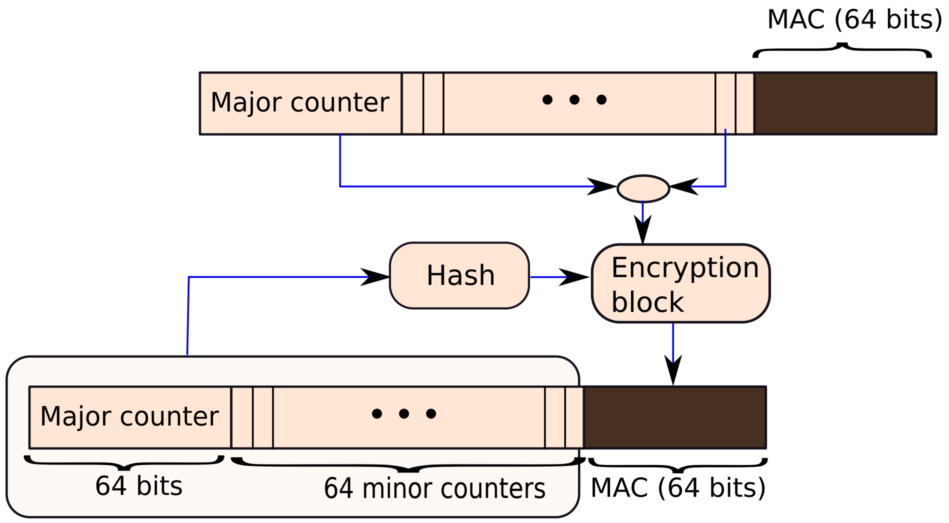


Figure 13.11: Counters and MAC stored in main memory

It is important to stress that we still have a parent-child connection, and here also, the root of the Merkle tree captures the values of all the counters in the system. We can have similar optimisations here also, where we can cache the nodes in the Merkle tree such that every time we do not have to traverse the tree till the root. Let us quantify the advantage.

Recall that the parent node has the same counter structure as the child node; we can thus create a 64-ary tree. For a 1-GB secure memory, we just need 4 levels (including the level that contains all the leaves). If a minor counter overflows in an intermediate node, we need to increment the major counter, set all the minor counters to zero, and recompute the MACs for all the child nodes. Given that nodes at higher levels are expected to receive relatively more updates, the tree can have a lower arity at higher levels.

## 13.3.4 The Software Environment

Most secure processors assume that the OS is not trusted. They provide security guarantees in spite of an untrusted OS, and even assign the OS key responsibilities such as issuing privileged instructions to manage secure execution environments and managing the page tables. This may sound somewhat contradictory, but it is possible to do so.

#### Creating and Initialising an Enclave

A secure processor needs to provide a set of instructions that can be used by the secure program or the operating system to create and manage enclaves (secure execution environments). The typical configuration is that a regular process known as the *host process* creates these enclaves, executes code in it, and ultimately tears it down.

Most secure processors such as Intel SGX provide an `ECREATE` instruction that creates the data structures for an enclave. At boot time, we partition the physical memory into a non-secure part and a secure part. The frames in the secure part can only be used by active enclaves. Furthermore, for every secure process (running instance of a program), we also partition the virtual address space into two parts: secure and non-secure. When an enclave is created a portion of the secure part of the physical address space (frames) and a set of contiguous pages from the virtual address space of the secure process are assigned to it.

Subsequently, we need to provide an `EADD` instruction that adds code and data to the secure enclave. The OS needs to copy code/data from non-secure pages to secure pages. Once the starting state has been set up, we need an `EINIT` instruction to launch the enclave. Before an enclave is launched, the TPM computes its measurement, and digitally signs it. Sometimes, the process of generating an authenticated measurement is very complicated, and thus processors have a dedicated enclave known as the *quoting enclave* that performs cryptographic operations on the measurement in software. The signed measurement is then stored in dedicated control structures. It has several uses. It can be used by an enclave to prove its identity to other secure enclaves running locally, or it can be used to convince a remote third party with regards to its identity.

**Execution within an Enclave**

Subsequently, the enclave starts executing the secure code. During this process the operating system manages the enclave's page tables, and the core running the secure enclave gets the address translations from the TLB. If the OS is not trusted, it can mount an *address translation attack*. This means that it can make a virtual page in the secure region point to a frame in the non-secure region. The secure program will not get to know and it will get tricked into writing secure information to non-secure pages. The OS can thus steal the secure program's secrets unbeknownst to it.

To prevent such attacks, it is necessary to maintain a mapping within secure memory. This is usually an inverted page table (IPT) that is indexed by the physical frame number where each entry contains the virtual page ids that point to it. The reason for using an IPT is because in this case we are more concerned about all the virtual pages that point to a given physical frame; it should never be the case that a secure and non-secure virtual page point to the same frame unless we explicitly want this to be the case. Whenever, pages are added to an enclave by `EADD` instructions, entries in the IPT are created. Before the enclave is launched, we need to check that all the entries in the IPT point to valid pages and frames. A page is valid if it is a known code or data page, and it lies within the portion of the virtual address space marked as secure. A frame is valid if it falls within the secure region of the physical address space. Subsequently, after the enclave is launched, we monitor all the updates to the TLB. Whenever a mapping is created that involves either a secure page or frame, the mapping needs to be first checked in the IPT. If it is an invalid mapping or the same frame is mapped to a different page, then we need to generate a fault and terminate the operation. This ensures that address translation attacks cannot be mounted because any change to the TLB needs to be validated by looking up the IPT first. The crux of this idea is that all updates to the TLB are monitored and we do not allow any update to the TLB to go through without consulting the IPT first. We can implement custom logic in this phase.

Note that we can still have page faults and TLB misses. Page faults need to be handled by the OS. For a TLB miss, we have two options: a dedicated hardware unit can populate the TLB by accessing the page tables or the TLB can be populated by a software module. Since the OS populates the page tables, this process can expose several security risks. They are handled as follows.

1. For secure pages, the hardware needs to zero out the bits that point to the exact memory word in the page that caused a miss, and just report the page id. The OS will thus not be able to see the word-access sequence. It can still see the page-access sequence because it can induce page faults and TLB misses by deliberately clearing the TLB or by swapping out pages.

2. We cannot simply allow a secure page to be swapped out from main memory. The OS can tamper with its contents. Hence, before swapping a page out it is necessary to perform some bookkeeping. First, we need to create a tree akin to the Merkle tree that we used for memory data. Second, we need to compute a MAC for the page with a nonce based scheme (see Section 13.2.4) that preserves the ACIF guarantees and store the $\langle key, nonce \rangle$ pair at a dedicated location in the secure physical address space. When the page is swapped in, we verify its contents.

   Highly flexible implementations can also allow the metadata pages that contain such page-specific nonces and keys to be swapped out. Again, we need to follow the same process and maintain the encryption information in the secure space. This approach allows us to support large enclaves, and also many enclaves simultaneously.

In spite of such measures, the OS can definitely see the page-access sequence upon a page fault. Furthermore, if a TLB miss is handled in software, the OS can see the page id for that too. It has been shown that, from just the page-access sequence, it is possible to derive important information in some cases. Most secure processors do not protect against such *page fault snooping attacks*.

#### System Calls and Interrupts

Interrupts can be delivered to cores executing secure enclaves. In this case, they need to store a *secure context* of the running thread in the secure region. This will include the values of all the registers, the PC, and all encryption information including the enclave id. For such asynchronous enclave exits, it is necessary to flush the TLB such that subsequent accesses cannot be made to the secure region. A dedicated hardware mechanism is required for this.

We also need to support enclave enter (`EENTER`) and enclave exit (`EEXIT`) instructions that can be used to voluntarily enter an enclave and exit it, respectively. Normally, the `EENTER` instruction is issued by the host process to enter an enclave and resume its execution. In this case, the OS can treat this as a regular context switch and store the context of the host process. Additionally, the hardware needs to record the fact that a given secure enclave is executing on the core.

The enclave code needs to issue an `EEXIT` instruction if there is a need to temporarily leave the enclave and execute code in the host process. For example, enclaves are not allowed to issue system calls because we wish to limit their interaction with the OS. Hence, an enclave needs to send the relevant system call arguments to the host process via a shared page, and then issue an `EEXIT` instruction. After the secure context is stored and the TLB is flushed, the OS can resume the host process, which can execute the system call on behalf of the enclave code. The host process can then invoke the `EENTER` instruction to resume the execution of the enclave.

#### Tearing Down an Enclave

It is necessary to provide an `EREMOVE` instruction to tear down an enclave. The OS can invoke it to dismantle the enclave. A dedicated hardware units clears the state of the secure thread including its pages in memory and pages swapped to the disk.

### 13.3.5 Oblivious RAM

Till now we have only focused on protecting the data that is sent or received from main memory. Unfortunately, addresses need to be sent to main memory in an unencrypted form. Conventional DRAM does not support encrypted addresses. An attacker can easily snoop these addresses and gain a lot of information about the execution of the program in terms of the data it is accessing and the code it is running. An attacker can further increase the information content of this process by executing a few instructions of the secure code, and then causing an enclave exit by sending an interrupt. Until DIMMs start supporting encrypted addresses this problem will be there. This is a generic problem and afflicts any bus that carries addresses.

We need to thus create an *oblivious RAM* or *ORAM*, which is a virtual layer over DRAM that obfuscates the access sequence in such a manner that an attacker gets as little information as possible by studying the sequence of addresses. An absolutely naive approach will be to access all the locations in the DRAM for every single access. The CPU can then only choose that value, which it actually needed. Another equally impractical idea is to randomly permute all the addresses at the beginning, and then subsequently when we read a memory line, we write it to a new location. This means that at the memory line level we need to maintain a long list of mappings. These impractical ideas do give us two important insights though. The first is *redundancy* – we need to access a set of locations for every single access. The second is *permutation* –

it is necessary to permute the memory locations such that the physical location for a memory address keeps changing. In fact both are required (the proof is beyond the scope of the book).

ORAM as proposed in the original paper by Goldreich and Ostrovsky [Goldreich and Ostrovsky, 1996] was considered to be very slow to be used in practical settings. Off late a practical implementation known as *Path ORAM* [Stefanov et al., 2013] has been proposed, which still is associated with large slowdowns (2 to 10 times), yet can be used if there is a dire need.

### Path ORAM

Assume that $N$ is the size of the secure memory in terms of blocks. The block addresses are between 0 and $N-1$. This algorithm proposes to maintain a small local cache at the memory controller known as the *stash*($S$), and a *position map*($posMap$) that maps each block address to a unique position in the range $0 \ldots (N-1)$. Given an address $a$, let $S[a]$ represent its entry in the stash, and if $S[a] = \phi$, then it means that address $a$ is not present in the stash. Before presenting the algorithm let us define all the terms and subroutines (read it very carefully).

In the main memory we maintain a complete binary tree with $2^L$ leaves and $L+1$ levels, where $L = log_2(N)$. Each node stores a bucket containing $B$ blocks. These can be real memory blocks or dummy blocks. We assume a function $readPath(k)$ that returns the contents of all the blocks in the path from the root of the tree to the $k^{th}$ leaf. Here the *position* of the leaf is $k$. Let $P(k)$ represent the path from the root (level 0) to the $k^{th}$ leaf (level $L+1$), and let $P(k)[l]$ represent the bucket at the $l^{th}$ level in this path. Let us define the function $getCousins(k,l)$ that returns a set $W$ of block addresses, where each address $a' \in W$ satisfies the following property: $P(k)[l] = P(posMap[a'])[l]$ and $S[a'] \neq \phi$. This function essentially returns all the addresses whose corresponding leaves(positions) are in the subtree rooted at $P(k)[l]$ and whose data is present in the stash. Finally, assume a function $trim$ that takes the set $W$ (output of $getCousins$) and selects a random subset of $B$ elements (adds dummy elements if $\mid W \mid < B$).

The key insight is that the data for address $a$ can be present in any bucket along the path $P(posMap[a])$. We thus need to fetch the entire path, and then slightly permute and rearrange the data in the tree. The pseudocode for a memory access to address $a$ is as follows [Stefanov et al., 2013]. The proof is beyond the scope of this book.

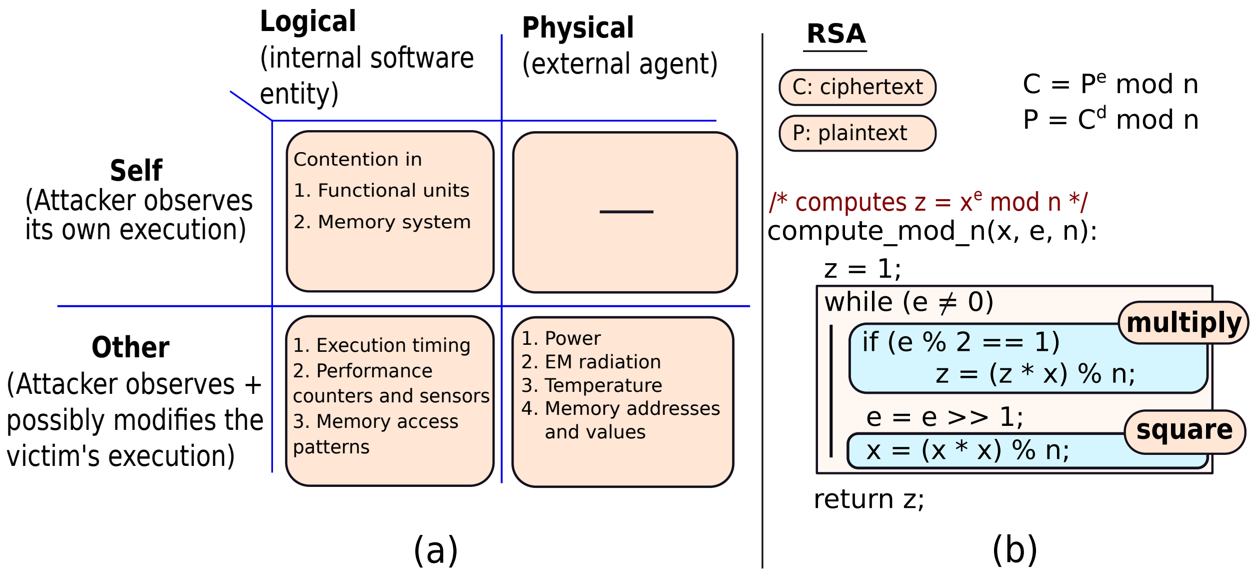
```
/* op is the operation, a is the address, new_data is the data to be written (if
   op==write) */
access(op, a, new_data):

/* 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]
if (op == write) S[a] ← new_data

for l ∈ (L, L−1, ..., 1, 0) {
        W ← getCousins(pos, l) /* get all cousin nodes that are in the stash */
        W ← trim(W)                   /* |W| = B  */
        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 */
        ∀a′ ∈ W, P(pos)[1].add (a′, S[a′])
}
```

## 13.4 Side-Channel Attacks

Recently, your author was narrated a story by a cybercrime investigator. A lady in France was being blackmailed by a kidnapper to pay ransom money. The kidnapper used to call her on WhatsApp and one day as she was moving around with the phone in her house, the kidnapper told her that she was currently in her living room, and was moving towards the kitchen. She got totally unnerved and got convinced that her apartment was bugged. This is why she did not call the police immediately. The fun part is that the kidnapper had not placed any camera in her apartment. He was a known person who knew the layout of her apartment and was guessing her position based on the strength of the Wi-Fi signal, which could again be guessed from the Whatsapp call quality! This is an example of a *side channel* where information leaks out in unusual and unforeseeable ways. We have many such side channels in the CPU and the memory system. Note that such channels have an extremely low information content and bandwidth, yet they can give us very crucial information.

In this section, we present a brief introduction to such attacks. For more details, readers can refer to a survey paper by Szefer [Szefer, 2019]. There is a related term known as a *covert* channel where a process running in a secure environment communicates with a process outside the environment via such a mechanism, particularly, when they are not supposed to communicate. We will focus our attention on side channel based attacks in this section, and use the generic terms *victim* and *attacker*.

### 13.4.1 Classification of Side Channels



```
/* computes z = x^e mod n */
compute_mod_n(x, e, n):
    z = 1;
    while (e ≠ 0)
        if (e % 2 == 1)
            z = (z * x) % n;
        e = e >> 1;
        x = (x * x) % n;
    return z;
```

Figure 13.12: (a) Classification of different kinds of side channels [Szefer, 2019], (b) Running example: the binary modular exponentiation operation in the RSA algorithm

Figure 13.12(a) shows a taxonomy for side-channel attacks. Figure 13.12(b) shows the key part of the RSA algorithm where we perform a modular exponentiation. We will use this code as a running example throughout this section. Note that the number of iterations of the loop is dependent on the number $e$.

Furthermore, each iteration has two basic operations: modular square and modular multiply. Whenever the LSB of $e$ is 1, we execute the instructions for the modular multiply operation. Just by monitoring this operation, which has a unique signature in terms of instruction latencies, power consumption, and i-cache line addresses, we can figure out all the bits in the key, e!

Let us now go through the different types of side-channel attacks. The square at the top left refers to scenarios, where the attacker monitors its own execution. Note that whenever there is a contention between two threads in a shared structure, it is possible for one thread's behaviour to influence the behaviour of the other thread. Thus, a thread can monitor its own behaviour and get some valuable information about the other thread (explained later). The second row refers to scenarios where the attacker monitors the victim and tries to also modify the environment in which the victim operates. Such channels can be of two types: one in which the attacker is a software entity, and the other in which it has physical access to the processor.

### 13.4.2 Type 1: Attacker Monitoring Itself

Whenever there is contention at a shared structure such as the branch predictor, BTB, floating point unit, d-cache, i-cache, L2 cache, interconnect, directory, or the memory controller, we can deduce important information from the nature of the destructive interference.

For a large number of contention-based attacks the key idea is very similar. They rely on *data-dependent accesses*, which means that the addresses of memory instructions (instruction or data) are dependent on data values such as bits in the key. If such an access has been made, then its corresponding data will be present in the cache, and a subsequent access will result in a cache hit. We can use a high-resolution timer to measure the latency of a memory operation and infer if there is a hit in the cache or not. Alternatively, we can check if the contents at an address have been evicted or not by inferring if there is a cache miss. Both these approaches will help us detect and characterise data-dependent accesses. There are many ways of implementing this in practice.

Let us explain how to do this in the context of cache-based side channels for our running example: RSA encryption.

The most popular example in this space is the Prime+Probe technique. Here the attacker thread accesses all the cache lines in the L1 or L2 levels. This is the *priming* phase. Subsequently, it yields the processor to the victim thread. After the victim has executed for some time, the attacker starts to execute again. It measures the time of accessing each cache line with a high resolution nanosecond-level timer (*probing* phase). It can automatically infer the lines that have suffered a cache miss because it takes longer to access them – they need to be fetched from lower levels. In this case, there will be a cache miss if the victim has evicted the block. This can give the attacker some idea about the memory addresses accessed by the victim. For example, if we can execute the victim for very short intervals of time, then in our running example we can find out if it executed the modular multiply operation (a data-dependent instruction access) or not. This will give us the value of one bit in the key.

A similar technique is the Flush+Reload technique that can be used when we have shared pages between the attacker and the victim. Here the attacker first flushes a given set of cache lines from the caches. Many processors already provide such flush instructions. It is alternatively possible to do so by accessing another memory space that is as large as the cache. After this is done, the attacker allows the victim thread to run for a short duration. Then it checks if any of the flushed lines are back in the cache using a high-resolution timer by checking for cache hits and misses. This is a practical technique when the code pages of a cryptographic library such as RSA are shared between the attacker and the victim. The attacker can use this approach to find out which instruction blocks the victim accessed. It can thus find out if the victim executed the modular multiply operation or not.

We can on similar lines exploit contention at the functional unit level. For example, if the victim and attacker are running as simultaneous hardware threads, then the attacker can just issue multiply instructions and analyse the slowdown. If the victim is also issuing multiply instructions, then the attacker will perceive a slowdown because of structural hazards (limited number of multiply units). We can do very similar things with the branch predictor where we try to create an aliasing between the *if* statement in the modular multiply operation and a conditional branch statement in the attacker's code.

Another very interesting attack type in this category is the *Rowhammer attack*. If a given DRAM row is accessed repeatedly, it causes the neighbouring row to leak faster, and the neighbouring row ultimately has bit flips. Any subsequent access to the neighbouring row takes more time because error correction needs to be done; the time required for a memory access operation can be measured with a high resolution timer. To leverage this effect, the attacker first accesses two neighbouring DRAM rows: $R_0$ and $R_1$. We assume that it can control the page-to-frame mapping process. It first repeatedly accesses $R_0$ to increase the probability of $R_1$ developing a bit-flip fault in the future. Just before $R_1$ has bit flips, it schedules the victim. Assume that row $R_0$ is the target of a data-dependent access by the victim thread. Subsequently, if $R_0$ is accessed, $R_1$ will see bit flips. This can be detected with a high-resolution timer, and thus we can get an idea of the victim's memory access pattern.

### 13.4.3 Type 2: Attacker Monitoring or Manipulating the Victim using Software Techniques

This is another class of attacks where the aim is to measure different aspects of the victim's execution.

Here the Evict+Time technique is typically used to exploit cache based side channels. We first measure the execution time of the victim program. Then we evict all the lines from certain cache sets by accessing other data that maps to the same sets. We measure the time of the victim program again. In this case, if the difference in times is more than a certain threshold, then we can conclude that the victim accessed some of the evicted cache sets. Even page fault snooping attacks (Section 13.3.4) by the OS fall in this category.

Similarly, an attacker can use instruction level timing information to learn some features of the victim program. For example, if it has access to accurate timing information such as detailed statistics regarding the execution of functional units, then it can estimate the number of modular multiply operations that have been performed. This will give it an estimate of the number of 1s in the key.

Performance counters that measure the total number of cache hits, misses, memory accesses, and overall instructions can be a very important source of information as well. In the case of our running example, we can also use this information to predict the number of 1s in the key by counting the number of arithmetic operations.

Recently a new class of attacks known as *speculative execution attacks* or *transient execution attacks* have been proposed such as Spectre [Kocher et al., 2019], Meltdown [Lipp et al., 2018], and Foreshadow [Van Bulck et al., 2018]. These attacks extract information from memory read accesses made in the wrong path of a conditional branch. Consider the following code snippet that needs to be a part of the victim program.

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

Assume in this case that *val* is in the control of the attacker – it is an input that can somehow be modified. The attacker can deliberately set it to be greater than the threshold. The double array access may still go through to memory because such wrong-path memory read instructions can still get executed (not committed) before the branch instruction of the *if* statement reaches the head of the ROB. The address that will be accessed is $array2[x] * 4 + array1\_base$ assuming the size of each array element is 4 bytes, and the starting address of $array1$ is $array1\_base$. This access is clearly against the semantics of the victim program. Sadly, we can find this address using the Prime+Probe technique. From this address, we can find the value of $array2[x]$. Furthermore, if the attacker can control $x$, we can read any memory location. It could contain a secret key!

### 13.4.4 Type 3: Attacker Monitoring the Victim by Physically Accessing the System

If the attacker has physical access to the system, then it can monitor the power consumption signature, temperature profile, and the electromagnetic (EM) radiation. If we consider our running example then we can clearly see that the modular multiply and modular square operations will have very distinct power and EM profiles. These can be analysed using sophisticated signal processing algorithms to identify several bits of the key. Temperature of course has a much larger time scale. However, by running the victim program repeatedly, we can get an estimate of the aggregate power consumption and possibly estimate the number of 1s in the key.

The attacker can also physically insert probes into the memory bus and read out all the addresses. Unless ORAM is used, the address pattern will be visible to the attacker.

### 13.4.5 Countermeasures

There are three generic countermeasures used to reduce the probability of such attacks.

1. Partition the on-chip resources to eliminate contention. For example, if there are two threads, we can partition all the cache sets between them. Likewise can partition other structures such as the branch predictor or the floating point unit. This will eliminate destructive interference and stop the attacker from getting any information regarding the victim's behaviour.

2. Deliberately add noise to the computation such that we can eliminate data-dependent accesses. This means that we access many more locations than what is required such that the attacker cannot derive any information from the access pattern.

3. Turn off features such as the high-resolution timer or speculative execution. This is most of the time very expensive.

## 13.5 Summary and Further Reading

### 13.5.1 Summary

**Summary 12**

1. *The goal of any hardware security architecture is to provide the ACIF guarantees: authenticity, confidentiality, integrity, and freshness.*
2. *A good cryptographic cipher should have both the properties of* ***confusion*** *(a single bit of the key determines a large number of ciphertext bits) and* ***diffusion*** *(if we change a single bit in the plaintext, roughly half the bits in the ciphertext change).*
3. *The AES algorithm is divided into four rounds that shift, substitute, and permute the bits. The default AES algorithm typically operates on 16 bytes of data. To encrypt larger pieces of text we need to use one of the AES modes. The most relevant mode is counter mode encryption. Here two counters – a major counter and a minor counter – are concatenated and encrypted with the private key to produce a one-time pad (OTP). This is XORed with the plaintext to produce the cipher text.*
4. *RC4 is the most common stream cipher that produces one byte at a time.*
5. *The RSA algorithm uses two keys: a public key and a private key. Both encryption and decryption rely on modular exponentiation to produce their results. They can be used to create digital signatures to verify the authenticity of the sender, which is tantamount to encrypting a message with the sender's private key. The third party can decrypt this message with the sender's public key and thus verify its identity.*
6. *To verify the integrity of the message, we typically need to compute a short 1-way hash. Sometimes this hash is encrypted with a key to establish authenticity. Such a keyed hash is known as a MAC.*
7. *We can have different kinds of attacks in secure architectures.*

   **Snooping or Eavesdropping** *An attacker reads the data. This can be avoided by using encryption.*

   **Spoofing** *An attacker masquerades as some other node. Solution: use digital signatures.*

   **Splicing** *A part of one message is replaced with a part of another message. Use sender specific MACs to establish integrity and identity.*

   **Replay** *Old messages including their hashes are replayed. We need to add nonces or sequence numbers to messages, or use counter mode encryption.*
8. *The most common attack on software is based on buffer overflows.*
9. *In any secure hardware architecture, we need to assume a trusted computing base, an attack surface, and a threat model. These will be used to create the specifications of a trusted execution environment (TEE). It is necessary to ensure that such a TEE is setup correctly. We need a root of trust (RoT) that first verifies the boot process and provides some cryptographic services that are assumed to be correct. Subsequently, we need to establish a chain of trust from the RoT to the TEE. This is established by computing hashes (measurements) of the relevant code and data of the TCB, and then verifying the overall measurement with known values (locally or remotely). Secure data can also be stored outside the TCB, however it needs to be* sealed *– encrypted with the correct measurement as the key.*

10. *We can design a secure architecture that uses counter mode encryption to provide the ACIF guarantees for blocks in main memory. We just need to protect the integrity of all the counters, which can easily be done with a Merkle tree. For the sake of efficiency, these counters can be stored in a dedicated on-chip counter cache, and a few of the Merkle tree nodes can be stored in the L1/L2 caches. The root of the Merkle tree however needs to always be kept on chip.*

11. *It is necessary to provide a set of instructions to create and manage secure enclaves. Even though the OS is typically not trusted, it still manages the secure applications' page tables. It cannot mount address translation attacks because any update to the TLB needs to be vetted by an inverted page table stored in the secure memory region of each enclave.*

12. *Oblivious RAM (ORAM) introduces redundant memory accesses and permutes the locations to obfuscate the memory access pattern.*

13. *Side-channel attacks are mostly based on deriving information out of destructive interference in contended shared structures. The most common side channels are the caches. In most side-channel attacks on caches, the attacker first runs and sets the state of the cache, then the victim accesses specific caches lines, and finally the attacker runs once again. The timing differences between the first and third runs often yield the set of cache lines that are accessed. If they are dependent on the data, then we can derive useful information about the victim's secrets.*

14. *We can also mount a set of attacks by analysing the victim process's power, EM radiation, or temperature traces. They provide important information regarding the set of instructions that must have executed, which can give us an idea about the secret data.*

15. *Effective countermeasures seek to either partition the hardware resources among the threads, or deliberately introduce noise into the computation.*

### 13.5.2 Further Reading

To work on hardware security a strong foundation in applied cryptography is essential. We would recommend a standard text on cryptography such as the book by Padhye [Padhye et al., 2018] or Stallings [Stallings, 2006]. The e-book by Lee [Lee, 2013] is also very comprehensive. Students should also educate themselves about techniques in light-weight cryptography [Eisenbarth et al., 2007] and study the PRESENT algorithm [Bogdanov et al., 2007].

For hardware security one of the best resources is the e-book by Szefer [Szefer, 2018] and his survey on side-channel attacks [Szefer, 2019]. For papers published in academia, readers should start with some of the seminal papers such as XOM [Thekkath et al., 2000], Aegis [Suh et al., 2005], Bonsai Merkle trees [Rogers et al., 2007], Ascend [Ren et al., 2017] (contains secure ORAM), Bastion [Champagne and Lee, 2010] (support for trusted software), and Vault [Taassori et al., 2018]. Reading the description of Intel SGX [Costan and Devadas, 2016] by Costan et al. is mandatory in this area, along with relevant literature on ARM® Trustzone® [Ngabonziza et al., 2016,Pinto and Santos, 2019]. For ORAM, readers should start with Goldreich and Ostrovsky's paper [Goldreich and Ostrovsky, 1996] and then study Path ORAM [Stefanov et al., 2013].

The survey paper by Szefer on side-channel attacks [Szefer, 2019] is a good starting point in this area. Some of the most publicised exploits today use speculative execution such as Spectre [Kocher et al., 2019] (transient execution attacks), Meltdown [Lipp et al., 2018] (read kernel data), and Foreshadow [Van Bulck et al., 2018] (read data in an SGX enclave). Readers should also make themselves familiar with power analysis attacks [Ors et al., 2004] and attacks based on analysing electromagnetic emanations [Sehatbakhsh et al., 2020].

## Exercises

**Ex. 1** — Design an LLC replacement policy that prevents eviction-based attacks.

**Ex. 2** — Design a scheme that prevents the denial-of-service attack at the DRAM level. For mounting such an attack, an attacker sends a flurry of requests to the DRAM, and this causes other threads to starve. How can we incorporate fairness in the DRAM memory controller to ensure that this does not happen?

**Ex. 3** — Most secure architectures such as Intel SGX have a threat model that assumes that the processor is trusted and the OS is untrusted. Such Trusted Execution Environments ensure that the applications run securely even in the presence of a malicious OS. However, it lacks trusted I/O paths, and thus I/O messages need to pass through the OS. It is possible for the OS to maliciously read and modify I/O data. Propose a solution to this problem.

**Ex. 4** — Design a scheme that improves the performance of integrity trees used to prevent replay attacks. The access frequencies of different blocks in the memory space is non-uniform. How can we design our integrity trees to take this into account?

**Ex. 5** — Simulate a secure architecture such as Intel SGX in an architectural simulator.