

Lecture 6: Secret-key Encryption in Practice

MIT - 6.5620

Lecture 6 (September 23, 2026)

Warning: This document is a rough draft and may contain bugs.
Please feel free to email us with corrections.

Today

We will:

- Finish up the proof of the Goldreich-Levin Theorem (see Lecture 5 notes for the complete writeup);
- Define Message Authentication Codes (MAC)
- Construct a MAC from any PRF.
- Describe the Advanced Encryption Standard (AES), a pseudorandom function (in fact, a pseudorandom permutation) that is used in practice.

PRF for Authentication

So far we were focused on encryption, the task of communicating in a secret manner. We spent three weeks figuring out how to do this. We showed how to use a OWP to construct a PRG,¹ how to use a PRG to construct a PRF, and finally how to use a PRF to construct a CPA-secure encryption (i.e., a scheme that remains secure even if the adversary gets to see ciphertexts of any messages of its choice, which he can choose adaptively).

Next, we will tackle a different problem: Suppose Bob has received a message claimed to be sent by Alice, but he is worried that perhaps an adversary sent the message and claims it is from Alice. Even if the message was encrypted using a CPA-secure encryption scheme with a secret key that only Alice and Bob hold, this does not guarantee that the message was actually sent by Alice.

Until now, we were only concerned with the secrecy of their communication, the guarantee that nobody else can read Alice and Bob's messages. But we have not been concerned at all with the authenticity of their communication, the guarantee that nobody can forge a message in a way that makes it look like it was sent by Alice. This will be our focus next.

¹ As mentioned, it is known how to construct a PRF from any OWP, which is a minimal assumption [1], but this proof is much more involved.

One solution that may come to mind, is to have Alice “sign” her message, i.e., add a (ridiculous) scribble to the end of her message. This of course does not work! An attacker can take a message “Attack at dawn” with Alice’s scribble, and change it by only changing the word “dawn” to “dusk.”

A much much better solution would be for Alice to produce a signature that somehow depends on the message she is signing. In other words, for different messages, Alice should produce different signatures that the receiver is convinced that indeed it was Alice who sent the message, and the message was not altered by an adversary.

Not to mention that it is quite easy to copy these scribbles! It is surprising that it works reasonably well in practice.

Message Authentication Codes (MACs)

Definition 1. A MAC corresponding to a key space $\mathcal{K} = \{\mathcal{K}_\lambda\}_{\lambda \in \mathbb{N}^2}$ and a message space $\mathcal{M} = \{\mathcal{M}_\lambda\}_{\lambda \in \mathbb{N}}$ is a poly-time computable function

$$\text{MAC} : \mathcal{K}_\lambda \times \mathcal{M}_\lambda \rightarrow \{0,1\}^*$$

that satisfies the following security guarantee, referred to as *existential unforgeability against adaptive chosen message attacks*:

Security:³ For every poly-size adversary $\mathcal{A}$ there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$ the adversary wins in the following game with probability $\leq \mu(\lambda)$:

² We typically assume that $\mathcal{K}_\lambda = \{0,1\}^\lambda$.

³ This security notion should sound familiar :-)

1. The challenger chooses a random key $k \leftarrow \mathcal{K}_\lambda$.
2. The adversary sends the challenger a message $m_i \in \mathcal{M}_\lambda$ and receives a “tag” $\tau_i = \text{MAC}(k, m_i)$.

This step can be repeated polynomially many times.

3. The adversary outputs (m^*, τ^*)
4. The adversary wins if $m^* \notin \{m_i\}$, where $\{m_i\}$ is the set of messages the adversary requested tags for, and if $\tau^* = \text{MAC}(k, m^*)$.

Remark. A more concise way of writing the above security definition is as follows: For every poly-size $\mathcal{A}$ there exists a negligible μ such that for every $\lambda \in \mathbb{N}$

$$\Pr[\mathcal{A}^{\text{MAC}(k, \cdot)}(1^\lambda) = (m^*, \tau^*) : \tau^* = \text{MAC}(k, m^*) \wedge m^* \notin Q] \leq \mu(\lambda),$$

where Q denotes all the oracle queries that $\mathcal{A}$ sends its oracle.

Remark. There are many variants of this security definition. For example, a weaker security guarantee is *random* unforgeability against adaptive chosen message attacks, where the adversary wins if it generates a valid tag $\tau = \text{MAC}(k, r)$ for a *random* message $r \leftarrow \mathcal{M}_\lambda$ chosen by the challenger, after seeing polynomially many tags for

messages of its choice (chosen before r was given). Another, even weaker, security notion is random unforgeability against random chosen message attacks, where the adversary wins if it generates a valid tag $\tau = \text{MAC}(k, r)$ for a random message $r \leftarrow \mathcal{M}_\lambda$ chosen by the challenger, after seeing tags for polynomially many *random* messages $r_1, \dots, r_t \leftarrow \mathcal{M}_\lambda$, where $r^* \notin \{r_i\}_{i \in [t]}$.

Remark. Often a MAC is associated with a verification algorithm. With our definition, a tag τ on a message m is verified by checking that $\tau = \text{MAC}(k, m)$. Separating the task of verifying a tag from the task of generating one, allows the task of verification to be possibly more efficient than the task of computing a MAC. In practice (and in our theoretical constructions) verification is done by recomputing the MAC.

Also, as with an encryption scheme, a MAC is often associated with a key generation algorithm. We assume for simplicity, without loss of generality, that the key is uniformly sampled from the key spaces.

MAC Construction from any PRF

The construction is extremely simple: Take any PRF $: \mathcal{K}_\lambda \times \mathcal{M}_\lambda \rightarrow \{0, 1\}^\lambda$. Define:

$$\text{MAC}(k, m) = \text{PRF}(k, m).$$

Theorem 2. *The above MAC is existentially unforgeable against adaptive chosen message attacks.*

PRFs in Practice: The Advanced Encryption Standard

In practice, we typically construct pseudorandom functions, in fact pseudorandom permutations, directly. A *block cipher* is a family of efficiently computable permutations

$$E_K : \{0, 1\}^n \rightarrow \{0, 1\}^n,$$

where the secret key K selects a permutation, and knowledge of K also allows efficient inversion. The security property is *pseudorandomness*: for a randomly chosen secret key, oracle access to E_K should be computationally indistinguishable from oracle access to a uniformly random permutation on n -bit strings.

In other words, a block cipher is simply another term for a pseudorandom permutation (PRP).

The Advanced Encryption Standard (AES) is a concrete block cipher with $n = 128$. It is a basic component of constructions for encryption and authentication, and modern processors often provide

instructions that accelerate its evaluation. To understand its design, we first describe two general construction frameworks.

Key-alternating ciphers

Fix a public permutation $\pi : \{0,1\}^n \rightarrow \{0,1\}^n$ for which both π and π^{-1} are easy to compute. An r -round *key-alternating cipher* (KAC) alternates applications of π with XORs of secret round keys $K_0, K_1, \dots, K_r \in \{0,1\}^n$. For an input x , define

$$\begin{aligned} X_0 &= x \oplus K_0, \\ X_i &= \pi(X_{i-1}) \oplus K_i, \quad i = 1, \dots, r, \\ E_{K_0, \dots, K_r}(x) &= X_r. \end{aligned}$$

There are r permutation applications and $r + 1$ key additions, including the initial and final key additions. For example, two rounds give

$$E_{K_0, K_1, K_2}(x) = \pi(\pi(x \oplus K_0) \oplus K_1) \oplus K_2.$$

Every step is invertible. Given X_i and K_i , recover the preceding state as $X_{i-1} = \pi^{-1}(X_i \oplus K_i)$, then undo the initial XOR with K_0 .

In this simplified model, the round keys are independent and uniform. This gives a total of $(r + 1)n$ key bits. Practical ciphers instead derive their round keys from a shorter *master key* using a *key schedule*.

Substitution-permutation networks

A *substitution-permutation network* (SPN) gives the public permutation π an explicit internal structure. Write $n = kb$ and view a state as a column vector of k words of b bits each:

$$\mathbf{x} = (x_1, \dots, x_k)^T \in \mathbb{F}_{2^b}^k.$$

The parameter k is the *width*. Under this representation, field addition is bitwise XOR. An SPN round uses two ingredients:

1. A public nonlinear permutation $S : \mathbb{F}_{2^b} \rightarrow \mathbb{F}_{2^b}$, called the *S-box*. Apply it separately to every word:

$$\bar{S}(\mathbf{x}) = (S(x_1), \dots, S(x_k))^T.$$

2. An invertible matrix $M \in \mathbb{F}_{2^b}^{k \times k}$, called the *linear mixing layer*. Apply it to the substituted words.

Thus the public round permutation is

$$\boxed{\pi(\mathbf{x}) = M \circ \bar{S}(\mathbf{x}).}$$

Substituting this π into the KAC construction gives the SPN. The S-box introduces nonlinearity within each word. The mixing layer spreads the influence of those words across the state so that subsequent rounds combine information from different positions. Both ingredients matter: an entirely linear substitution layer would leave the whole cipher affine, while independent substitutions without mixing would keep the words separate.

AES as an SPN

AES uses sixteen bytes per block, so

$$n = 128, \quad b = 8, \quad k = 16.$$

Its three standardized variants have the following parameters:

Variant	Master-key length	Block length	Rounds r
AES-128	128 bits	128 bits	10
AES-192	192 bits	128 bits	12
AES-256	256 bits	128 bits	14

The suffix specifies the master-key length. Every round key has 128 bits.

The S-box. Identify a byte with an element of $\mathbb{F}_{2^8}$. First apply the *patched inverse*

$$I(x) = \begin{cases} x^{-1}, & x \neq 0, \\ 0, & x = 0. \end{cases} \quad \text{Equivalently,} \quad I(x) = x^{254}.$$

The AES S-box is

$$S_{\text{AES}}(x) = A(I(x)),$$

where A is a fixed invertible affine map on the eight-dimensional vector space $\mathbb{F}_2^8$. Thus $A(u) = Lu \oplus c$ for an invertible binary 8×8 matrix L and a fixed byte c .

The mixing layer. Arrange the sixteen bytes as a 4×4 state. The linear layer is

$$M_{\text{AES}} = \text{MixColumns} \circ \text{ShiftRows}.$$

ShiftRows cyclically shifts row i left by i byte positions, for $i = 0, 1, 2, 3$. *MixColumns* then multiplies each column by the fixed matrix

$$C = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}.$$

The entries are hexadecimal byte representations of field elements. Arithmetic takes place in the AES field $\mathbb{F}_2[z]/(z^8 + z^4 + z^3 + z + 1)$, so multiplication here is finite-field multiplication.

MixColumns combines the four bytes within each column, while ShiftRows changes which bytes share a column in later rounds.

The actual AES round structure. AES begins with AddRoundKey. Its ordinary rounds apply SubBytes, ShiftRows, MixColumns, and AddRoundKey, in that order. With $X_0 = x \oplus K_0$, this is

$$X_i = M_{\text{AES}} \circ \overline{S_{\text{AES}}}(X_{i-1}) \oplus K_i, \quad 1 \leq i < r.$$

The final round omits MixColumns:

$$E_K(x) = \text{ShiftRows}(\overline{S_{\text{AES}}}(X_{r-1})) \oplus K_r.$$

All round keys come from the AES key schedule applied to the master key.

Security via Iteration

An individual round has simple, visible structure: small S-boxes act on separate bytes, followed by a linear map and a key addition. The design goal is that repeating these operations produces a family that behaves like random permutations to an efficient adversary. That claim is usually supported by cryptanalytic evidence.

This motivates the question raised at the end of the lecture: which security properties can we prove for such explicit constructions? One approach studies limited statistical guarantees, such as almost t -wise independence under independent round keys. These compare the joint outputs on a fixed collection of distinct inputs with the outputs of a random permutation. They provide a way to analyze specified classes of attacks.

References

- [1] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.