AMERICAN UNIVERSITY OF BEIRUT

Applied Cryptography

CMPS 297AD/396AI

Fall 2025

Part 1: Provable Security

1.5: Chosen-Plaintext & Chosen-Ciphertext Attacks

Nadim Kobeissi

https://appliedcryptography.page

Section 1

Chosen-Plaintext Attacks

CPA Security

Security Against Chosen-Plaintext Attacks (CPA Security)

Let Σ be an encryption scheme, and let $\Sigma.\mathcal{C}(\ell)$ denote the set of possible ciphertexts for plaintexts of length ℓ. If Σ supports only plaintexts of a single length, we can simply write $\Sigma.\mathcal{C}$ to denote the entire set of ciphertexts.[a]

Σ has security against **chosen-plaintext attacks** if the following two libraries are indistinguishable:

[a] i.e. "just forget about the length".

$\mathcal{L}^{\Sigma}_{\text{cpa-real}}$

$K \twoheadleftarrow \Sigma.\mathcal{K}$

$\underline{\text{CPA.ENC}(M)}$:
$C \coloneqq \Sigma.\text{Enc}(K, M)$
return C

$\approx$

$\mathcal{L}^{\Sigma}_{\text{cpa-rand}}$

$\underline{\text{CPA.ENC}(M)}$:
$C \twoheadleftarrow \Sigma.\mathcal{C}(|M|)$
return C

Why CPA security matters

- CPA security means: *"even if I let you encrypt any message you want, you can't obtain any distinguisher regarding my scheme."*
- **Why this matters in real life:**
  - Attackers often can trick systems into encrypting data they choose.
  - Without CPA security, seeing these encryptions could reveal your secrets.
  - Example: If bank transactions always encrypt to the same ciphertexts, attackers could identify your purchases.
- The only thing that's allowed to leak is the length of messages/ciphertext.
- Most modern encryption is designed to have this important property.

Message length: not important, not unimportant

- Even with CPA security, the **length** of messages is still leaked.
- This seemingly minor leak can reveal surprising information:
  - **Encrypted VoIP calls**: Length patterns can reveal which language is spoken.
  - **Encrypted web traffic**: Sizes of requests/responses identify websites.
  - **Encrypted messages**: Length patterns can reveal the type of content (document, image, etc.)
  - **Encrypted commands**: Length often reveals which command was issued.
- Mitigation typically involves padding messages to fixed lengths or standard increments.
- This is why many secure protocols use fixed-size packets or add random padding.

CPA security is why we avoid deterministic encryption

- Deterministic encryption will always fail CPA security!
- If the same message always encrypts to the same ciphertext:
  - Attacker can recognize repeat messages.
  - Can build a “dictionary” of known plaintexts.
  - etc.
- ECB mode is a classic example of insecure deterministic encryption.

$\mathcal{A}$

$M \twoheadleftarrow \mathcal{M}$

$C_1 := \text{CPA.ENC}(M)$

$C_2 := \text{CPA.ENC}(M)$

return $C_1 == C_2$

Non-deterministic encryption isn't hard...

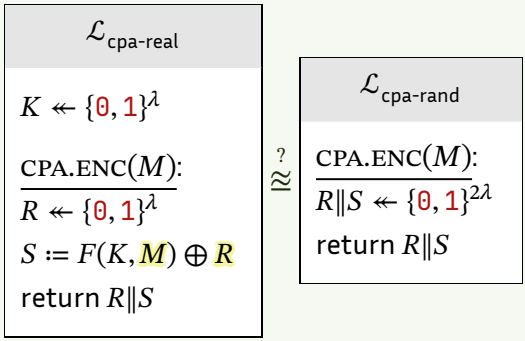
- AES-CTR turns AES into a non-deterministic PRF...
- AES-GCM even turns it into a non-deterministic authenticated cipher...[a]
- We can just as easily make a PRF non-deterministic:

$\underline{\text{ENC}(K, M)}$:
$R \twoheadleftarrow \{0,1\}^\lambda$
$S \coloneqq F(K, R) \oplus M$
return $R \| S$

[a] More on these later. An authenticated cipher is one where the ciphertext can't be modified by the adversary without that being detected.

...but it can be fragile

- We switch M and R's places in $\mathcal{L}_{\text{cpa-real}}$'s encryption function.
- Is the scheme still secure?
- **No!**



The Golden Rule of PRFs

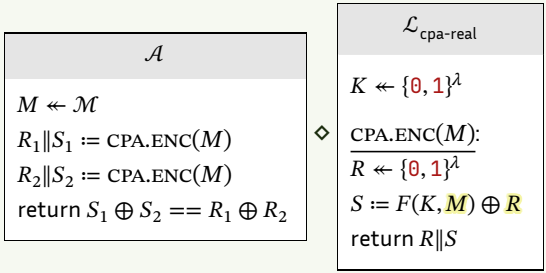
The Golden Rule of PRFs

If a PRF F is being used as a component in a larger construction H, then security usually rests on how well H can ensure distinct inputs to F.

- When analyzing PRF security, focus on input uniqueness.
- Repeated inputs to a PRF create exploitable patterns.
- Even if F is secure, H can be broken if it causes F to receive duplicate inputs.
- Don't try to directly distinguish F's outputs from uniform.
- Instead, exploit how H uses F incorrectly.
- Find input patterns that force collisions within F.

...but it can be fragile

- $F(K, M)$ is now constant.
- $\Pr\left[\mathcal{A} \diamond \mathcal{L}_{\text{cpa-real}} \Rightarrow \texttt{true}\right] = 1$

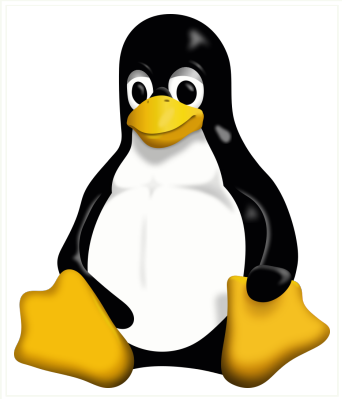


AES is a block cipher

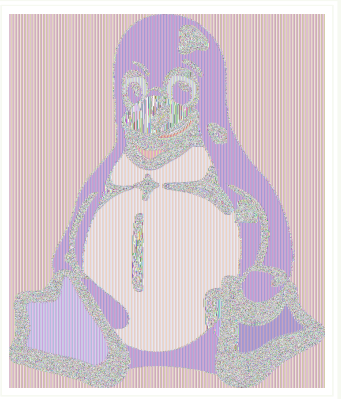
Reminder

- AES takes a 16-byte input, produces a 16-byte output.
- Key can be 16, 24 or 32 bytes.
- OK, so what if we want to encrypt more than 16 bytes?
- **Proposal**: split the plaintext into 16 byte chunks, encrypt each of them with the same key.

Block cipher examples



What we start with



What we get

What we actually want

AES by itself is deterministic

- AES as a block cipher is inherently **deterministic**:
  - $\mathsf{AES}(K, M) = C$ will always produce the same C.
  - This makes raw AES fail CPA security.
- What makes AES secure in practice?
  - **Block cipher modes** (CBC, CTR, GCM) add randomness/state using "initialization vectors"
    - IV is fancy name for "random bytes used once".[a]
  - IVs ensure the same plaintext encrypts differently each time.
- Without a proper mode, AES is like ECB mode: predictable patterns

$\mathcal{A}_{\mathsf{AES}}$

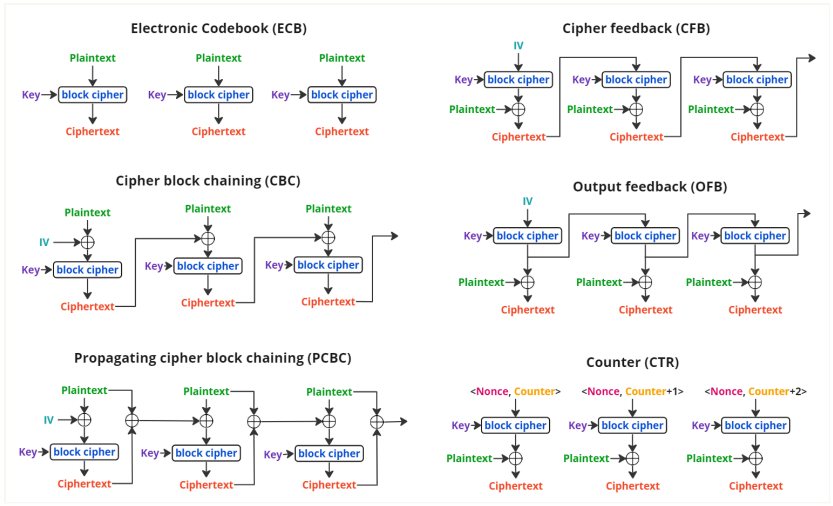
$K \twoheadleftarrow \{0,1\}^{\lambda}$
$M \twoheadleftarrow \mathcal{M}$

$C_1 := \mathsf{AES}(K, M)$
$C_2 := \mathsf{AES}(K, M)$

$C_1 == C_2$

[a] Also referred to as "nonces" for "number used once", although in some block cipher modes such as AES-CBC, they can technically be used multiple times without problems.

Block cipher modes of operation



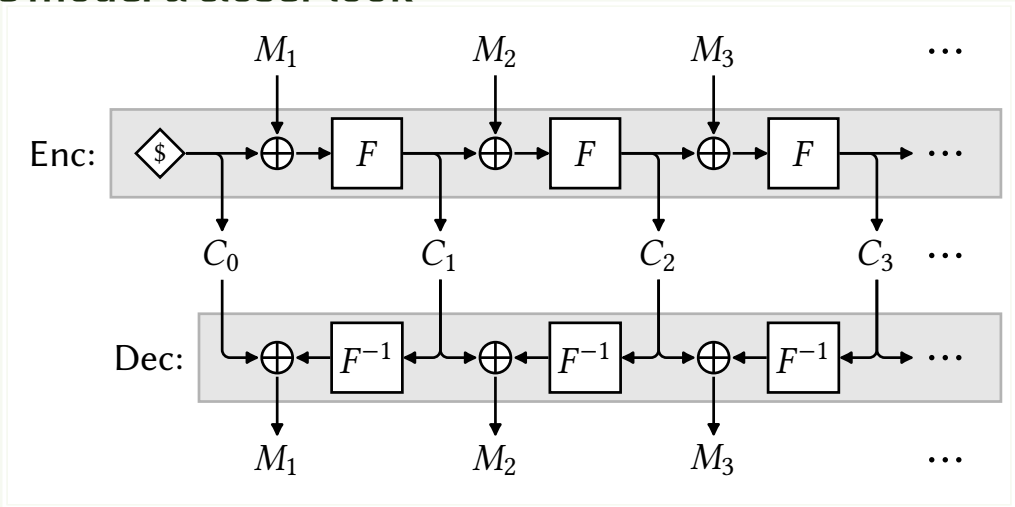
Source: Wikipedia

CBC mode: a closer look

- CBC (Cipher Block Chaining) uses previous ciphertext to randomize current block.
- Requires a random initialization vector (IV).
- Each block's encryption depends on all previous blocks.
- Changes to one block affect all subsequent ciphertext blocks.
- Sequential encryption (can't parallelize).

$\underline{\text{ENC}(K, M_1 \| \cdots \| M_\ell):}$
$C_0 \twoheadleftarrow \{0,1\}^\lambda$
for $i = 1$ to ℓ :
  $C_i := F(K, C_{i-1} \oplus M_i)$
return $C_0 \| C_1 \| \cdots \| C_\ell$

CBC mode: a closer look



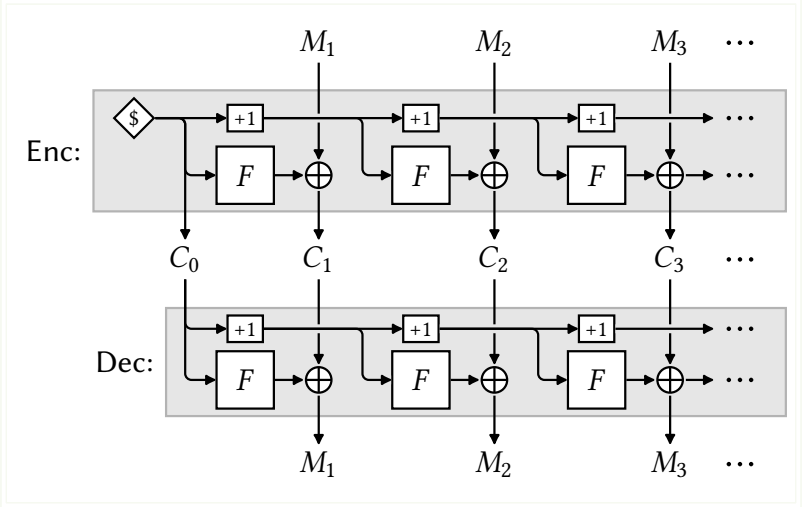
Source: The Joy of Cryptography

CTR mode: a closer look

- CTR (Counter) mode turns a block cipher into a stream cipher.
- Uses a nonce (C_0) plus a counter to create unique inputs to F.
- Each block's encryption is independent of other blocks.
- Highly parallelizable (unlike CBC).
- The nonce must never be reused with the same key.
- Widely used in modern protocols (TLS, SSH, etc.)

$\underline{\text{ENC}(K, M_1 \| \cdots \| M_\ell):}$
$C_0 \twoheadleftarrow \{0,1\}^\lambda$
for $i = 1$ to ℓ :
  $C_i := F(K, C_0 + i - 1) \oplus M_i$
return $C_0 \| C_1 \| \cdots \| C_\ell$

CTR mode: a closer look



Source: The Joy of Cryptography

Section 2

Chosen-Ciphertext Attacks

Introducing chosen-ciphertext attacks

- Imagine this scenario at your company:
  - You discover a bug: software crashes when decrypting ciphertexts that yield plaintexts with null characters.
  - You think: “We never encrypt messages with null bytes, so no problem!”
- An attacker can:
  - Send specially crafted ciphertexts to your system
  - Observe which ones crash your software (contain null bytes) and which don’t
  - Use this information to completely break your encryption scheme!

Introducing chosen-ciphertext attacks

- This is a **chosen-ciphertext attack** (CCA):
  - Attacker can submit arbitrary ciphertexts for decryption.
  - System leaks information about the decryption results (the crash).
  - Even small information leaks can completely compromise security.
  - All encryption schemes we've seen so far are vulnerable to this!

Format-oracle attacks & malleability

- Scenario: victim using CTR mode decryption reveals if result contains a null character.
- Information can leak through crashes, error messages, or behavior differences.
- A single yes/no bit of information can be enough to break encryption!
- Malleability: many encryption schemes allow attackers to make predictable changes to plaintexts by modifying ciphertexts.
- CTR mode is especially vulnerable because XOR allows targeted bit-flipping.

$\underline{\text{NULLORACLE}(C)}$:
$M := \texttt{CTR.Dec}(K, C)$
if M contains `00` character:
    return `true`
else return `false`

Why is it called $\mathrm{nulloracle}(c)$?

- In mythology and colloquial language:
  - An oracle was a mystic who answered questions on behalf of the gods.
  - Often gave enigmatic answers requiring interpretation.

The Oracle by Camillo Miola, 1880[a]

[a] Absolutely beautiful painting, well-worth viewing in high resolution: https://upload.wikimedia.org/wikipedia/commons/9/94/Camillo_Miola_%28Biacca%29_-_The_Oracle_-_72.PA.32_-_J._Paul_Getty_Museum.jpg

Why is it called $\mathrm{nulloracle}(c)$?

- In cryptography and computer science:
  - An oracle is an algorithm that solves a specific problem.
  - You can query it but can't see its internal workings.
  - Oracles are often used as theoretical tools in security proofs.
- Our $\textsc{nulloracle}(c)$ reveals only:
  - Whether a decrypted ciphertext contains null (`00`) characters.
  - Just a single bit of information (yes/no).

Null-oracle attack

- An adversary who has access to $\text{NULLORACLE}(C)$ can efficiently compute $\texttt{Dec}(K, C)$ for any C!
- Yes, it really is enough.

$\text{NULLORACLE}(C)$:
$M := \texttt{CTR.Dec}(K, C)$
if M contains `00` character:
    return `true`
else return `false`

CTR mode is malleable

Defining malleability

Malleability

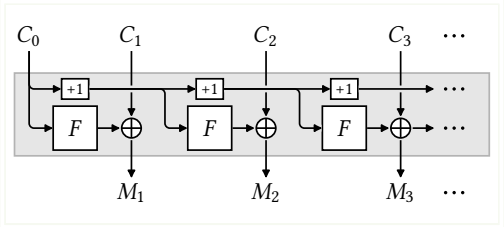
An encryption scheme is **malleable** if, given an encryption C of an unknown plaintext M, it is possible to create a new ciphertext $C' \neq C$ where $M' = \mathsf{Dec}(K, C')$ is somehow related to M, so that M' reveals some information about M.

- In CTR mode: $C_i = F(K, C_0 + i - 1) \oplus M_i$
- Flipping a bit in C_i flips exactly the same bit in M_i.
- Attackers can make targeted modifications without knowing the key.
- Example: change “transfer $100” to “transfer $900” by modifying just one byte.

CTR mode is malleable

Defining malleability

- In CTR mode:
  $C_i = F(K, C_0 + i - 1) \oplus M_i$
- Flipping a bit in C_i flips exactly the same bit in M_i.
- Attackers can make targeted modifications without knowing the key.
- Example: change “transfer $100” to “transfer $900” by modifying just one byte.

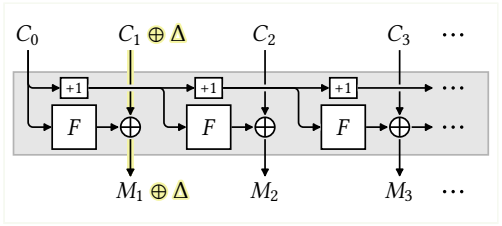


CTR mode decryption.
Source: The Joy of Cryptography

CTR mode is malleable

Defining malleability

- In CTR mode:
  $C_i = F(K, C_0 + i - 1) \oplus M_i$
- Flipping a bit in C_i flips exactly the same bit in M_i.
- Attackers can make targeted modifications without knowing the key.
- Example: change “transfer $100” to “transfer $900” by modifying just one byte.



CTR mode decryption.
Source: The Joy of Cryptography

CTR mode is malleable

The attack

- Suppose adversary captures ciphertext C and wants to find plaintext $M = \texttt{Dec}(K, C)$.
- Let's focus on discovering just the last byte of M.
- If the last byte of M is b, then $M = M' \| b$.
- What if we modify the ciphertext by XORing the last byte with b?
- This would turn the last byte of the plaintext into all zeros (null byte)!

$C \oplus (\ldots \texttt{00} \| b)$ decrypts to:

$$
\begin{aligned}
& M \oplus (\ldots \texttt{00} \| b) \\
&= (M' \| b) \oplus (\ldots \texttt{00} \| b) \\
&= M' \| (b \oplus b) \\
&= M' \| \texttt{00}
\end{aligned}
$$

CTR mode is malleable

Finding one byte

- Problem: We don't know b in advance!
- Solution: Try all 255 possible values for b.
- For each guess g, we:
  - Create $C' = C \oplus (\ldots \texttt{00} \| g)$
  - Send C' to the $\textsc{nulloracle}()$
  - If oracle returns `true`, then $g = b$!
- When we find the right value, we've discovered the last byte of M.

Try these ciphertexts:

$$C \oplus (\ldots \texttt{00} \| \texttt{01})$$
$$C \oplus (\ldots \texttt{00} \| \texttt{02})$$
$$C \oplus (\ldots \texttt{00} \| \texttt{03})$$
$$\vdots$$
$$C \oplus (\ldots \texttt{00} \| \texttt{fe})$$
$$C \oplus (\ldots \texttt{00} \| \texttt{ff})$$

CTR mode is malleable

Finding all bytes

- We can use the same approach for any byte in the plaintext.
- To find the i-th byte:
  - Create a string of all zeros
  - Set only the i-th byte to our guess g
  - XOR this with the ciphertext
  - Query the oracle
- By repeating for all bytes, we can recover the entire plaintext!

$\mathcal{A}_{\text{null-oracle}}$

$n := |C_1\| \cdots \|C_\ell|$
$M := ""$

for $i = 1$ to n:
  for each $b \in \{\texttt{01}, \ldots, \texttt{ff}\}$:
    $\Delta := (\texttt{00})^n$
    $\Delta[i] := b$
    if $\text{NULLORACLE}(C \oplus (0^\lambda \| \Delta))$:
      $M := M \| b$
      next i
return M

CTR mode is malleable

The power of chosen-ciphertext attacks

- With just 255 queries per byte, we can completely decrypt any ciphertext!
- For context: decrypting a 1KB file would take about 250,000 queries.
- This is extremely practical for an attacker.
- All from a single bit of information about the plaintext (contains null or not).
- This attack works because:
  - CTR mode is malleable (we can make predictable changes to plaintext)
  - The system leaks a tiny bit of information about decrypted plaintexts
  - Together, these flaws completely break the encryption

Is the null-oracle attack just brute-force?

- Yes and no!
- It's brute-force on each byte independently:
  - To recover n-byte plaintext: at most $255n$ oracle queries
  - True brute-force on entire plaintext: 255^n (exponentially worse!)
- For a 16-byte message:
  - Null-oracle attack: 4,080 queries (16 ×255)
  - True brute-force: 10^{38} queries (255^{16})
- This attack is exponentially more efficient than traditional brute-force.

Can we just rate-limit the number of queries?

- Rate-limiting might help, but:
  - It only increases attack time, doesn't prevent it.
  - Attackers can be patient or use multiple accounts.
  - Legitimate users suffer from the rate-limiting.
- Better approach: cryptographic solutions!
  - Fix the fundamental vulnerability, not just limit its exploitation.
  - Create systems that are mathematically proven to resist chosen-ciphertext attacks.

Is fixing the null-byte bug enough?

- The null-character bug is just one example of a broader class of attacks.
- Format-oracle attacks can exist without implementation bugs!
- They only need:
  - A system that accepts untrusted ciphertexts
  - Decrypts them
  - Behaves differently based on the decryption result
  - In a way the attacker can observe
- This behavior can come from:
  - Accidental bugs (null-byte crashes)
  - Intentional features (error messages, timing differences)
  - Normal application logic (web app behaving differently based on decrypted data)

Real-world format-oracle attacks

Padding oracle

- CBC mode requires padding to handle plaintexts that aren't block-aligned.
- Many implementations crash when encountering invalid padding.
- This exposes an oracle that tells attackers: "Does $\mathtt{Dec}(K, C)$ have valid padding?"
- Attackers can systematically exploit this to decrypt arbitrary ciphertexts.
- Has led to major vulnerabilities in SSH and SSL/TLS protocols.
- Example: POODLE attack against SSL 3.0 affected millions of websites.[a]

[a] https://appliedcryptography.page/papers/#google-poodle

Real-world format-oracle attacks

Timing side-channel

- Victim interprets plaintext as a number n and performs n operations.
- Attackers can measure how long the system takes to respond.
- Response time reveals approximate numerical values inside $\texttt{Dec}(K, C)$.
- Extremely subtle - even microsecond differences can leak information.
- Successfully used to break older SSH and SSL/TLS implementations.
- Example: Lucky Thirteen attack against TLS revealed message contents through timing differences.[a]

[a] https://appliedcryptography.page/papers/#lucky-thirteen

Real-world format-oracle attack

Apple iMessage

- Older Apple iMessage versions used gzip compression.
- System responded differently when a ciphertext decrypted to:
  - A valid gzip file (processed normally)
  - An invalid gzip file (error reported)
- This created an oracle revealing: "Is $\mathsf{Dec}(K, C)$ a valid gzip file?"[a]
- Attackers who understood the gzip format could exploit this to:
  - Silently recover private messages
  - Bypass encryption entirely

[a] https://appliedcryptography.page/papers/#jhu-imessage

Real-world format-oracle attack

XML format Oracles

- Many systems decrypt data expecting valid XML format.
- If decrypted data isn't valid XML, system returns an error.
- This exposes an oracle: "Is $\mathtt{Dec}(K, C)$ valid XML?"
- XML has complex syntax rules that attackers can exploit.
- Can allow complete decryption of arbitrary ciphertexts.
- Similar attacks exist for other formats (JSON, HTML, etc.)

I thought CTR mode was secure?

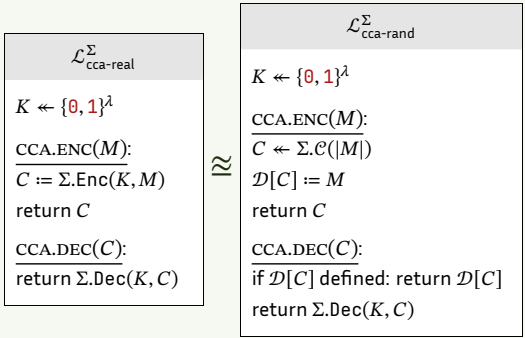
CPA vs. CCA security

- CTR mode *is* secure against chosen-plaintext attacks (CPA-secure).
  - It uses randomness to ensure identical messages encrypt differently each time.
  - The adversary cannot distinguish encryptions of known plaintexts.
- But CPA security isn't enough in many real-world scenarios!
  - CPA security only considers attackers who can request encryptions.
  - It doesn't protect against attackers who can submit chosen ciphertexts.
- In the null-oracle attack:
  - The victim decrypts ciphertexts chosen by the adversary.
  - Even leaking one bit about the plaintext (contains nulls or not) is fatal.
  - CPA security doesn't model or prevent this type of attack.
- This is why we need stronger security notions (CCA security).

CCA Security

Security Against Chosen-Ciphertext Attacks (CCA Security)

An encryption scheme Σ has security against **chosen-ciphertext attacks** if the following two libraries are indistinguishable:



Remember our CPA-secure encryption scheme?

- Not CCA-secure!

$$\underline{\text{ENC}(K, M):}$$
$$R \twoheadleftarrow \{0,1\}^{\lambda}$$
$$S := F(K, R) \oplus M$$
$$\text{return } R \| S$$

Remember our CPA-secure encryption scheme?

- Not CCA-secure!
- In other words, we can trivially distinguish between these libraries:

$\mathcal{L}_{\text{cca-real}}$

$K \twoheadleftarrow \{0,1\}^\lambda$

$\underline{\text{CCA.ENC}(M)}$:
$R \twoheadleftarrow \{0,1\}^\lambda$
$S := F(K,R) \oplus M$
return $R \| S$

$\underline{\text{CCA.DEC}(R \| S)}$:
$M := F(K,R) \oplus S$
return M

$\approx$

$\mathcal{L}_{\text{cca-rand}}$

$K \twoheadleftarrow \{0,1\}^\lambda$

$\underline{\text{CCA.ENC}(M)}$:
$R \| S \twoheadleftarrow \{0,1\}^{2\lambda}$
$\mathcal{D}[R \| S] := M$
return $R \| S$

$\underline{\text{CCA.DEC}(R \| S)}$:
if $\mathcal{D}[R \| S]$ defined: return $\mathcal{D}[R \| S]$
$M := F(K,R) \oplus S$
return M

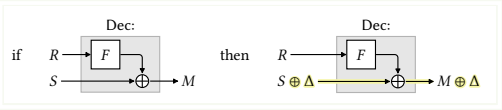
Malleability strategy

To break the CCA security of a scheme:

1. Study the decryption algorithm of the scheme. It often helps to draw a schematic diagram.
2. See whether any changes to a ciphertext make a *predictable* change to the plaintext.
3. Formalize an attack in which the adversary:
   - 3.1 Requests the encryption of a chosen plaintext,
   - 3.2 Modifies the ciphertext as above,
   - 3.3 Asks for the modified ciphertext to be decrypted.

Malleability once again

- Not CCA-secure!
- Here's a distinguisher:



Source: The Joy of Cryptography

$\mathcal{A}$

$M \twoheadleftarrow \{0,1\}^\lambda$

$\Delta :=$ arbitrary, nonzero, λ-bit string

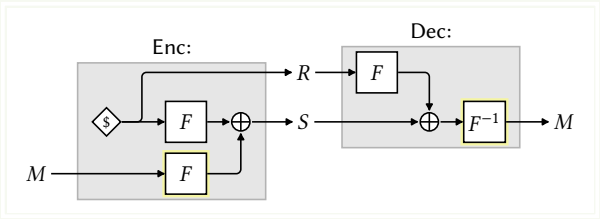
$R \| S := \text{CCA.ENC}(M)$

$M' := \text{CCA.DEC}(R \| (S \oplus \Delta))$

return $M' == M \oplus \Delta$

Another example

- Let's try a harder challenge.



Source: The Joy of Cryptography

$\mathcal{L}_{\text{cca-real}}$

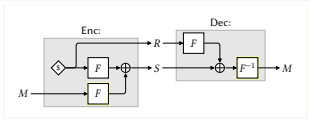
$K \twoheadleftarrow \{0,1\}^\lambda$

$\underline{\text{CCA.ENC}(M)}$:
$R \twoheadleftarrow \{0,1\}^\lambda$
$S := F(K,R) \oplus F(K,M)$
return $R \| S$

$\underline{\text{CCA.DEC}(R \| S)}$:
$M := F^{-1}(K, F(K,R) \oplus S)$
return M

Another example

- Let's try a harder challenge.



Source: The Joy of Cryptography

$\mathcal{L}_{\text{cca-real}}$	$\mathcal{L}_{\text{cca-rand}}$
$K \twoheadleftarrow \{0,1\}^\lambda$	$K \twoheadleftarrow \{0,1\}^\lambda$
$\underline{\text{CCA.ENC}(M)}$:	$\underline{\text{CCA.ENC}(M)}$:
$R \twoheadleftarrow \{0,1\}^\lambda$	$R\|S \twoheadleftarrow \{0,1\}^{2\lambda}$
$S \coloneqq F(K,R) \oplus F(K,M)$	return $R\|S$
return $R\|S$	$\underline{\text{CCA.DEC}(R\|S)}$:
$\underline{\text{CCA.DEC}(R\|S)}$:	if $\mathcal{D}[R\|S]$ defined: return $\mathcal{D}[R\|S]$
$M \coloneqq F^{-1}(K,F(K,R) \oplus S)$	$M \coloneqq F^{-1}(K,F(K,R) \oplus S)$
return M	return M

Frankenstein strategy

Try the following approach in a chosen-ciphertext attack:

1. Request two separate encryptions of chosen plaintexts; it often helps to use the same plaintext.
2. Mix and match parts of the resulting ciphertexts to obtain two Frankenstein ciphertexts.
3. Ask for the Frankenstein ciphertexts to be decrypted, and see whether anything interesting happens.

Another example

- Not CCA-secure!
- Here's a distinguisher:

$\mathcal{A}$

$M \twoheadleftarrow \{0,1\}^\lambda$

$R_1 \| S_1 \coloneqq \text{CCA.ENC}(M)$

$R_2 \| S_2 \coloneqq \text{CCA.ENC}(M)$

$M_1^* \coloneqq \text{CCA.DEC}(R_1 \| S_2)$

$M_2^* \coloneqq \text{CCA.DEC}(R_2 \| S_1)$

return $M_1^* == M_2^*$

Section 2: Chosen-Ciphertext Attacks

Subsection 2.1

Message Authentication Codes

Symmetric primitive example: hash functions

Reminder

Hash Function Properties

- Takes input of **any size**[<+->]
- Produces output of **fixed size**
- Is **deterministic** (same input → same output)
- Even a **tiny change** in input creates completely different output
- Is **efficient** to compute

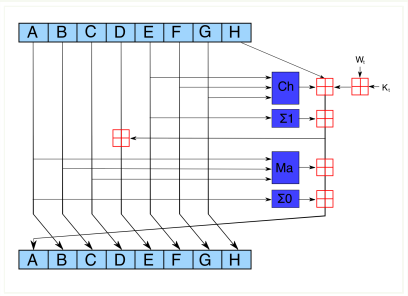
```
SHA256(hello) =
2cf24dba5fb0a30e26e83b2ac5
b9e29e1b161e5c1fa7425e7304
3362938b9824
SHA256(hullo) =
7835066a1457504217688c8f5d
06909c6591e0ca78c254ccf174
50d0d999cab0
```

Note: One character change → completely different hash!

Expected properties of a hash function

Reminder

- **Collision resistance**: computationally infeasible to find two different inputs producing the same hash.
- **Preimage resistance**: given the output of a hash function, it is computationally infeasible to reconstruct the original input.
- **Second preimage resistance**: given an input and an output, it's computationally infeasible to find another different input producing the same output.



SHA-2 compression function. Source: Wikipedia

Hash functions: what are they good for?

Reminder

- **Password storage**: Store the hash of the password on the server, not the password itself. Then check candidate passwords against the hash.
- **Data integrity verification**: Hash a file. Later hash it again and compare hashes to check if the file has changed, suffered storage degradation, etc.
- **Proof of work**: Server asks client to hash something a lot of times before they can access some resource. Useful for anti-spam, Bitcoin mining, etc.

Message authentication codes

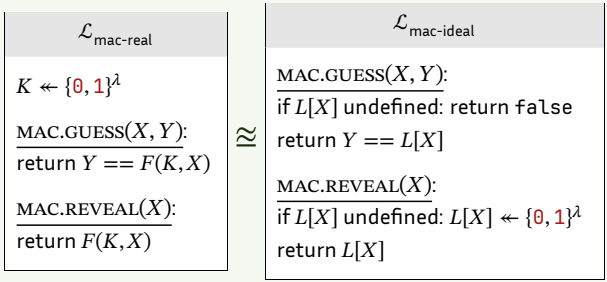
Message Authentication Code (MAC)

A MAC is a cryptographic function that takes a key K and a message M and produces a tag T that authenticates the message. Only someone with the same key can verify the tag.

- A MAC provides **integrity** and **authenticity** for messages.
- Unlike hash functions, MACs require a secret key
- MACs address the malleability problem we saw with encryption schemes. Without a MAC, attackers could modify ciphertexts.
- A secure MAC should be unforgeable, even after seeing MACs for chosen messages.

PRFs as MACs

- A pseudorandom function (PRF) can be used directly as a MAC!
- The MAC key is the PRF key K.
- To authenticate a message X:
  - Compute the tag $T = F(K, X)$
  - Send both X and T to the recipient



From CPA to CCA security

The Encrypt-then-MAC approach

- CPA security isn't enough in the real world.
- We need protection against chosen-ciphertext attacks.
- Solution: combine encryption with a MAC!.
- **Encrypt-then-MAC**:
  - Encrypt the message normally.
  - Compute a MAC tag of the *ciphertext*.
  - Send both ciphertext and tag.
  - Receiver verifies the tag before decrypting.
- This prevents adversaries from creating valid modified ciphertexts.

Encrypt-then-MAC: formal construction

Encrypt-then-MAC Construction

Let Σ be an SKE scheme and F be a PRF with output length λ whose domain includes $\Sigma.\mathcal{C}$. Define a new encryption scheme:

$$\mathcal{K} = \Sigma.\mathcal{K} \times \{0,1\}^\lambda$$
$$\mathcal{M} = \Sigma.\mathcal{M}$$
$$\mathcal{C}(\ell) = \Sigma.\mathcal{C}(\ell) \times \{0,1\}^\lambda$$

$\underline{\text{ENC}((K_e, K_m), M)}$:
$C \coloneqq \Sigma.\text{Enc}(K_e, M)$
$T \coloneqq F(K_m, C)$
return $C \| T$

$\underline{\text{DEC}((K_e, K_m), C\|T)}$:
if $F(K_m, C) \neq T$: return err
return $\Sigma.\text{Dec}(K_e, C)$

CCA security of Encrypt-then-MAC

CCA Security Claim

If Σ is a CPA-secure encryption scheme and F is a secure PRF, then the Encrypt-then-MAC construction is CCA-secure.

- Key insight: **the MAC prevents tampering**
- Without the MAC key, adversary can't create valid tags.
- **Decryption oracle only returns plaintexts for ciphertexts with valid tags.**

$\mathcal{L}_{\text{cca-real}}$

$K_e \twoheadleftarrow \Sigma.\mathcal{K}$
$K_m \twoheadleftarrow \{0,1\}^\lambda$

$\underline{\text{CCA.ENC}(M)}$:
$C \coloneqq \Sigma.\text{Enc}(K_e, M)$
$T \coloneqq F(K_m, C)$
return $C \| T$

$\underline{\text{CCA.DEC}(C \| T)}$:
if $F(K_m, C) \neq T$: return err
return $\Sigma.\text{Dec}(K_e, C)$

$\approx$

$\mathcal{L}_{\text{cca-rand}}$

$K_e \twoheadleftarrow \Sigma.\mathcal{K}$
$K_m \twoheadleftarrow \{0,1\}^\lambda$

$\underline{\text{CCA.ENC}(M)}$:
$C \twoheadleftarrow \Sigma.\mathcal{C}(|M|)$
$T \twoheadleftarrow \{0,1\}^\lambda$
$\mathcal{D}[C \| T] \coloneqq M$
return $C \| T$

$\underline{\text{CCA.DEC}(C \| T)}$:
if $\mathcal{D}[C \| T]$ defined: return $\mathcal{D}[C \| T]$
if $F(K_m, C) \neq T$: return err
return $\Sigma.\text{Dec}(K_e, C)$

Combining MACs and encryption

- Not every way of combining a MAC and CPA-secure encryption achieves CCA security.
- There are three common approaches to combining them:
  - **Encrypt-then-MAC**: Encrypt message, then MAC the ciphertext.
  - **Encrypt-and-MAC**: Encrypt message and MAC the plaintext separately.
  - **MAC-then-encrypt**: MAC the plaintext, then encrypt both message and tag.
- Only one approach guarantees CCA security when using any CPA-secure encryption.

Encrypt-then-MAC

CCA-secure

- **MAC verifies ciphertext integrity** before decryption.
- Prevents attackers from submitting modified ciphertexts.
- **Always CCA-secure** if encryption is CPA-secure and MAC is secure.

$\underline{\textsc{Enc}((K_e, K_m), M)}$:
$C \coloneqq \Sigma.\mathsf{Enc}(K_e, M)$
$T \coloneqq F(K_m, C)$
return $C \| T$

$\underline{\textsc{Dec}((K_e, K_m), C \| T)}$:
if $F(K_m, C) \neq T$: return `err`
return $\Sigma.\mathsf{Dec}(K_e, C)$

Encrypt-and-MAC

Not even CPA-secure!

- **MAC is computed on the plaintext**
- Same plaintext always produces same tag, leaking equality information.
- **Not even CPA-secure**, let alone CCA-secure.

$\underline{\textsc{Enc}((K_e, K_m), M):}$
$C \coloneqq \Sigma.\mathsf{Enc}(K_e, M)$
$T \coloneqq F(K_m, M)$
return $C \| T$

$\underline{\textsc{Dec}((K_e, K_m), C\|T):}$
$M \coloneqq \Sigma.\mathsf{Dec}(K_e, C)$
if $F(K_m, M) \neq T$: return $\mathtt{err}$
return M

MAC-then-encrypt

It's complicated

- **Tag is hidden inside the ciphertext**
- Whether this is CCA-secure depends on the specific encryption scheme.
- **Not generally CCA-secure** for all CPA-secure encryption schemes.

$\underline{\text{Enc}((K_e, K_m), M)}$:
$T \coloneqq F(K_m, M)$
$C \coloneqq \Sigma.\text{Enc}(K_e, M \| T)$
return C

$\underline{\text{Dec}((K_e, K_m), C)}$:
$M \| T \coloneqq \Sigma.\text{Dec}(K_e, C)$
if $F(K_m, M) \neq T$: return err
return M

Encrypt + MAC security comparison

Construction	CPA-secure?	CCA-secure?
Encrypt-then-MAC	Yes	Yes
Encrypt-and-MAC	No	No
MAC-then-encrypt	Yes	Maybe

- **Encrypt-then-MAC** is the safest option.
- **Encrypt-and-MAC** should never be used.
- **MAC-then-encrypt** requires case-by-case analysis.

Section 2: Chosen-Ciphertext Attacks

Subsection 2.2

Authenticated Encryption

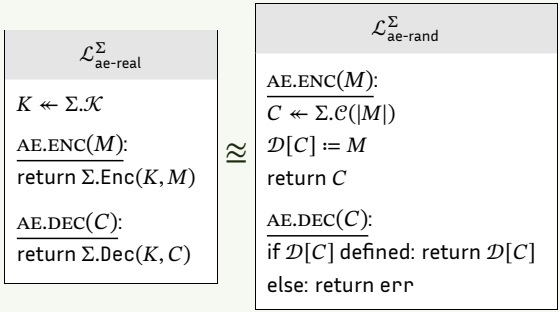
Authenticated Encryption: beyond CCA security

- CCA security is stronger than CPA security, but still not the gold standard.
- CCA security says: adversary-generated ciphertexts won't reveal useful information.
- But CCA security doesn't require that they decrypt to `err`.
- For many applications, we want a stronger guarantee:
  - Only key-holders can create valid ciphertexts.
  - All other ciphertexts should be rejected as invalid.
- This property is called **Authenticated Encryption (AE)**.
- Authenticated encryption provides both confidentiality and authenticity.

Authenticated Encryption: formal definition

Authenticated Encryption

A SKE scheme Σ is a secure authenticated encryption (AE) scheme if the following two libraries are indistinguishable:



AE vs. CCA security

- Key difference: **how we handle adversary-created ciphertexts**.
- In $\mathcal{L}^{\Sigma}_{\text{ae-rand}}$, any ciphertext not created by the library always decrypts to `err`.
- In $\mathcal{L}^{\Sigma}_{\text{cca-rand}}$, such ciphertexts could decrypt to anything (not necessarily `err`).
- So AE requires:
  - Adversary cannot tell real from random ciphertexts (as in CPA)
  - Adversary cannot create new valid ciphertexts (authentication)

- AE is **strictly stronger** than CCA security.
- Every AE scheme is CCA-secure, but not every CCA-secure scheme is an AE.
- Making the distinction explicit helps us design better protocols.
- AE is what you should aim for in practice.

How Encrypt-then-MAC achieves AE

- Remember our Encrypt-then-MAC construction:
  - Encrypt the plaintext: $C := \Sigma.\mathsf{Enc}(K_e, M)$
  - MAC the ciphertext: $T := F(K_m, C)$
  - Send both: $C \| T$
- It achieves AE security because:
  - Without K_m, adversary can't forge valid tags.
  - Any ciphertext not created by the system will fail MAC verification.
  - MAC verification failures lead to $\mathtt{err}$.
- The proof is nearly identical to CCA security proof.

AE Security of Encrypt-then-MAC

Encrypt-then-MAC is a secure AE, if the underlying Σ is a CPA-secure SKE and F is a secure PRF.

AE implies CCA

If an encryption scheme Σ is a secure AE, then it is also CCA-secure.

Authenticated Encryption: in practice

- Modern cryptographic protocols almost always use authenticated encryption.
- Common AE implementations:
  - **AES-GCM** (Galois/Counter Mode): most widely used, combines CTR mode with a MAC.
  - **ChaCha20-Poly1305**: popular alternative to AES-GCM, especially on devices without AES hardware.
  - **AES-CBC + HMAC-SHA256**: older approach, uses Encrypt-then-MAC with AES in CBC mode.
- Important implementation rule: **verify before decrypt**!
  - Always check the MAC before decrypting.
  - Prevents timing side-channels based on decryption behavior.
  - Helps protect against padding oracle attacks and similar vulnerabilities.

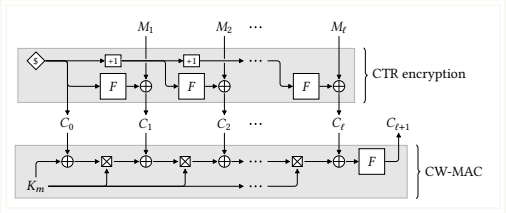
AES-GCM: Galois/Counter Mode

- AES-GCM (Galois/Counter Mode) is the most widely used AEAD scheme.
- Combines AES in CTR mode (for encryption) with GMAC (for authentication).
- Extremely efficient:
  - Single pass over the data.
  - Parallelizable.
  - Hardware acceleration widely available.
- Used in: TLS 1.2/1.3, IPsec, SSH, and many other protocols.

$\underline{\text{AES-GCM.Enc}(K, N, A, M)}$:
$H \coloneqq \mathtt{AES}(K, 0^{128})$
for $i = 0$ to $\lceil |M|/128 \rceil - 1$:
  $C_i \coloneqq M_i \oplus \mathtt{AES}(K, N \| i)$
$T \coloneqq \mathtt{GHASH}_H(A, C) \oplus$
  $\mathtt{AES}(K, N \| 0)$
return $C \| T$

AES-GCM: Galois/Counter Mode

- **Inputs**:
  - Key K (128, 192, or 256 bits)
  - Nonce N (usually 96 bits)
  - Associated data A (optional)
  - Plaintext M
- **Encryption process**:
  - AES-CTR for confidentiality.
  - GHASH (Galois field multiplication) for authentication.
- Authentication tag T protects both ciphertext and associated data.



Source: The Joy of Cryptography

AES-GCM: Galois/Counter Mode

- **Security strengths**:
  - Provides confidentiality, integrity, and authenticity
  - Formally proven secure (assuming AES is secure)
  - Fast and widely trusted
- **Critical implementation requirements**:
  - **Never reuse a nonce** with the same key!
  - A repeated nonce can lead to complete loss of confidentiality and authentication
  - 96-bit nonces are recommended (other sizes are less efficient)
  - Authentication tag should be at least 128 bits long

Key commitment in authenticated encryption

- **Key commitment**: a ciphertext should only decrypt to a valid plaintext under the key used to generate it.
- Most AEAD schemes (including AES-GCM) don't guarantee this property![a]
- Attack scenario:
  1. Attacker creates special ciphertext C.
  2. When decrypted with key K_1: harmless message.
  3. When decrypted with key K_2: malicious content!
  4. Enables plausible deniability, content smuggling, etc.
- Practical impact:
  - Attacker can create ciphertexts that decrypt differently under different keys.
  - Enables attacks in multi-recipient contexts.
  - Affects real applications (e.g., messaging, encrypted files).

[a] https://appliedcryptography.page/papers/#key-commitment

ChaCha20-Poly1305

- ChaCha20-Poly1305 is a modern AEAD construction:
  - ChaCha20 stream cipher for encryption.
  - Poly1305 MAC for authentication.
- Designed by Daniel J. Bernstein.[a]
- Key characteristics:
  - No table lookups (better resistance to timing attacks)
  - Excellent performance on devices without AES hardware.
- Widely used in TLS 1.3, Signal, WireGuard...

$\textsc{ChaCha20-Poly1305.Enc}(K, N, A, M)$:

$\text{key}_p \coloneqq \texttt{ChaCha20}(K, N, 0)_{0..31}$

$C \coloneqq M \oplus \texttt{ChaCha20}(K, N, 1)$

$\text{data} \coloneqq \text{pad}(A) \| \text{pad}(C) \| \text{len}(A) \| \text{len}(C)$

$T \coloneqq \texttt{Poly1305}_{\text{key}_p}(\text{data})$

return $C \| T$

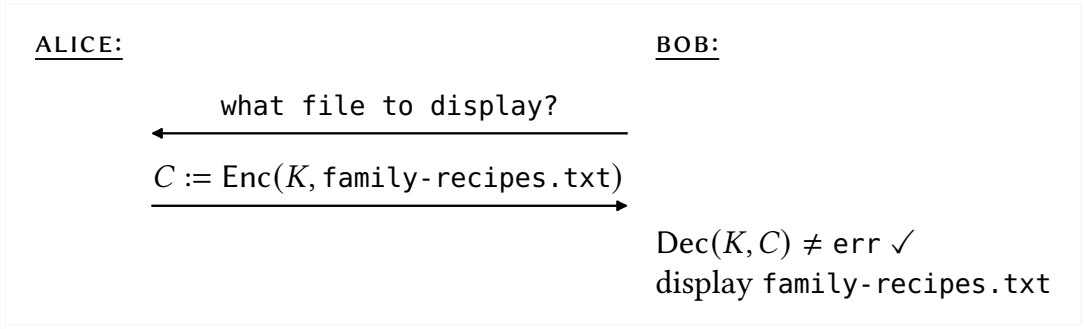
[a] Does the class want to hear about Bernstein vs. United States?

Comparing AEAD implementations

Property	AES-GCM	ChaCha20-Poly1305	AES+HMAC
Performance (HW accel.)	Excellent	Good	Good
Performance (no accel.)	Poor	Excellent	Moderate
Security level	128-256 bits	256 bits	128-256 bits
Side-channel resistance	Moderate	Excellent	Moderate
Parallelizable	Yes	Partially	No
Nonce sensitivity	Very high	High	Moderate
Overhead	Low (single pass)	Low (single pass)	Higher (two pass)
Implem. complexity	Moderate	Low	High

Replay attacks

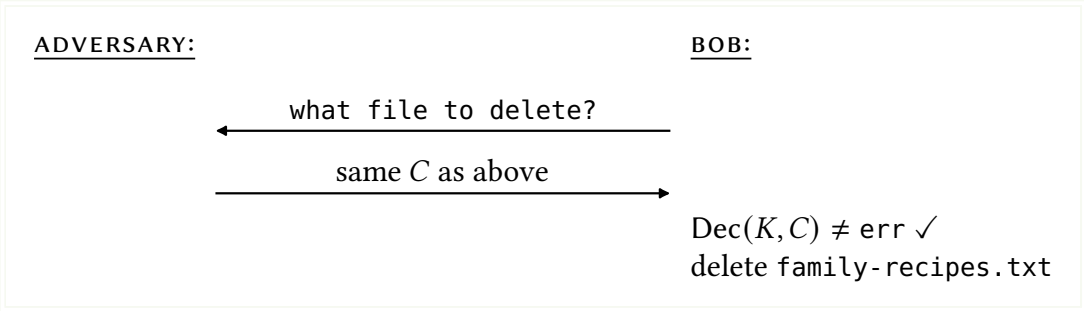
Step 1: Alice sends original message



Source: The Joy of Cryptography

Replay attacks

Step 2: Attacker replays message in a different context



Source: The Joy of Cryptography

Replay attacks

Authenticated encryption didn't save us

- Even with authenticated encryption, context matters!
- Scenario: Alice sends Bob encrypted commands.
  - Each ciphertext contains Alice's genuine intent.
  - Bob trusts any ciphertext that decrypts successfully.
- Vulnerability: An adversary can replay legitimate ciphertexts.
  - Alice once sent "Delete temporary files".
  - Adversary replays it when Alice meant to say "Display files".

Associated data

- Solution 1: Include context in the plaintext
  - "`ACTION: DISPLAY`" before the actual message.
  - Inefficient - increases message size.
  - Both parties already know the context.
- Better solution: Associated Data (AD)
  - Context that sender and receiver already know.
  - Used during encryption and decryption.
  - Doesn't increase ciphertext size.
- How it works:
  - $\mathsf{Enc}(K, A, M) \to C$ where A is associated data.
  - $\mathtt{Dec}(K, A, C) \to M$ or `err`

AEAD: formal definition

AEAD

An encryption scheme Σ with associated data is a secure authenticated encryption with associated data (AEAD) scheme if the following two libraries are indistinguishable:

$\mathcal{L}^{\Sigma}_{\text{aead-real}}$

$K \twoheadleftarrow \Sigma.\mathcal{K}$

$\underline{\text{AEAD.ENC}(A, M)}$:
return $\Sigma.\text{Enc}(K, A, M)$

$\underline{\text{AEAD.DEC}(A, C)}$:
return $\Sigma.\text{Dec}(K, A, C)$

$\approx$

$\mathcal{L}^{\Sigma}_{\text{aead-rand}}$

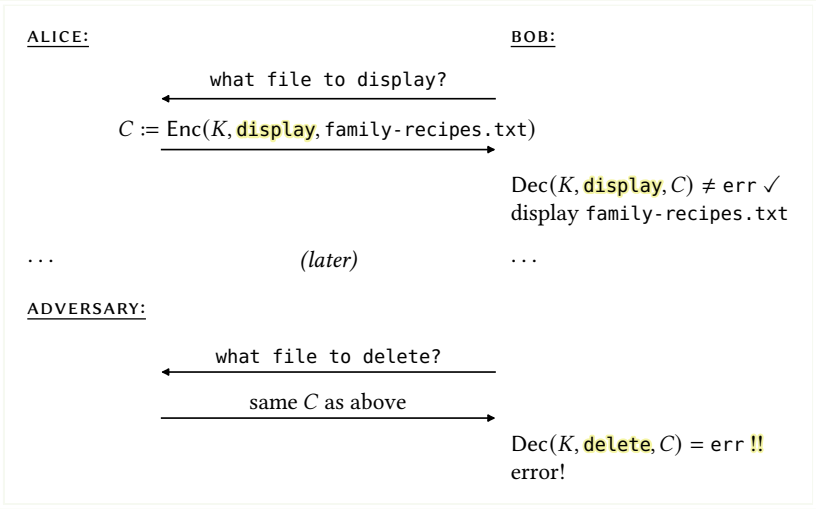
$\underline{\text{AEAD.ENC}(A, M)}$:
$C \twoheadleftarrow \Sigma.\mathcal{C}(|M|)$
$\mathcal{D}[A, C] \coloneqq M$
return C

$\underline{\text{AEAD.DEC}(A, C)}$:
if $\mathcal{D}[A, C]$ defined:
  return $\mathcal{D}[A, C]$
else: return err

Usefulness of AEADs

- AEAD (Authenticated Encryption with Associated Data) scheme guarantees:
  - Ciphertexts reveal nothing about plaintexts (confidentiality).
  - Only ciphertexts created by the legitimate sender will decrypt without error (authenticity).
  - Decryption only succeeds when the correct associated data is used (context binding).
- What to use as associated data?
  - Protocol information: “`DISPLAY`” vs “`DELETE`”.
  - Session identifiers or timestamps.
  - Previous messages in the conversation.
  - Any contextual information both parties already know.
- Use as much associated data as relevant - it’s cryptographically “free”!

Solution: use associated data to provide context



Source: The Joy of Cryptography

AMERICAN UNIVERSITY OF BEIRUT

Applied Cryptography

CMPS 297AD/396AI

Fall 2025

Part 1: Provable Security

1.5: Chosen-Plaintext & Chosen-Ciphertext Attacks

Nadim Kobeissi

https://appliedcryptography.page