ISE Cryptography – Lecture 07

Digital Signatures, CCA Security, Authenticated Encryption and HPKE

Signatures vs MACs

The public-key alternative to message authentication.

The Comparison

- MACs and ***digital signatures*** both provide message integrity and authenticity
  - But they differ in a fundamental way
- MACs are symmetric: both parties share the same secret key
  - Any keyholder can produce and verify tags
  - If Alice and Bob share a key, Bob can verify Alice's tag, but he could also have produced it himself
  - No non-repudiation: Alice can deny sending a message, and Bob can't prove otherwise to a third party
- Digital signatures are asymmetric: only the signer has the secret key
  - Anyone with the public key can verify, but only the signer can produce
  - ***Non-repudiation***: once Alice signs a message, she can't credibly deny it
  - A judge, a server, or any third party can verify the signature independently
- MACs prove origin to someone who already trusts you
  - Signatures prove origin to anyone

Signatures as a Public-Key MAC

- A ***digital signature scheme*** $\mathcal{S} = (G, S, V)$ mirrors the MAC structure:
  - G: probabilistic key generation, outputs (pk, sk)
  - S: probabilistic signing, $\sigma \leftarrow S(sk, m)$
  - V: deterministic verification, $V(pk, m, \sigma)$ outputs `accept` or `reject`
- Correctness: $\Pr[V(pk, m, S(sk, m)) = \texttt{accept}] = 1$
- $\mathcal{S}$ is defined over $(\mathcal{M}, \Sigma)$
  - All messages m lie in a finite message space $\mathcal{M}$
  - All signatures σ lie in a finite signature space Σ
- Unlike a MAC, no shared secret is needed
  - You create a signature with your secret key
  - Anyone can verify that signature with your public key

Use Cases

- Software distribution: Microsoft signs update U with its secret key
  - Each customer checks $V(pk, U, \sigma) = \texttt{accept}$ before installing
  - Public key pk is obtained from the previous version, not from U
  - No shared keys, no trusted third party, just public key and signature!
- Authenticated email (DKIM): domains sign every outgoing email
  - The corresponding public key is published in a DNS record
  - Recipients can verify if an email from notascam.com is authentic
- Non-repudiation: once Alice signs, she can't claim she didn't send it
  - Anyone can verify the signature using her public key
  - This is stronger than MACs, and not always desirable
  - In law, electronic signature $\neq$ cryptographic digital signature

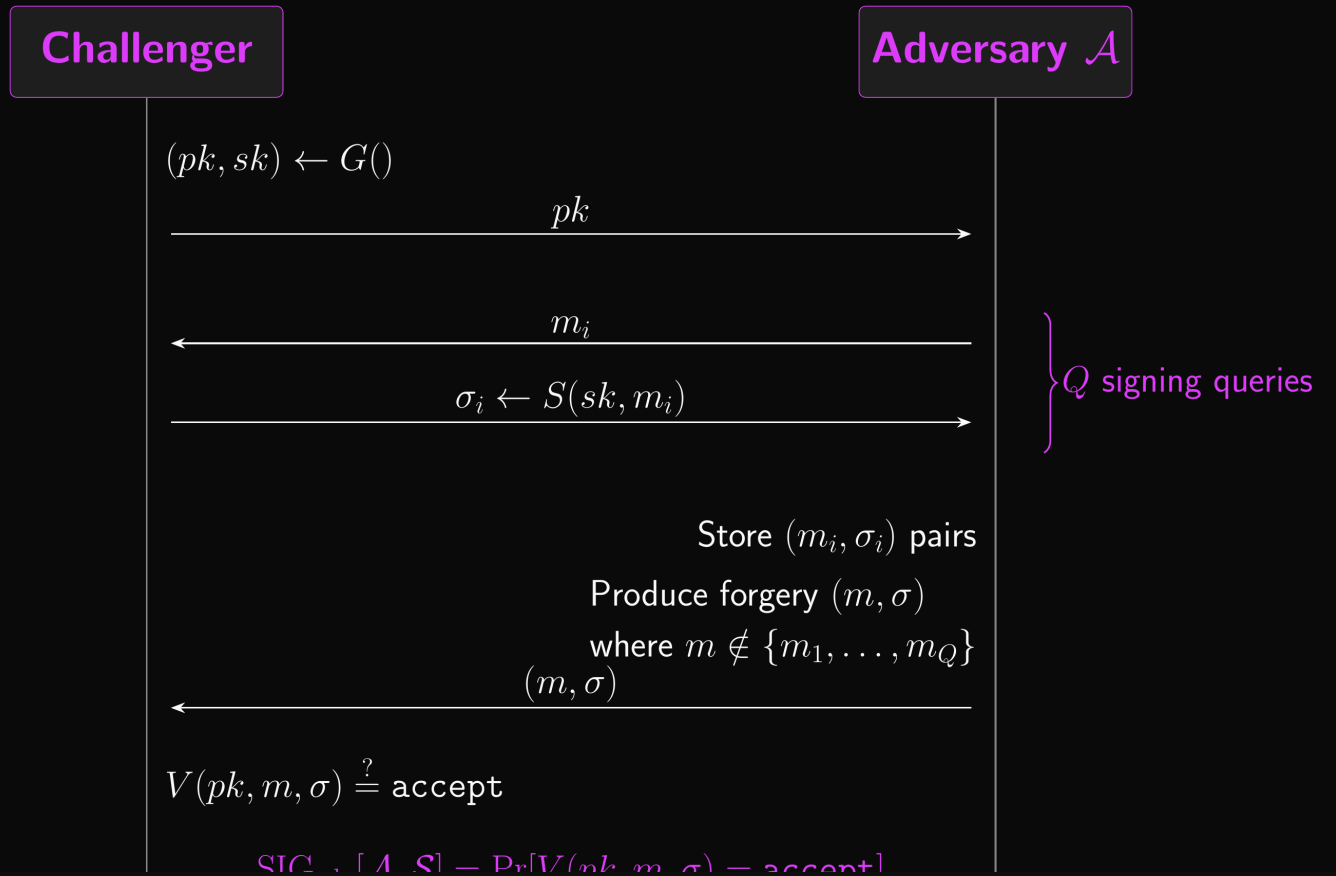
UF-CMA for Signatures

The same security framework, a new setting.

The Security Game

- The game is structurally identical to MAC UF-CMA from Lecture 04
  - The framework transfers: same game, same name, different setting
- The challenger runs $(pk, sk) \leftarrow G()$ and sends pk to $\mathcal{A}$
  - $\mathcal{A}$ is an efficient Q-query adversary
- $\mathcal{A}$ gets to query the challenger Q times
  - The ith signing query is a message $m_i \in \mathcal{M}$
  - The challenger computes $\sigma_i \leftarrow S(sk, m_i)$ and sends σ_i to $\mathcal{A}$
- Eventually, $\mathcal{A}$ outputs a candidate forgery pair (m, σ)
  - $\mathcal{A}$ wins if $V(pk, m, \sigma) = \texttt{accept}$ and m hasn't been queried before
- $\mathrm{SIG}_{\mathrm{adv}}[\mathcal{A}, \mathcal{S}]$ must be negligible for all efficient adversaries
- Secure signature schemes are ***existentially unforgeable under chosen message attack***

Signature Security: Attack Game



The Key Difference from MAC UF-CMA

- Unlike with MACs, we don't need to consider verification queries
  - $\mathcal{A}$ has the public verification key, so they can verify anything themselves!
- The adversary knows pk, which they can use in their forgery strategy
  - This is qualitatively different from the MAC setting where the key is fully secret
  - The adversary has strictly more information, yet must still be unable to forge
- ***Strongly secure*** signature schemes add a further constraint
  - $\mathcal{A}$ can't forge a new valid signature even for an already-queried message
  - Useful for specific proofs and protocols
  - Any secure scheme can be converted into a strongly secure one

Hash-and-Sign

- Public key signature schemes typically work on short inputs (***hash-and-sign*** extends the domain)
- Hash functions extend the domain, as usual
  - Given $\mathcal{S} = (G, S, V)$ over $(\mathcal{M}, \Sigma)$ where $\mathcal{M}$ is short
  - Let $H : \mathcal{M}' \to \mathcal{M}$ be a hash function (e.g. SHA-256)
  - $S'(sk, m) = S(sk, H(m))$
  - $V'(pk, m, \sigma) = V(pk, H(m), \sigma)$
- Security: $\mathrm{SIG}_{\mathrm{adv}}[\mathcal{A}, \mathcal{S}'] \leq \mathrm{SIG}_{\mathrm{adv}}[\mathcal{B}_S, \mathcal{S}] + \mathrm{CR}_{\mathrm{adv}}[\mathcal{B}_H, H]$

RSA Signatures

Digital signatures with trapdoors.

Textbook RSA Signatures are Broken

- The naive approach: sign m by computing $\sigma = m^d \mod n$
  - Verify by checking $\sigma^e \mod n = m$
- This is broken by ***multiplicative homomorphism***
  - Given $\sigma_1 = m_1^d$ and $\sigma_2 = m_2^d$
  - Compute $\sigma = \sigma_1 \cdot \sigma_2 = (m_1 \cdot m_2)^d \mod n$
  - This is a valid signature on $m_1 \cdot m_2$ without knowing d!
- This is an ***existential forgery***: the adversary wins the UF-CMA game
  - Request signatures on m_1 and m_2, forge a signature on their product
  - The algebraic structure of RSA is the attack surface

Full Domain Hash (FDH)

- The fix: hash the message before signing (***Full Domain Hash***)
  - The hash destroys the algebraic structure that enables the homomorphism attack
- Given a trapdoor permutation $\mathcal{T} = (G, F, I)$ over $\mathcal{X}$ and a hash $H : \mathcal{M} \to \mathcal{X}$
  - Sign: $\sigma \leftarrow I(sk, H(m))$ (hash the message, then invert)
  - Verify: check if $F(pk, \sigma) = H(m)$
- With RSA: $\sigma = H(m)^d \mod n$, verify by checking $\sigma^e \mod n = H(m)$
- $\mathcal{S}_{\mathrm{FDH}}$ is provably secure in the random oracle model (ROM)
  - The ROM from Lecture 04: treat H as a truly random function
  - Security reduces to the RSA assumption
- Signatures are relatively large (at least 256 bytes for 2048-bit RSA)
- RSA is the fastest standardised scheme for verification
  - Excellent for signing certificates: sign offline, verify online

PKCS#1 v1.5 and RSA-PSS

- PKCS#1 v1.5: the most commonly deployed RSA signature standard
  - Used for signing X.509 certificates
  - Hash, then embed in a padded structure, then sign
  - No security proof! The FDH proof breaks down for partial domain hash
  - Bleichenbacher's attack exploits implementation mistakes in verification
- ***RSA-PSS***: the provably secure replacement
  - Adds randomisation via a salt; provably reduces to RSA hardness
  - Should be used instead of PKCS#1 v1.5 in new applications

Schnorr Signatures and ECDSA

Discrete log-based signatures.

Schnorr Signatures

- ***Schnorr signatures*** are the cleanest discrete log-based scheme
  - Security reduces directly to the discrete log problem (in the ROM)
- Setup: group $\mathbb{G}$ of prime order q with generator g, hash function H
  - Secret key: $\alpha \xleftarrow{R} \mathbb{Z}_q$
  - Public key: $u = g^{\alpha}$
- Signing message m:
  - Pick random $\beta \xleftarrow{R} \mathbb{Z}_q$, compute $u_t = g^{\beta}$ (the commitment)
  - Compute challenge $c = H(m \| u_t)$
  - Compute response $\alpha_z = \beta - \alpha c \mod q$
  - Signature: $\sigma = (u_t, \alpha_z)$
- Verification: check that $g^{\alpha_z} \cdot u^c = u_t$ where $c = H(m \| u_t)$
  - This works because $g^{\beta - \alpha c} \cdot g^{\alpha c} = g^{\beta} = u_t$

ECDSA

- ***ECDSA*** is the standardised variant using elliptic curves
  - Used in TLS, code signing, cryptocurrency (Bitcoin, Ethereum)
  - Based on the same discrete log structure as Schnorr, but with a different construction
- ECDSA signatures are short (64 bytes on a 256-bit curve, for 128-bit security)
  - Compared to 256+ bytes for RSA signatures
  - A significant advantage for bandwidth-constrained protocols

The Catastrophic Failure: Nonce Reuse

- Both Schnorr and ECDSA require a fresh random nonce β for every signature
- ***Nonce reuse*** is catastrophic: if the same β is used for two different messages m_1, m_2:
  - The commitment $u_t = g^{\beta}$ is identical in both signatures
  - The adversary has two equations with two unknowns (α and β)
  - They can solve for the secret key α!
- This is not a theoretical concern. The Sony PlayStation 3 ECDSA attack (2010):
  - Sony used the same nonce for every signature
  - The private signing key was recovered, enabling homebrew software
  - One constant nonce, total key compromise
- This is an instance of a general principle: randomness failure is catastrophic in cryptographic protocols
  - The same principle applies to nonce discipline for AEAD (Lecture 04)

PKI and Certificates

How does the verifier get the right public key?

The Trust Problem

- Digital signatures are useless if the verifier has the wrong public key
  - If Mallory gives Bob her public key and claims it's Alice's...
  - Bob will accept Mallory's signatures as if they came from Alice
- This is the natural question after teaching signatures:
  - How does the verifier obtain the correct public key?

Certificates

- A ***certificate*** is a signature by a trusted authority over a public key binding
  - The CA verifies Alice's identity
  - The CA signs a statement: "public key pk belongs to Alice"
  - The CA signs with its own secret key: $\sigma_{\mathrm{CA}} \leftarrow S(sk_{\mathrm{CA}}, m)$
  - The result $(m, \sigma_{\mathrm{CA}})$ is Alice's certificate
- Bob verifies using the CA's public key pk_{CA}
  - pk_{CA} is pre-installed in browsers and operating systems
- ***Chain of trust***: CAs can delegate to intermediate CAs
  - Root CA signs intermediate CA's certificate
  - Intermediate CA signs end-entity certificates
  - Verification follows the chain up to a trusted root

When Trust Breaks: DigiNotar

- The trust model's critical weakness is not cryptographic
  - It's the assumption that CAs are trustworthy: a governance problem, not a mathematical one
- DigiNotar (2011): a Dutch CA was compromised
  - Attackers issued fraudulent certificates for google.com, *.google.com, and others
  - Used to intercept Gmail traffic of Iranian dissidents
  - The forged certificates were cryptographically valid
  - All major browsers revoked DigiNotar's root certificate; the company went bankrupt
- ***Certificate Transparency (CT)*** is the structural fix
  - All issued certificates must be logged in public, append-only logs
  - Anyone can audit the logs for fraudulent certificates
  - Browsers can refuse certificates that aren't logged

CCA Security

The right security target for encryption.

Motivation: Why IND-CPA is Not Enough

- IND-CPA security protects against passive adversaries
  - The adversary can encrypt arbitrary messages, but nothing more
- Real adversaries are active: they can submit ciphertexts for decryption
  - A server's error response to a malformed ciphertext is information
  - Even "decryption failed" vs "padding error" is a useful oracle

The Bleichenbacher Attack

- Bleichenbacher's attack (1998) on PKCS#1 v1.5 RSA encryption:
  - The server decrypts an RSA ciphertext and checks the PKCS#1 padding format
  - If the padding is malformed, the server returns an error
  - This single-bit oracle ("valid padding or not") is enough to recover the plaintext
- The adversary submits millions of modified ciphertexts
  - Each modification multiplies the underlying plaintext by a chosen factor (RSA's multiplicative homomorphism)
  - The oracle's responses progressively narrow down the plaintext value
  - Eventually, the full session key is recovered
- The adversary never sees any decrypted plaintext
  - Just "yes/no" responses about padding validity
  - This is exactly what a CCA2 decryption oracle provides

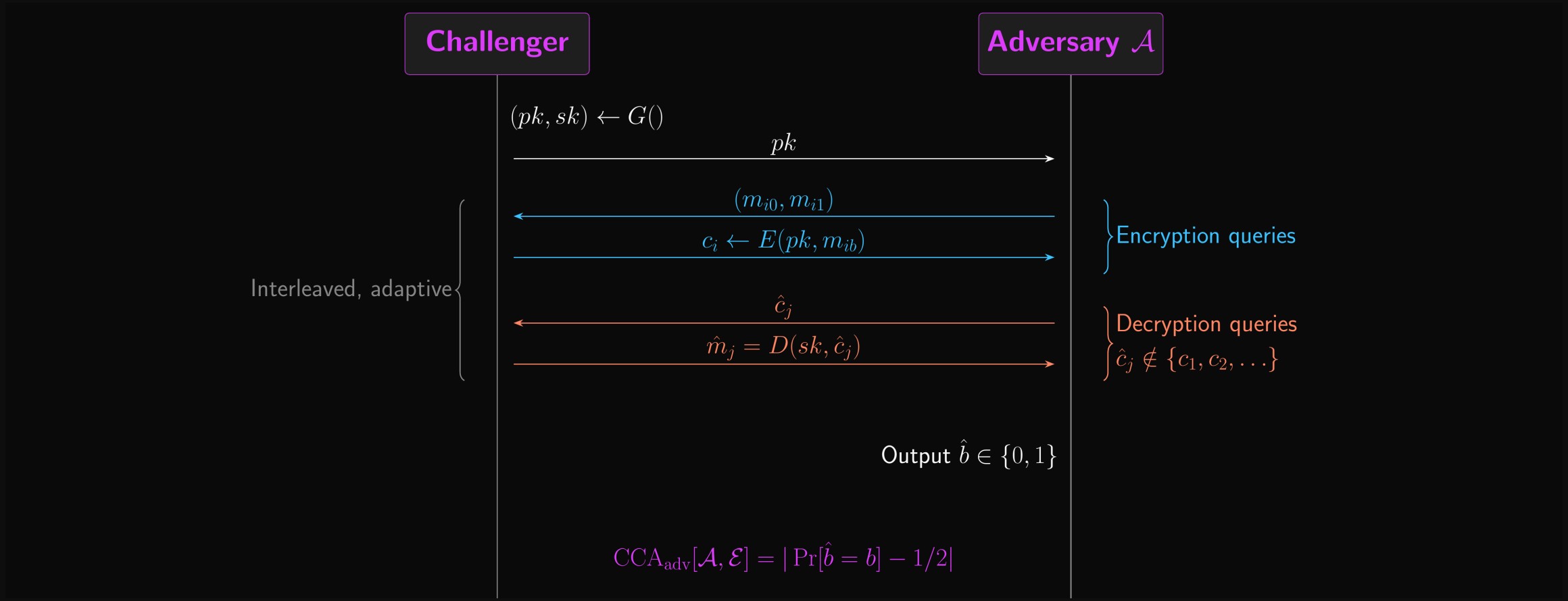
IND-CCA2: Formal Definition

- We've looked at several notions of security based on ciphertext indistinguishability:
  - IND-CPA: adversary can encrypt arbitrary messages
  - IND-CCA1: adversary can also decrypt, but only before the challenge (non-adaptive)
  - ***IND-CCA2***: adversary can decrypt before and after the challenge (adaptive)
- Each level gives $\mathcal{A}$ strictly more power:
  - IND-CCA2 $\implies$ IND-CCA1 $\implies$ IND-CPA

Public Key CCA Attack Game

- Let $\mathcal{E} = (G, E, D)$, defined over message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$
- Two experiments for $b \in \{0, 1\}$
- Challenger generates $(pk, sk) \leftarrow G()$ and sends pk to $\mathcal{A}$
- $\mathcal{A}$ may submit encryption queries
  - Submit $(m_{i0}, m_{i1}) \in \mathcal{M}^2$, receive $c_i \leftarrow E(pk, m_{ib})$
- $\mathcal{A}$ may submit decryption queries
  - Submit $\hat{c}_j \in \mathcal{C}$, receive $\hat{m}_j = D(sk, \hat{c}_j)$
  - $\hat{c}_j$ can't be the result of an encryption query: $\hat{c}_j \notin \{c_1, c_2, \ldots\}$
- $\mathcal{A}$ outputs a bit $\hat{b} \in \{0, 1\}$
- $\mathrm{CCA}_{\mathrm{adv}}[\mathcal{A}, \mathcal{E}] = |\Pr[W_0] - \Pr[W_1]|$
- $\mathcal{E}$ is CCA-secure if $\mathrm{CCA}_{\mathrm{adv}}[\mathcal{A}, \mathcal{E}]$ is negligible for all efficient $\mathcal{A}$

CCA Security: Attack Game



CCA and Malleability

- Plenty of ciphers we've studied are ***malleable***
  - Given a ciphertext, we can create a new ciphertext for a related plaintext
  - With a stream cipher: XORing bits of the ciphertext flips bits of the plaintext
  - With RSA: multiplying ciphertexts multiplies plaintexts
- If ciphertexts are malleable, then CCA security fails
  - The adversary modifies the challenge ciphertext and submits it for decryption
  - The decryption oracle reveals information about the original plaintext
- CCA security $\implies$ ***non-malleability***
  - We already know how to achieve this for symmetric ciphers: Encrypt-then-MAC (Lecture 04)
  - For public key encryption: we need OAEP

RSA-OAEP

Closing the loop on textbook RSA.

The Problem with Textbook RSA

- In Lecture 06, we flagged that textbook RSA ($c = m^e \mod n$) is deterministic
  - Therefore not IND-CPA secure, let alone IND-CCA2
- RSA's multiplicative homomorphism makes it malleable
  - Given $c = m^e$, compute $c' = r^e \cdot c = (rm)^e$ for any chosen r
  - Submit c' for decryption, get rm, divide by r to recover m

OAEP: Randomised Padding

- ***Optimal Asymmetric Encryption Padding (OAEP)*** adds randomisation
  - Uses two hash functions H and W in a Feistel-like structure
  - Mixes a random value r into the plaintext before RSA encryption
- The padded value x is the input to RSA: $c = x^e \mod n$
  - x depends on both m and the random r
  - Same plaintext produces different ciphertexts each time

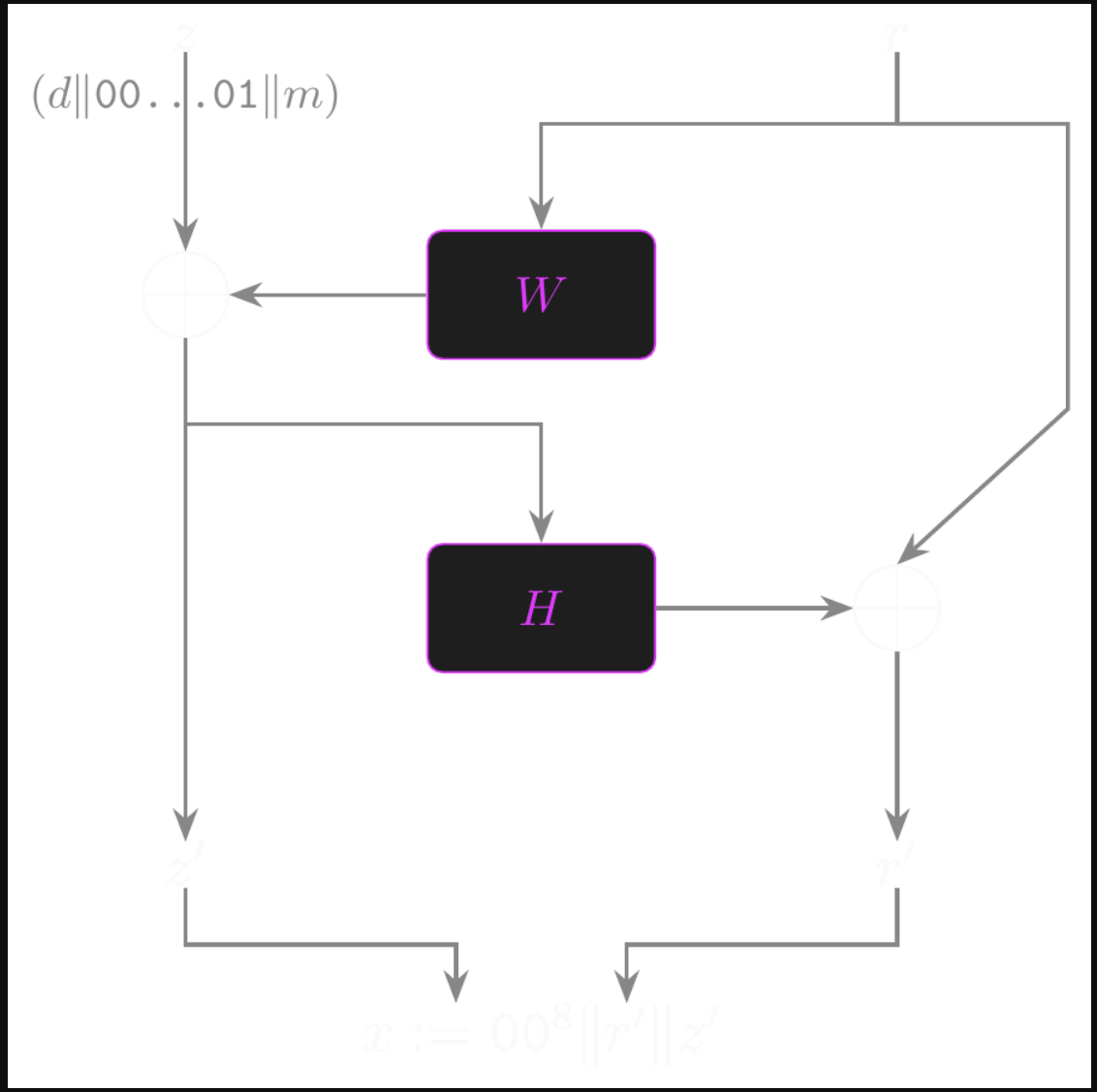
OAEP: Padding Algorithm

- $P(m, r, d)$:

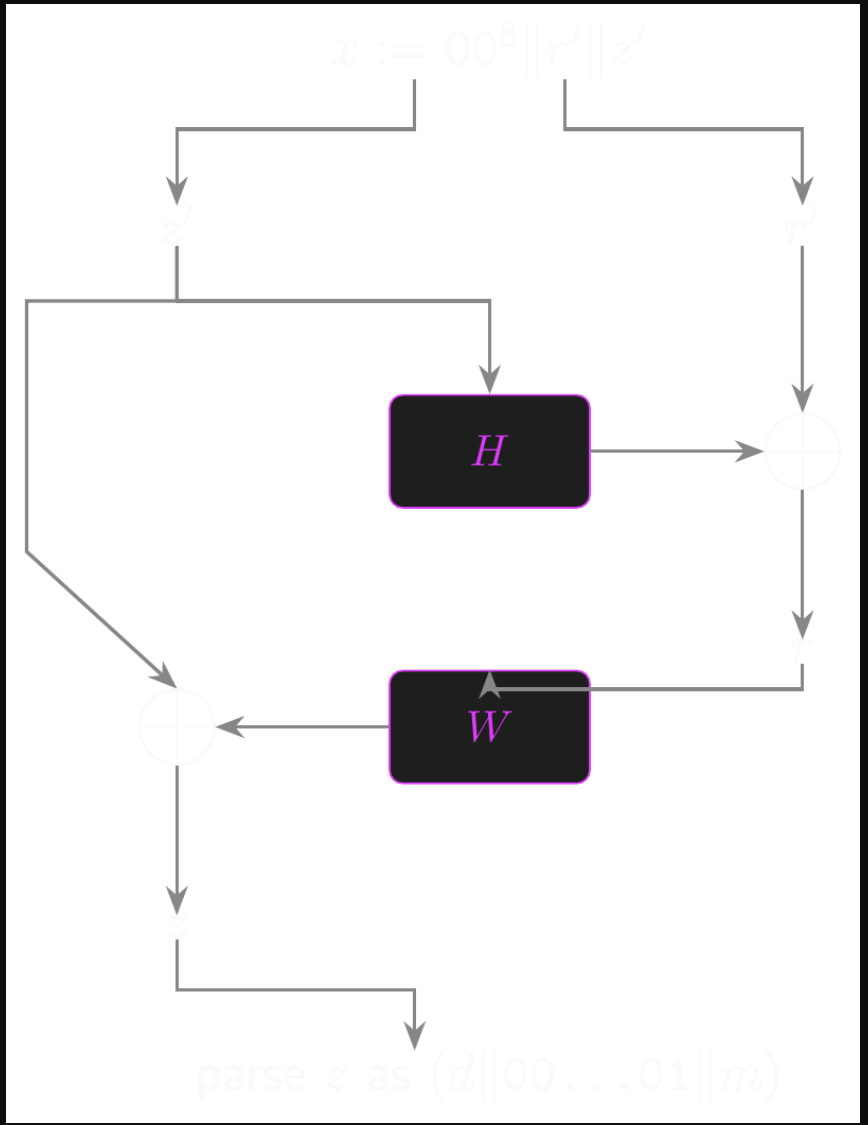
$$
\begin{aligned}
&z \leftarrow (d \| \texttt{00\ldots01} \| m) \quad \text{s.t. } |z| = (t - h - 8) \text{ bits} \\
&z' \leftarrow z \oplus W(r) \\
&r' \leftarrow r \oplus H(z') \\
&x \leftarrow (00^8 \| r' \| z') \quad \text{s.t. } |x| = t \text{ bits} \\
&\textbf{return } x
\end{aligned}
$$

- Unpadding reverses the process and rejects invalid formats

OAEP: Padding Structure



OAEP: Unpadding Structure



Security of RSA-OAEP

- RSA-OAEP is IND-CCA2 secure in the random oracle model
  - Security requires RSA to be a partial one-way function
  - If RSA is one-way (the standard assumption), it is also partial one-way
  - So RSA-OAEP is CCA-secure under the standard RSA assumption
- This closes the loop from Lecture 06:
  - RSA trapdoor permutation (the primitive)
  - Textbook RSA encryption (IND-CPA broken, deterministic)
  - RSA-OAEP (IND-CCA2 secure, randomised)
- The progression illustrates that randomisation is essential for public key encryption security
  - The same principle as IND-CPA requiring randomised encryption in the symmetric setting
- The Web Crypto API provides an implementation of RSA-OAEP
  - Please don't attempt to roll your own...

Authenticated Encryption

From CCA security to practical AEAD.

From CCA to Authenticated Encryption

- We just saw two approaches to CCA-secure encryption:
  - RSA-OAEP: randomised padding for public-key encryption
  - But what about symmetric encryption?
- In Lecture 04, we stated the ***Encrypt-then-MAC theorem***:
  - Combining a CPA-secure cipher with a secure MAC via Encrypt-then-MAC achieves IND-CCA2 security
- Earlier in this lecture, we defined IND-CCA2 formally and saw why it matters
  - Active adversaries with decryption oracles can break CPA-secure ciphers
  - The Bleichenbacher attack demonstrated this in practice
- Now let's prove the theorem

Recall: The CCA Theorem

- In Lecture 04, we stated: Encrypt-then-MAC with a CPA-secure cipher and a secure MAC achieves IND-CCA2 security
- The argument has two steps:
  - Step 1: The MAC prevents the adversary from using the decryption oracle effectively
  - Step 2: Without useful decryption queries, the adversary faces only CPA security

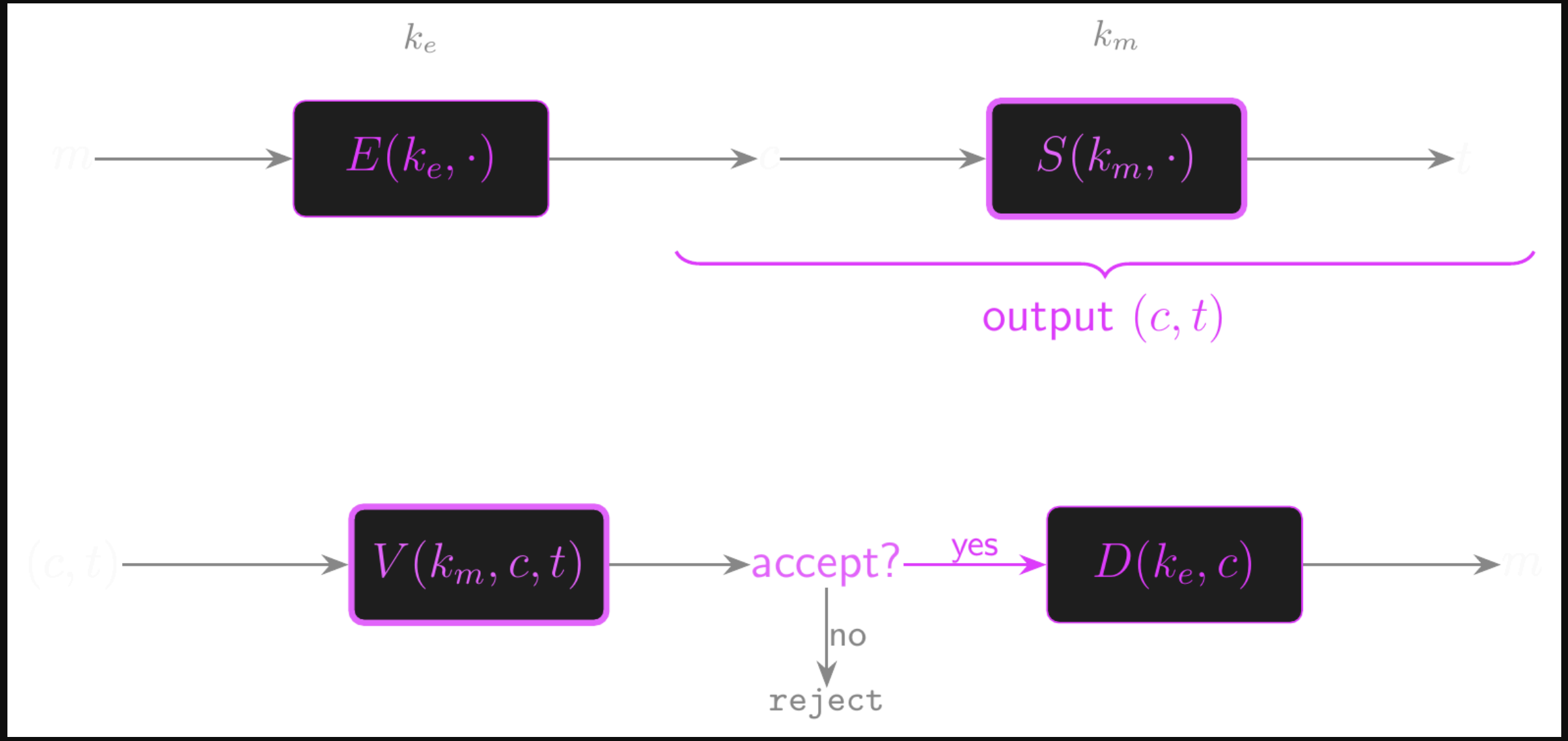
Step 1: The MAC Neuters the Decryption Oracle

- In the CCA2 game, the adversary can submit ciphertexts (c, t) for decryption
  - But they can't submit any (c_i, t_i) received from an encryption query
- For any other (c', t'), the MAC verification $V(k_m, c', t')$ must pass
  - If the MAC is secure (UF-CMA), the adversary can't produce a valid tag for a ciphertext they didn't receive
  - The probability of a valid forgery is $\mathrm{MAC}_{\mathrm{adv}}[\mathcal{A}, \mathcal{I}]$, which is negligible
- So with overwhelming probability, every decryption query returns `reject`
  - The decryption oracle gives the adversary no useful information!

Step 2: CPA Security Handles the Rest

- With the decryption oracle effectively disabled, the adversary is left with:
  - The public parameters
  - Encryption queries (which they can make freely)
  - Decryption queries that always return `reject` (useless)
- This is functionally equivalent to the CPA game
  - The CPA-secure cipher handles this case
- Formally: $\mathrm{CCA}_{\mathrm{adv}}[\mathcal{A}, \mathcal{E}_{\mathrm{EtM}}] \leq \mathrm{CPA}_{\mathrm{adv}}[\mathcal{B}, \mathcal{E}] + \mathrm{MAC}_{\mathrm{adv}}[\mathcal{C}, \mathcal{I}]$
  - Both terms are negligible, so the combined advantage is negligible
- This is why Encrypt-then-MAC is the gold standard for symmetric authenticated encryption

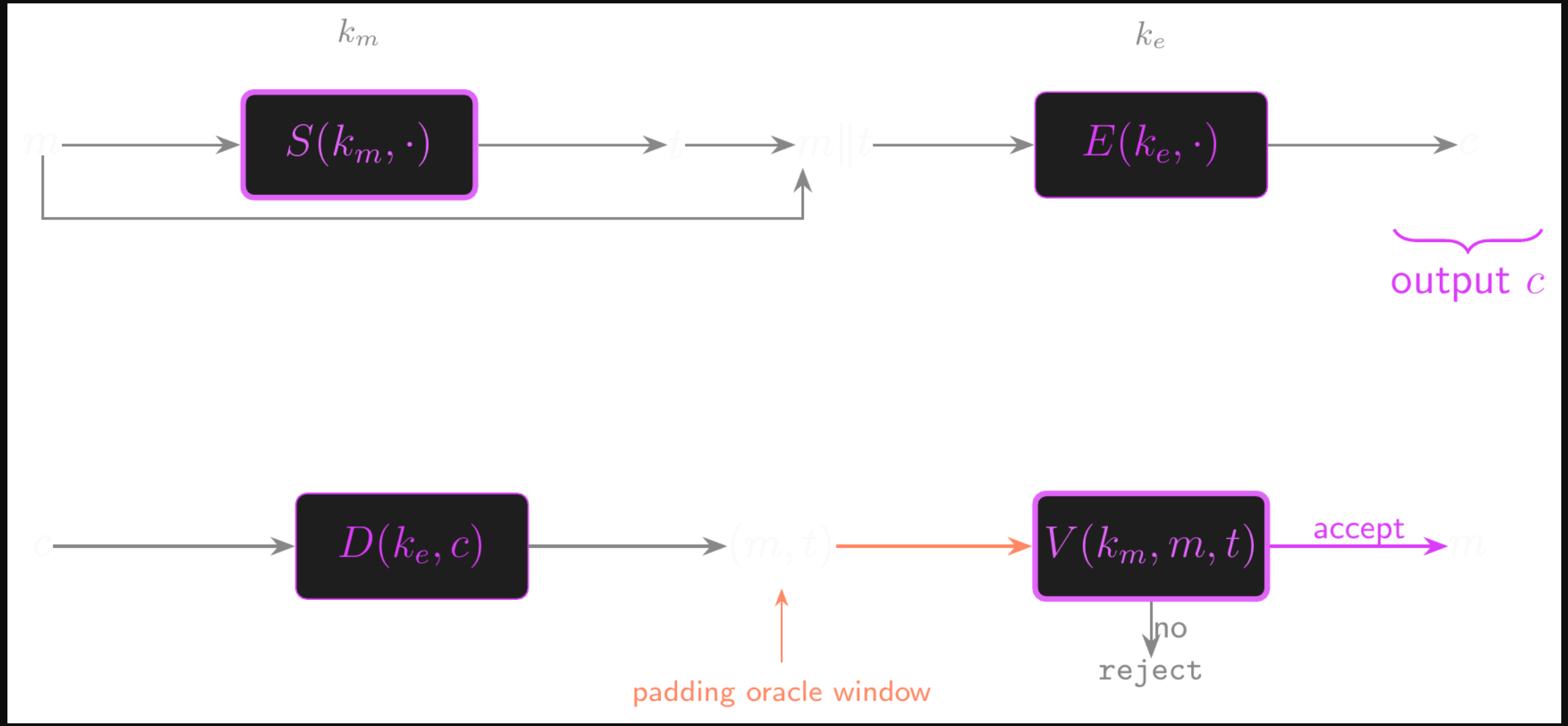
Encrypt-then-MAC



MAC-then-Encrypt

- The alternative composition: compute a tag on the plaintext, then encrypt everything
  - $t \leftarrow S(k_m, m)$, then $c \leftarrow E(k_e, m \| t)$
- This is ***not*** generally CCA-secure
  - The encryption layer hides the tag from the verifier
  - The decryptor must remove padding and parse the plaintext before it can check the tag
- This creates a ***padding oracle*** vulnerability:
  - If the server leaks whether the padding is valid (before checking the MAC), the adversary gets an oracle
  - The POODLE attack (2014) exploited exactly this in TLS 1.0-1.2 CBC suites
  - Even a single bit of information ("valid padding" vs "invalid padding") can be enough
- TLS 1.3 dropped all CBC cipher suites, requiring AEAD-only
  - Encrypt-then-MAC (or integrated AEAD) avoids the problem entirely

MAC-then-Encrypt



Nonce-Based AEAD

- In practice, authenticated encryption is packaged as a single primitive: ***AEAD***
  - ***Authenticated Encryption with Associated Data***
- A nonce-based AEAD cipher $\mathcal{E} = (E, D)$ is defined over $(\mathcal{K}, \mathcal{M}, \mathcal{C}, \mathcal{D}, \mathcal{N})$:
  - $E(k, m, d, n) \to c$: encrypt message m with associated data d and nonce n
  - $D(k, c, d, n) \to m$ or `reject`: decrypt or reject if tampered
- Correctness: for all k, m, d, n: $D(k, E(k, m, d, n), d, n) = m$
- Security requires two properties:
  - ***Ciphertext indistinguishability***: the ciphertext reveals nothing about the plaintext
  - ***Ciphertext integrity***: no efficient adversary can produce a valid ciphertext that wasn't generated by the encryption oracle
- The nonce n must be unique per encryption under the same key
  - It need not be secret, but repeating it is catastrophic

AEAD: Operational Details

AES-GCM internals and nonce misuse resistance.

GCM Structure

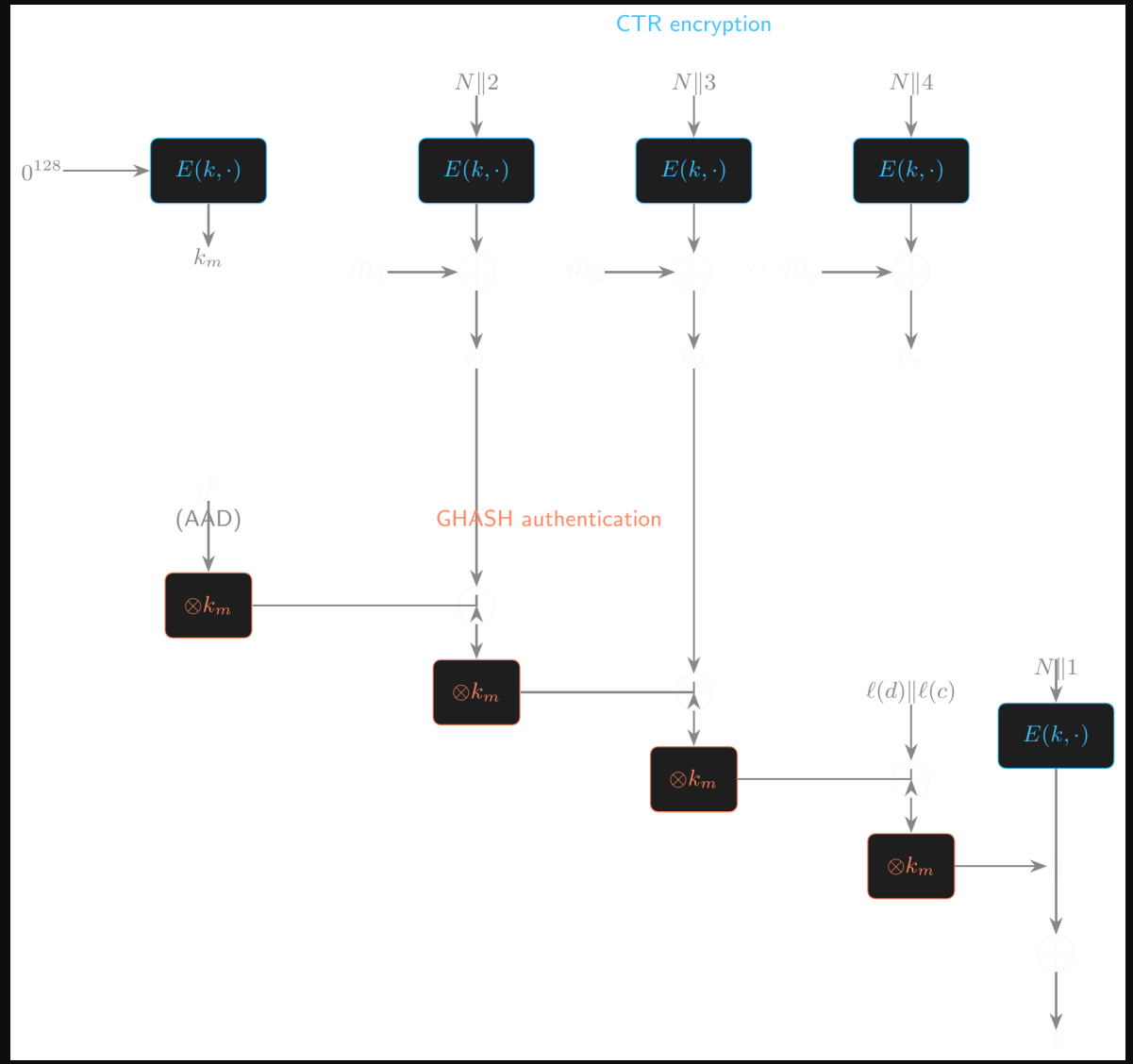
- ***Galois Counter Mode (GCM)*** is the most widely deployed AEAD cipher
  - Standardized by NIST in 2007; used in TLS 1.3, IPsec, and many other protocols
- GCM follows encrypt-then-MAC internally:
  - Encryption: AES in counter mode (CTR)
  - Authentication: Carter-Wegman MAC using GHASH over GF(2^{128})
- Single key k derives both the encryption key and the authentication key k_m
  - $k_m \leftarrow E(k, 0^{128})$ (encrypt the zero block)

GCM Encryption

- Input: key k, message m, associated data d, and nonce $N \in \{0,1\}^{96}$
- $\mathrm{length}()$ returns a 64-bit value containing the length of the field in bits

$$
\begin{aligned}
&k_m \leftarrow E(k, 0^{128}) \\
&x \leftarrow (N \| 0^{31} \| 1) \in \{0,1\}^{128} \\
&x' \leftarrow x + 1 \\
&c \leftarrow \text{CTR-Encrypt}(k, x', m) \\
&d' \leftarrow \text{ZeroPad}(d, 128), \quad c' \leftarrow \text{ZeroPad}(c, 128) \\
&h \leftarrow \text{GHASH}(k_m,\ d' \| c' \| \text{length}(d) \| \text{length}(c)) \\
&t \leftarrow h \oplus E(k, x) \\
&\textbf{return } (c, t)
\end{aligned}
$$

GCM Encryption Structure



GHASH

- ***GHASH*** is a Carter-Wegman MAC over GF(2^{128})
  - A polynomial hash function, similar to the UHF constructions from Lecture 04
- GF(2^{128}) is a Galois field defined by the irreducible polynomial:
  - $g(X) = X^{128} + X^7 + X^2 + X + 1$
  - Elements are 128-bit bitstrings; addition is XOR, multiplication is mod $g(X)$
- GHASH(k, z) for $z = (z_0, z_1, \ldots, z_{v-1}) \in$ GF(2^{128})^v:
  - $z_0 \cdot k^v + z_1 \cdot k^{v-1} + \ldots + z_{v-1} \cdot k \in$ GF(2^{128})
  - Evaluable incrementally via Horner's method (streaming)
- Modern processors have hardware support for Galois field multiplication
  - AES-GCM is extremely fast with AES-NI and PCLMULQDQ instructions

ChaCha20-Poly1305

- The alternative when AES hardware support is unavailable
  - ChaCha20 (stream cipher from Lecture 02) + ***Poly1305*** (one-time MAC)
  - Combined as an AEAD cipher in RFC 7539
- Poly1305 is a one-time MAC based on polynomial evaluation
  - The same polynomial MAC construction from Lecture 04
  - One-time key derived from ChaCha20's keystream for each message
- Both AES-GCM and ChaCha20-Poly1305 are available in TLS 1.3

Nonce Misuse Resistance: AES-GCM-SIV

- Standard AES-GCM has no ***nonce misuse resistance***
  - Nonce reuse catastrophically breaks both confidentiality and integrity
  - This was covered in detail in Lecture 04
- ***AES-GCM-SIV*** (RFC 8452) provides some protection:
  - The nonce is mixed into the tag computation, and the tag is used as the IV for CTR mode
  - If the nonce repeats but the plaintext differs, the ciphertexts still differ
  - If both nonce and plaintext repeat, it only leaks that the same message was sent twice
  - Authentication is not broken by nonce reuse
- The cost: slightly more complex, slightly slower
  - A worthwhile trade-off when nonce uniqueness cannot be guaranteed

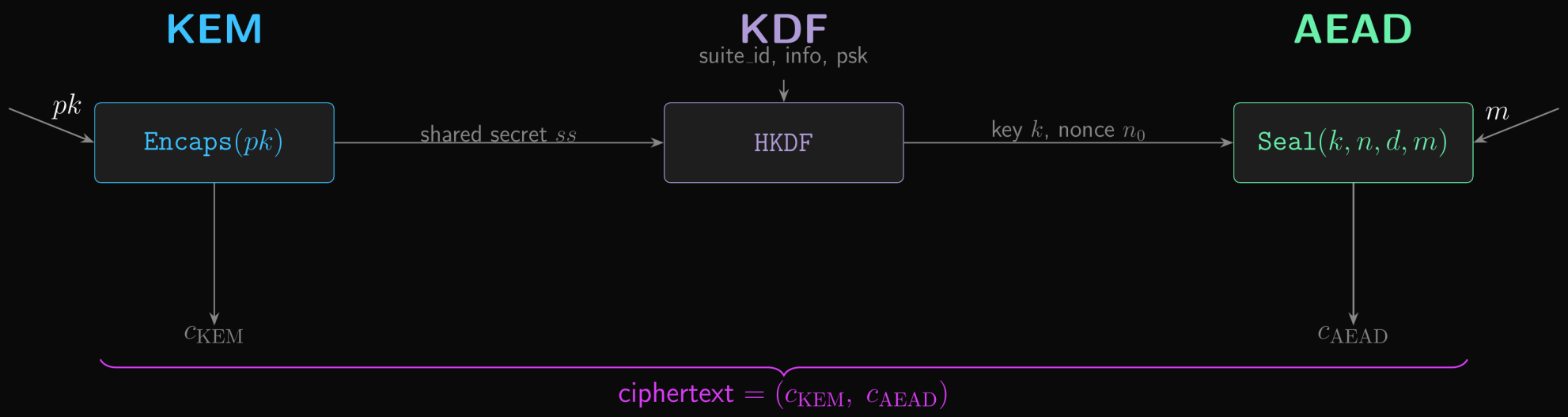
HPKE

Hybrid public-key encryption: RFC 9180.

HPKE

- ***Hybrid Public Key Encryption (HPKE)*** is defined in RFC 9180
  - Plug-and-play public-key encryption framework
- Why define a standard? There are many homebrew hybrid cryptosystems in the wild
  - Many use outdated primitives or lack proofs of IND-CCA2 security
  - No shared test vectors, no interoperability
- HPKE eliminates one-off “ECIES-ish” constructions
  - Easy to audit and reuse
  - Supports any KEM + KDF + AEAD trio
  - Already powering TLS-ECH, OHTTP, MLS and more

HPKE: Three-Stage Pipeline



Decrypt: $ss \leftarrow \mathrm{Decaps}(sk, c_{\mathrm{KEM}}), \quad (k, n_0) \leftarrow \mathrm{HKDF}(ss, \ldots), \quad m \leftarrow \mathrm{Open}(k, n, d, c_{\mathrm{AEAD}})$

Base mode shown; PSK and Auth modes add extra key material

HPKE Ciphersuites

- Each HPKE ***ciphersuite*** needs three principal components:
  - A key encapsulation mechanism (KEM)
  - A key derivation function (KDF)
  - An AEAD encryption algorithm
- The usual assumptions about security apply to each of those components
  - Familiar primitives assembled into a larger construct
- A HPKE ciphersuite can be identified by the triple of algorithms:
  - E.g. DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, AES-128-GCM

HPKE Ciphersuite Registry

- IANA maintains a public registry for each of the component types
  - Maps a 16-bit identifier to algorithms in each category
- KDFs and AEADs are more limited
  - KDF restricted to HKDF-SHA256 and similar
  - AEAD restricted to AES-GCM and ChaCha20-Poly1305
- Lots of possible KEMs, though
  - Various curves for elliptic curve Diffie-Hellman
  - Newer options like the X25519/Kyber-768 PQ/T hybrid
  - And the ML-KEM family
- Up to the protocol to negotiate what algorithms will be used
  - Be wary of downgrade attacks

HPKE Modes

Mode	Extra key material	Sender authentication	Typical use case
Base (0x00)	None	No	Simple message encryption
PSK (0x01)	PSK + PSK_ID	PSK only	IoT bootstrapping
Auth (0x02)	Sender static key	Yes	Secure logging
Auth-PSK (0x03)	Both	Yes	Hardened channels

HPKE Security

- HPKE has strong security guarantees
  - IND-CCA2 for all modes
  - Relies on AEAD integrity and non-repeating nonces
- Forward secrecy depends on the direction of compromise:
  - Forward secret with respect to sender compromise
  - Not forward secret with respect to recipient compromise
- No attempt made to hide message length
  - Pre-pad plaintexts if you need this property
- Replay protection is protocol-dependent
  - HPKE itself does not prevent replay; the outer protocol must handle it

Key Commitment

AEAD with an extra guarantee.

Key Commitment

- ***Key commitment*** is a property that crops up in AEAD discussions
  - A ciphertext/tag pair binds to exactly one secret key
  - There is negligible probability that the same (c, t, d, n) decrypts successfully under two distinct keys
- Why is this useful?
  - Prevents key-substitution attacks
  - Useful in protocols that re-encrypt or “frank” messages
  - Or where adversaries can try many keys

GCM and Key Commitment

- Galois/counter mode (GCM) is not key-committing by default
- An adversary who knows two keys k_1, k_2 can craft (c, t) that validates under both
  - This is called a ***multi-key forgery***
  - Practical exploits on message-franking systems were demonstrated in 2022
- There are simple workarounds:
  - Wrap GCM as ***KC-GCM*** to provide key commitment
  - Adding an all-zero plaintext block to the start of the message also works
  - If the first block doesn't decrypt to zero, reject
  - The catch is making this check constant time across both pathways

Conclusion

What did we learn?

So, what did we learn?

- Digital signatures: the (G, S, V) triple, UF-CMA security, hash-and-sign
  - RSA-FDH, PKCS#1 v1.5 and RSA-PSS
  - Schnorr and ECDSA (and the catastrophe of nonce reuse)
- PKI and certificates: the trust infrastructure that makes signatures useful
- CCA security: why IND-CPA is not enough
  - The Bleichenbacher attack: a single-bit oracle recovers the plaintext
  - RSA-OAEP: randomised padding achieves IND-CCA2 for public-key encryption
- Authenticated encryption: the Encrypt-then-MAC CCA proof
  - The MAC neuters the decryption oracle; CPA security handles the rest
  - MAC-then-Encrypt and the padding oracle vulnerability
- AEAD in practice: AES-GCM, ChaCha20-Poly1305, nonce misuse resistance
- HPKE: plug-and-play hybrid PKE from KEM + KDF + AEAD (RFC 9180)
- Key commitment: why standard GCM is not key-committing, and how to fix it

Where Do We Go from Here?

- The theoretical arc is nearly complete
  - One more operational topic before the project: key derivation and password hashing
- Next lecture covers:
  - The entropy gap between passwords and cryptographic keys
  - PBKDF2 and its limitations (compute-hard but not memory-hard)
  - Argon2id: the current gold standard for password hashing
  - A lab exercise building the EPIC project's core cryptographic stack

For Next Time

- Complete some assigned reading:
  - Boneh and Shoup, Chapters 9 (authenticated encryption) and 18 (signatures)
  - The OWASP Password Storage Cheat Sheet
  - Skim RFC 9180 (HPKE), Sections 1-4

Questions?

Ask now, catch me after class, or email eoin@eoin.ai