ISE Cryptography – Lecture 06

Public Key Cryptography

The Key Distribution Problem

Every symmetric primitive assumes a shared key.

The Problem

- We've spent four lectures building powerful symmetric primitives
  - Stream ciphers, block ciphers, MACs, hash functions and authenticated encryption
  - Every single one assumes Alice and Bob already share a secret key
- How did Alice and Bob get the key in the first place?
  - Can't just send the key in plaintext, after all!
  - Pretty much every secure messaging app must tackle this
  - Ever meet up with Jeff Bezos to get a secret key for amazon.com?
- Kerckhoffs's principle: the key is the only secret
  - Key establishment is the existential problem of symmetric cryptography
- Alice and Bob must set up a key exchange protocol to protect from eavesdropping
  - The protocol is anonymous: it doesn't authenticate Alice and Bob
  - We'll address authentication later (Lecture 07)

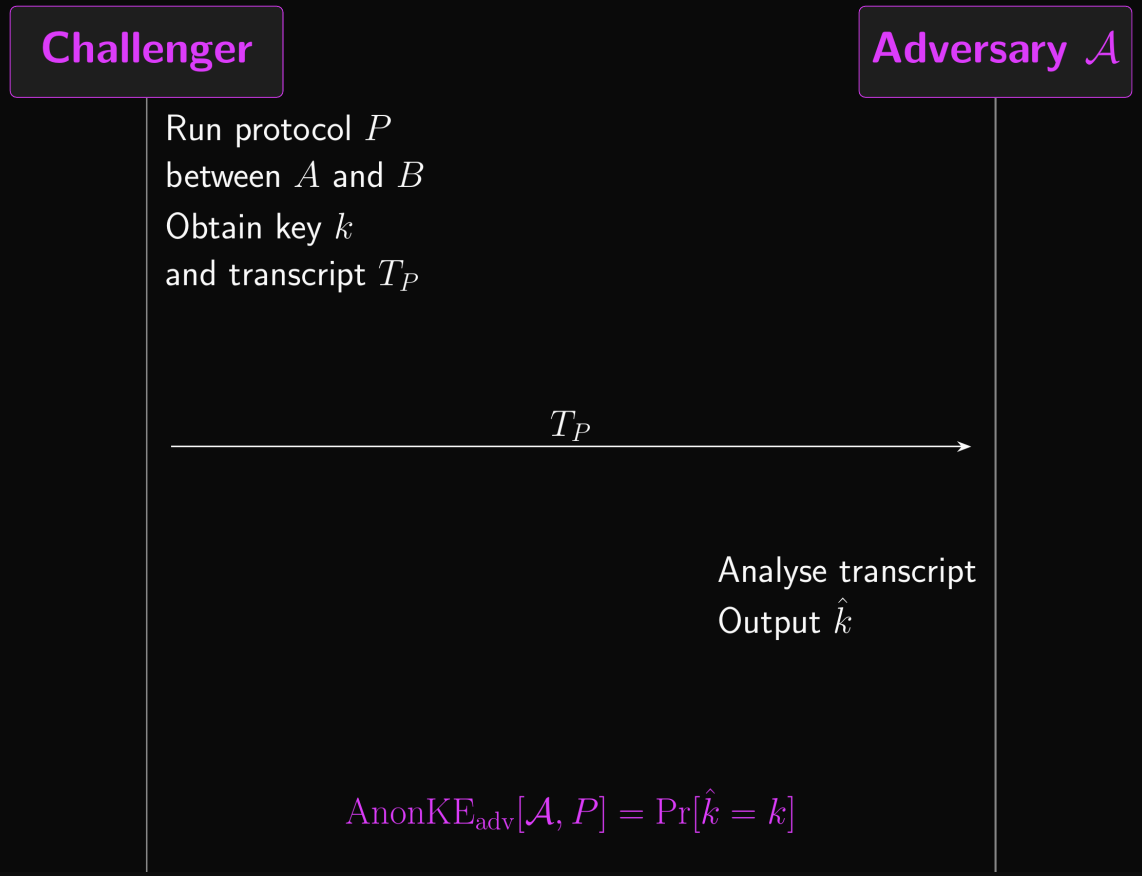
Key Exchange Protocol

- A key exchange protocol P is
  - A pair of probabilistic machines (A, B)
  - That take turns sending messages to each other
- At the end of the protocol, when both machines terminate
  - They both obtain the same value k as their secret key
- Note that they're not deciding on k ahead of time!
  - They'll both reach the same value for k
  - But neither party necessarily knows it in advance
- A protocol transcript T_P is the sequence of messages exchanged between (A, B)
  - In a single execution of the protocol P
  - A and B are probabilistic, so the transcript may be different every time
  - T_P is essentially a random variable based on A and B's random bits
- If an eavesdropping adversary $\mathcal{A}$ sees T_P, can they figure out the secret key k?

Key Exchange Protocol: Attack Game

- If an eavesdropping adversary $\mathcal{A}$ sees T_P, can they figure out the secret key k?
  - Let's build an attack game to get an idea of security!
- The challenger runs the anonymous key exchange protocol P between A and B
  - Generates a secret key k
  - And a transcript T_P
- The challenger sends the transcript T_P to $\mathcal{A}$
- $\mathcal{A}$ outputs a guess $\hat{k}$ for k
  - The hat symbol is convention to show that it's an estimator for k
- $\mathcal{A}$'s advantage is the probability that the guess is correct
  - $\mathrm{AnonKE}_{\mathrm{adv}}[\mathcal{A}, P]$ = Pr[$\hat{k} = k$]
- An anonymous key exchange protocol P is secure against an eavesdropper if
  - For all efficient adversaries $\mathcal{A}$, $\mathrm{AnonKE}_{\mathrm{adv}}[\mathcal{A}, P]$ is negligible

Key Exchange: Attack Game



Key Exchange Protocol: Security

- This is a weak definition of security with more holes than Emmental...
- We're assuming that the adversary doesn't tamper with the messages
  - This is unrealistic in real-world applications!
- We're only measuring if the adversary can guess the entire key
  - They could guess 90% of the bits correctly each time and we don't care...
- The protocol is anonymous
  - No one can be certain of anyone else's identity
  - A huge problem if, say, Mallory pretends to be Bob...
- Weak security or not, can we do this kind of key exchange with the symmetric ciphers we've encountered so far?
  - Actually, yes we can!
  - But it's horrifically inefficient, so we're not even going to look at it
  - We need better tools if we want to do this securely and efficiently!

Trapdoor Functions

One-way functions with a secret shortcut.

One-Way Functions

- We've seen one-way functions already (hash functions, Lecture 05) - easy to compute, hard to invert
  - Lots of functions meet this definition!
  - Can we recover possible inputs from a SHA-256 digest?
- But one-way functions aren't going to help us just yet
  - We need a special feature - a trapdoor
- A one-way trapdoor function should be easy to compute
  - And extremely difficult to invert
  - Unless you know the trapdoor - a secret value that lets you invert it efficiently!
- That's nice, but a bit hazy...
  - Let's define it more rigorously so we can build a key exchange protocol!

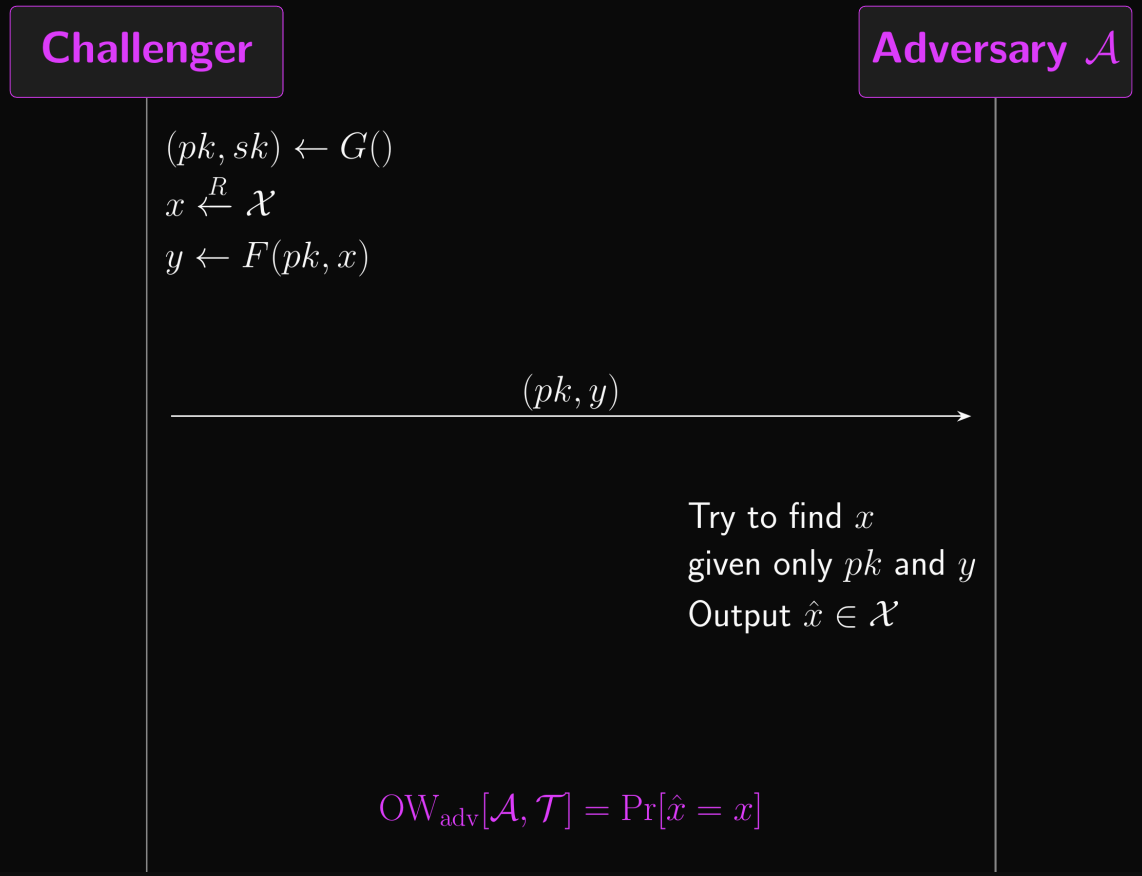
One-Way Trapdoor Functions

- A trapdoor function scheme $\mathcal{T}$, defined over finite sets $(\mathcal{X}, \mathcal{Y})$, is a triple of algorithms (G, F, I)
- G is a probabilistic key generation algorithm
  - Invoked as $(pk, sk) \leftarrow G()$
  - Where pk is called a public key and sk is called a secret key
- F is a deterministic algorithm
  - Invoked as $y \leftarrow F(pk, x)$
  - Where pk is a public key generated by G, $x \in \mathcal{X}$ and $y \in \mathcal{Y}$
- I is a deterministic algorithm
  - Invoked as $x \leftarrow I(sk, y)$
  - Where sk is a secret key generated by G, $y \in \mathcal{Y}$ and $x \in \mathcal{X}$
- The correctness property should be satisfied
  - For all possible outputs (pk, sk) of G and for all $x \in \mathcal{X}$
  - $I(sk, F(pk, x)) = x$

Trapdoors: Attack Game

- The challenger computes:
  - $(pk, sk) \leftarrow G()$
  - $x \xleftarrow{R} \mathcal{X}$
  - $y \leftarrow F(pk, x)$
- The challenger sends (pk, y) to the adversary
- The adversary outputs a guess $\hat{x} \in \mathcal{X}$
- $\mathcal{A}$'s advantage in inverting $\mathcal{T}$ is the probability that $\hat{x} = x$
  - $\mathrm{OW}_{\mathrm{adv}}[\mathcal{A}, \mathcal{T}]$ = Pr[$\hat{x} = x$]
- A trapdoor function scheme $\mathcal{T}$ is one-way if
  - For all efficient adversaries $\mathcal{A}$, $\mathrm{OW}_{\mathrm{adv}}[\mathcal{A}, \mathcal{T}]$ is negligible
- The function is hard to invert even if the output and public key are known
- Some elements of $\mathcal{Y}$ might not have preimages under $F(pk, \cdot)$
- If $\mathcal{X} = \mathcal{Y}$, then $F(pk, \cdot)$ is bijective, which means that $F(pk, \cdot)$ is a permutation on $\mathcal{X}$

Trapdoor One-Wayness: Attack Game



Trapdoors: Key Exchange

- Let's use a one-way trapdoor function scheme $\mathcal{T} = (G, F, I)$, defined over finite sets $(\mathcal{X}, \mathcal{Y})$ to build a secure anonymous key exchange protocol
  - Alice computes $(pk, sk) \leftarrow G()$ and sends pk to Bob
  - Bob picks a random $x \xleftarrow{R} \mathcal{X}$
  - Bob computes $y \leftarrow F(pk, x)$, and sends y to Alice
  - Alice computes $x \leftarrow I(sk, y)$
- The correctness property guarantees that Alice and Bob get the same result!
  - Alice and Bob have agreed on x as their shared secret key
- The transcript consists of pk (sent by Alice) and y (sent by Bob)
  - An eavesdropping adversary has no meaningfully better chance against this protocol than they do against the trapdoor function scheme $\mathcal{T}$ itself
- Problem solved!
  - If we actually had a one-way trapdoor function scheme, that is...

RSA

A plausibly one-way trapdoor permutation scheme.

RSA

- RSA is a trapdoor permutation - the domain and range are the same set!
  - Invented by Rivest, Shamir and Adleman (hence RSA) in 1977
  - That's /rɪˈvɛst/, /ʃəˈmɪər/ and /ˈeɪdəlmən/
- This is basically the only trapdoor permutation we know about
  - There are others, but they're closely related or equivalent to RSA
- RSA is arguably a simpler algorithm than most ciphers that we've looked at!
  - You don't need anything beyond LC maths and modular arithmetic to understand it
  - F and I are one-liners

RSA: Key Generation

- Let's start with key generation for RSA
- RSAGen is a probabilistic algorithm that takes two inputs (system parameters):
  - An integer $\ell > 2$
  - An odd integer $e > 2$ (typically 65,537)

RSA: Key Generation Algorithm

- $\mathrm{RSAGen}(\ell, e)$ proceeds as follows:

$$
\begin{aligned}
&p \xleftarrow{R} \{\text{random } \ell\text{-bit prime} : \gcd(e, p-1) = 1\} \\
&q \xleftarrow{R} \{\text{random } \ell\text{-bit prime} : \gcd(e, q-1) = 1,\ q \neq p\} \\
&n \leftarrow pq \\
&d \leftarrow e^{-1} \bmod (p-1)(q-1) \\
&\textbf{return } (n, d)
\end{aligned}
$$

- Luckily, there's already an efficient algorithm to generate primes of a given size! - Likewise, the extended Euclidean algorithm can compute d efficiently

RSA: Terminology

- n is called an ***RSA modulus***
  - Because we're working with $\mathbb{Z}_n$ (the integers modulo n)
  - Should be at least 2048 bits (i.e. each prime $\ell \geq 1024$ bits); some organisations recommend 4096
- e is called an encryption exponent or ***public exponent***
  - Because RSA encryption is $x^e \mod n$
  - Almost universally $e = 2^{16} + 1 = 65{,}537$
- d is called a decryption exponent
  - Because RSA decryption is $y^d \in \mathbb{Z}_n$
- You've probably noticed that the public key and secret key are related in some way
  - One needs to be able to invert the action of the other
  - So the security of the scheme also relates to the difficulty of figuring out the secret key (the trapdoor) given the public key

RSA: Trapdoor Permutation

- Defining G for $\mathcal{T}_{\text{RSA}}$ is simple now that we have RSAGen
  - $(n, d) \longleftarrow \text{RSAGen}(\ell, e)$
  - $pk \longleftarrow (n, e)$ [e is fixed, but it's still included in the public key]
  - $sk \longleftarrow (n, d)$
  - output (pk, sk)
- For a given public key $pk = (n, e)$, and $x \in \mathbb{Z}_n$, we define
  - $F(pk, x) = x^e \bmod n$
- For a given secret key $sk = (n, d)$, and $y \in \mathbb{Z}_n$, we define
  - $I(sk, y) = y^d \bmod n$
- To break RSA, an adversary would need to figure out d using n and e
  - So they'd need to find p and q by factoring n
- Once the modulus n reaches 2048+ bits (each prime $\approx$ 1024 bits), factoring becomes extremely difficult!
  - Or so we believe...

RSA: Attack Game

- Adapting our one-way attack game to use RSA is simple!
- The challenger and adversary both start off with (ℓ, e) as input parameters
- The challenger computes:
  - $(n, d) \longleftarrow \text{RSAGen}(\ell, e)$
  - $x \xleftarrow{R} \mathbb{Z}_n$
  - $y \longleftarrow x^e \in \mathbb{Z}_n$
- The challenger sends (n, y) to the adversary
- The adversary outputs $\hat{x} \in \mathbb{Z}_n$
- $\mathrm{RSA}_{\mathrm{adv}}[\mathcal{A}, \ell, e] = \Pr[\hat{x} = x]$
- We say that the RSA assumption holds for (n, y) if for all efficient adversaries, the quantity $\mathrm{RSA}_{\mathrm{adv}}[\mathcal{A}, \ell, e]$ is negligible
  - (n, y) is an instance of the RSA problem
  - x is a solution to this instance of the RSA problem

Textbook RSA is Not IND-CPA Secure

- The key exchange protocol follows naturally from the attack game!
  - But textbook RSA has a critical structural limitation
- RSA encryption $c = m^e \mod n$ is deterministic
  - Same plaintext always produces the same ciphertext under the same public key
- This is a general principle: ***any*** deterministic public-key encryption scheme is trivially broken because the adversary holds pk
  - The adversary submits m_0, m_1 and receives $c = m_b^e \mod n$
  - They simply compute $m_0^e \mod n$ themselves (they have pk!) and compare
  - If it matches c, they output $b = 0$; otherwise $b = 1$
  - Advantage = 1, trivially
- This is not a fixable bug in how you use RSA
  - It's a structural limitation of the raw trapdoor permutation
  - The fix (OAEP) adds randomisation and achieves IND-CCA2; we'll see it in Lecture 07

Diffie-Hellman Key Exchange (DHKE)

No mayonnaise jokes, please.

Another Approach

- Let's take a look at a different kind of key exchange at a high level!
  - Alice chooses a random secret α
  - Alice computes $E(\alpha)$ and sends $E(\alpha)$ to Bob
  - Bob chooses a random secret β
  - Bob computes $E(\beta)$ and sends $E(\beta)$ to Alice
  - With the information that each has, Alice and Bob somehow compute $F(\alpha, \beta)$
  - $F(\alpha, \beta)$ is the shared key
- The functions E and F should satisfy these properties:
  - E should be easy to compute
  - $F(\alpha, \beta)$ should be easy to compute given α and $E(\beta)$
  - $F(\alpha, \beta)$ should be easy to compute given $E(\alpha)$ and β
  - $F(\alpha, \beta)$ should be hard to compute given $E(\alpha)$ and $E(\beta)$
- This implies that E is hard to invert! Why?
- Efficient for Alice and Bob, hard for an adversary!

Logarithms

- If we want to implement this key exchange protocol, then we need to find suitable functions E and F
- Logarithms are pretty nasty, right? Let's use those...
- First, pick some base g and let...
  - $E(\alpha) = g^{\alpha}$
  - $F(\alpha, \beta) = g^{\alpha\beta}$
- Notice that our key exchange protocol works both ways...
  - $E(\alpha)^{\beta} = (g^{\alpha})^{\beta} = F(\alpha, \beta) = (g^{\beta})^{\alpha} = E(\beta)^{\alpha}$
- So long as exponentiation is efficient, this works for Alice and Bob!
- And taking logarithms is hard ($\mod p$), so E is difficult to invert
- The problem is that exponentiation creates gigantic numbers very quickly!
  - We need some way to limit the size of the numbers we're working with

Working Modulo a Prime

- The fix for exploding numbers: work modulo a large prime p
  - All values stay in $\{0, 1, \ldots, p-1\}$ – numbers never blow up
  - Going backwards – finding α from $g^\alpha \mod p$ – is the ***discrete log problem***
- Define $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$, the non-zero integers mod p
  - $\mathbb{Z}_p^*$ has $p-1$ elements (all non-zero integers mod a prime are invertible)
  - We multiply and reduce mod p – that's the whole game
- Example: let $p = 11$ and $g = 2$
  - $2^1 = 2,\ 2^2 = 4,\ 2^3 = 8,\ 2^4 = 5,\ 2^5 = 10 \pmod{11}$
  - $2^6 = 9,\ 2^7 = 7,\ 2^8 = 3,\ 2^9 = 6,\ 2^{10} = 1 \pmod{11}$
  - All 10 non-zero elements appear – $g = 2$ ***generates*** the entire group $\mathbb{Z}_{11}^*$
- This is what ***cyclic group*** means: one element, raised to successive powers, produces every group element

Generators and Order

- Not every element generates the whole group!
- Same $p = 11$, but try $g = 3$
  - $3^1 = 3,\ 3^2 = 9,\ 3^3 = 5,\ 3^4 = 4,\ 3^5 = 1 \pmod{11}$ – only 5 elements!
  - $g = 3$ has ***order*** 5: it cycles back to 1 after just 5 steps
  - It generates a subgroup $\{1, 3, 4, 5, 9\}$ – not all of $\mathbb{Z}_{11}^*$
- Why does this happen?
  - $|\mathbb{Z}_{11}^*| = 10 = 2 \times 5$
  - The group has subgroups of every order that divides 10: orders 1, 2, 5, and 10
  - $g = 10$ has order 2 (check: $10^2 = 100 = 1 \bmod 11$) – even worse!
- If an adversary can confine the problem to a small subgroup, discrete log becomes easy
  - An order-5 subgroup? Only 5 values to try. An order-2 subgroup? Trivial
  - This is the ***Pohlig-Hellman attack***: decompose the problem into small-subgroup pieces

Safe Primes and Prime-Order Subgroups

- The fix: choose p so that $p - 1$ has no small odd factors
  - A ***safe prime*** is a prime $p = 2q + 1$ where q is also prime
  - q is called a ***Sophie Germain prime***
  - Now $p - 1 = 2q$, and the only subgroup orders are $1, 2, q$, and $2q$
- We work in $\mathbb{G}$, the subgroup of order q
  - $\mathbb{G} = \{g^a \bmod p \mid a \in \mathbb{Z}_q\}$, with exactly q elements
  - Every non-identity element generates all of $\mathbb{G}$ (because q is prime)
  - No small subgroups to exploit
- Our example: $p = 11 = 2 \times 5 + 1$ is a safe prime, $q = 5$
  - $\{1, 3, 4, 5, 9\}$ is the order-5 subgroup (generated by $g = 3$)
  - Every element except 1 generates the whole subgroup
- All exponents are interpreted modulo q; we treat $\mathbb{Z}_q$ as the exponent space
- In practice, p is 2048+ bits and q is roughly the same size (~2047 bits)

Diffie-Hellman Key Exchange

- Public system parameters: a safe prime p, a generator g of the order-q subgroup $\mathbb{G}$
- The protocol:
  - Alice picks a random secret $\alpha \xleftarrow{R} \mathbb{Z}_q$ and computes $u \leftarrow g^{\alpha}$
  - Alice sends u to Bob
  - Bob picks a random secret $\beta \xleftarrow{R} \mathbb{Z}_q$ and computes $v \leftarrow g^{\beta}$
  - Bob sends v to Alice
  - Alice computes $w \leftarrow v^{\alpha} = g^{\beta\alpha}$
  - Bob computes $w \leftarrow u^{\beta} = g^{\alpha\beta}$
- The shared secret is $w = g^{\alpha\beta}$ – both sides arrive at the same value
- An eavesdropper sees only $u = g^{\alpha}$ and $v = g^{\beta}$ – both discrete log problems
- In practice, w is not used directly as a key
  - We'll see why shortly, and what to do about it

System Parameters

- So how do we pick our modulus and generator?
  - Can we just pick our own at random?
- Don't roll your own parameters
  - Use standardised values for the modulus and generator from RFC 3526
  - Primes of different sizes can have different security implications
  - Poor choices for parameters can have dire consequences!
- The equivalent security level is quite a bit smaller than the bit length of the modulus
  - Don't be fooled into thinking a 1024-bit prime gives 1024 bits of security
  - It's actually insecure by modern standards!
- The best known algorithms for discrete log can handle primes up to 795 bits (as of 2019)
  - In practice, we need primes of at least 2048 bits!
  - And quite a bit larger again for high-security apps

Diffie-Hellman Assumption

- The security of DHKE rests on a formal hardness assumption about $\mathbb{G}$
  - Or ***finite-field Diffie-Hellman***, to be precise
- Security depends on good choices for the modulus and generator
- We're also making one big assumption!
  - Given $g^\alpha, g^\beta \in \mathbb{G}$, where $\alpha, \beta \in \mathbb{Z}_q$
  - It should be hard to compute $g^{\alpha\beta} \in \mathbb{G}$
  - So an eavesdropper can't compute the shared secret $g^{\alpha\beta}$ from the transcript
- This is the computational Diffie-Hellman assumption (CDH)
  - It's based on the discrete logarithm assumption (DL)
  - It's assumed to be hard to invert the discrete exponentiation that we're doing
  - CDH is a stronger assumption, but seems to hold everywhere DL does
- But just like with RSA, there's no formal proof of either of these!

Key Derivation

From shared secrets to usable keys.

Why Not Use the Shared Secret Directly?

- The DH shared secret $w = g^{\alpha\beta}$ is a group element, not a uniformly random bitstring
  - Its distribution depends on the structure of $\mathbb{G}$
  - Using it directly as an AES key would be insecure
- Even if we extract the raw bytes, the entropy is not uniformly spread
  - Some bit positions may be biased or correlated
- We need a ***key derivation function (KDF)*** to extract uniform keying material
  - And potentially derive multiple keys from a single shared secret (e.g. one for encryption, one for MAC)

HKDF: Extract-then-Expand

- HKDF (RFC 5869) is a two-stage KDF built from HMAC (Lecture 05)
- ***Extract***: $\mathrm{PRK} \leftarrow \mathrm{HMAC}(\mathrm{salt}, \mathrm{IKM})$
  - Input keying material (IKM): the raw shared secret (high-entropy but non-uniform)
  - Salt: optional, public, ideally random
  - Output: a pseudorandom key (PRK) that is uniformly distributed
- ***Expand***: $\mathrm{OKM} \leftarrow \mathrm{HMAC}(\mathrm{PRK}, \mathrm{info} \| \mathrm{counter})$
  - Takes the PRK and derives one or more output keys of the desired length
  - The info parameter binds the derived key to its context (protocol, party, purpose)
- Used in TLS 1.3, HPKE, Signal, WireGuard

HKDF vs Password-Based KDFs

- HKDF assumes ***high-entropy input*** (like a DH shared secret or a random seed)
  - Fast: two HMAC calls
  - Not designed to resist brute-force attacks on the input
- Password-based KDFs (PBKDF2, Argon2id) assume ***low-entropy input*** (passwords)
  - Deliberately slow and memory-hard to resist offline dictionary attacks
  - We'll see these in Lecture 08
- The distinction matters
  - Using HKDF on a password gives no brute-force resistance
  - Using Argon2id on a DH shared secret wastes resources for no security gain
- For the EPIC project: HKDF is the right tool for DH-derived keys; Argon2id for user passwords

A Note on Quantum Computing

- Shor's algorithm solves both integer factorisation and discrete logarithms in polynomial time on a quantum computer
- RSA, finite-field DH, and elliptic curve DH are all broken by a sufficiently large quantum computer
- Current quantum computers are too small and noisy to attack real key sizes, but engineering progress is continuous
- The "harvest now, decrypt later" threat: adversaries record encrypted traffic today and decrypt it once quantum hardware matures
- NIST has standardised post-quantum alternatives: ML-KEM (FIPS 203) for key encapsulation, ML-DSA (FIPS 204) for signatures
- We'll cover post-quantum cryptography in detail in Lecture 09

Elliptic Curves and ECDH

More security per bit.

Elliptic Curve Cryptography

- An elliptic curve E over a finite field $\mathbb{F}_p$ is the set of points (x, y) satisfying:
  - $y^2 = x^3 + ax + b \mod p$, for some prime $p > 3$
  - Plus a special point $\mathcal{O}$ called the point at infinity
- If $E/\mathbb{F}_p$ is an elliptic curve...
  - Then $E(\mathbb{F}_p)$ is the set of points on that curve defined over $\mathbb{F}_p$
  - $E(\mathbb{F}_p)$ turns out to be a finite abelian group
  - If $E(\mathbb{F}_p)$ is also cyclic...
  - ...and the discrete log problem is hard in the group $E(\mathbb{F}_p)$...
  - ...we could swap it out and use it in Diffie-Hellman instead!

The Group Law

- Point addition gives elliptic curves their group structure
  - To add two points P and Q: draw a line through them, find the third intersection with the curve, reflect across the x-axis
  - Point doubling ($P + P$): use the tangent line at P
  - The point at infinity $\mathcal{O}$ acts as the identity element
- Scalar multiplication: $nP = P + P + \cdots + P$ (n times)
  - Efficient via double-and-add (analogous to square-and-multiply for modular exponentiation)
  - Computing nP given n and P is fast
  - Recovering n given P and nP is the elliptic curve discrete log problem (ECDLP)

Why Elliptic Curves?

- The discrete log problem is harder on elliptic curves than in $\mathbb{Z}_p^*$
  - The number field sieve (used to attack $\mathbb{Z}_p^*$) does not apply to elliptic curves
  - Best known attack is Pollard's rho: $O(\sqrt{q})$ where q is the group order
- This means much smaller keys for the same security level
  - 256-bit ECC $\approx$ 3072-bit RSA $\approx$ 128-bit symmetric security
  - Smaller keys, faster operations, less bandwidth
  - A significant practical advantage for constrained environments and network protocols
- But we can't just pick any curve! It must be specifically vetted to be secure
  - Bad curve choices can introduce weaknesses

X25519 and Curve25519

- Curve25519 is one of the most widely used elliptic curves
  - Designed by Daniel Bernstein with security and performance as primary goals
  - Equation: $y^2 = x^3 + 486662x^2 + x$ over $\mathbb{F}_p$ where $p = 2^{255} - 19$
- X25519 is the Diffie-Hellman function using Curve25519
  - Takes a 32-byte secret key and a 32-byte public key, outputs a 32-byte shared secret
  - Used in TLS 1.3, Signal, WireGuard, SSH, and many other protocols
- Why Curve25519 over NIST curves?
  - “Safe curve” criteria: no special structure that could enable attacks
  - All parameters are mathematically justified, with nothing arbitrary
  - NIST curves use seed parameters whose origins are not fully explained
  - This created trust concerns after Dual EC DRBG (a NIST-approved RNG later revealed to contain an NSA-exploitable weakness) shook confidence in NIST-selected parameters
- For the EPIC project, X25519 is the recommended KEM in the HPKE ciphersuite

Public Key Encryption

Putting it all together.

Back to Basics

- Let's return to the problem of encryption! Key exchange is possible using public and secret keys, so is it possible to build an encryption scheme like this as well?
  - Can we send and receive encrypted messages without sharing a secret key?
- As it turns out, we can! Public key encryption is similar to what we've seen already
- The recipient, Bob, runs a probabilistic key generation algorithm in advance
  - $(pk, sk) \longleftarrow G()$, where pk is the public key and sk is the secret key
- To send a message to Bob, Alice needs his contact details and his public key
- Alice computes the ciphertext for her message using the public key
  - $c \longleftarrow E(pk, m)$, where E is a probabilistic encryption function
- When Bob receives the ciphertext, he decrypts it using his secret key
  - $m \longleftarrow D(sk, c)$, where D is a deterministic decryption function
- Public key encryption is also called asymmetric encryption, because encryption and decryption use different keys

Public Key Encryption: Definition

- Let's define a public-key encryption scheme $\mathcal{E} = (G, E, D)$ as a triple of efficient algorithms:
- Key Generation: A probabilistic algorithm that outputs a pair (pk, sk):
  - Output: public key pk, secret key sk
- Encryption: A probabilistic algorithm invoked as $c \leftarrow E(pk, m)$
  - Input: Public key pk, message m
  - Output: Ciphertext c
- Decryption: A deterministic algorithm invoked as $m \leftarrow D(sk, c)$
  - Input: secret key sk, ciphertext c
  - Output: message m or a special `reject` symbol
- The correctness requirement states that decryption must undo encryption:
  - For all key pairs (pk, sk) and all messages m
    - Pr[$D(sk, E(pk, m)) = m$] = 1

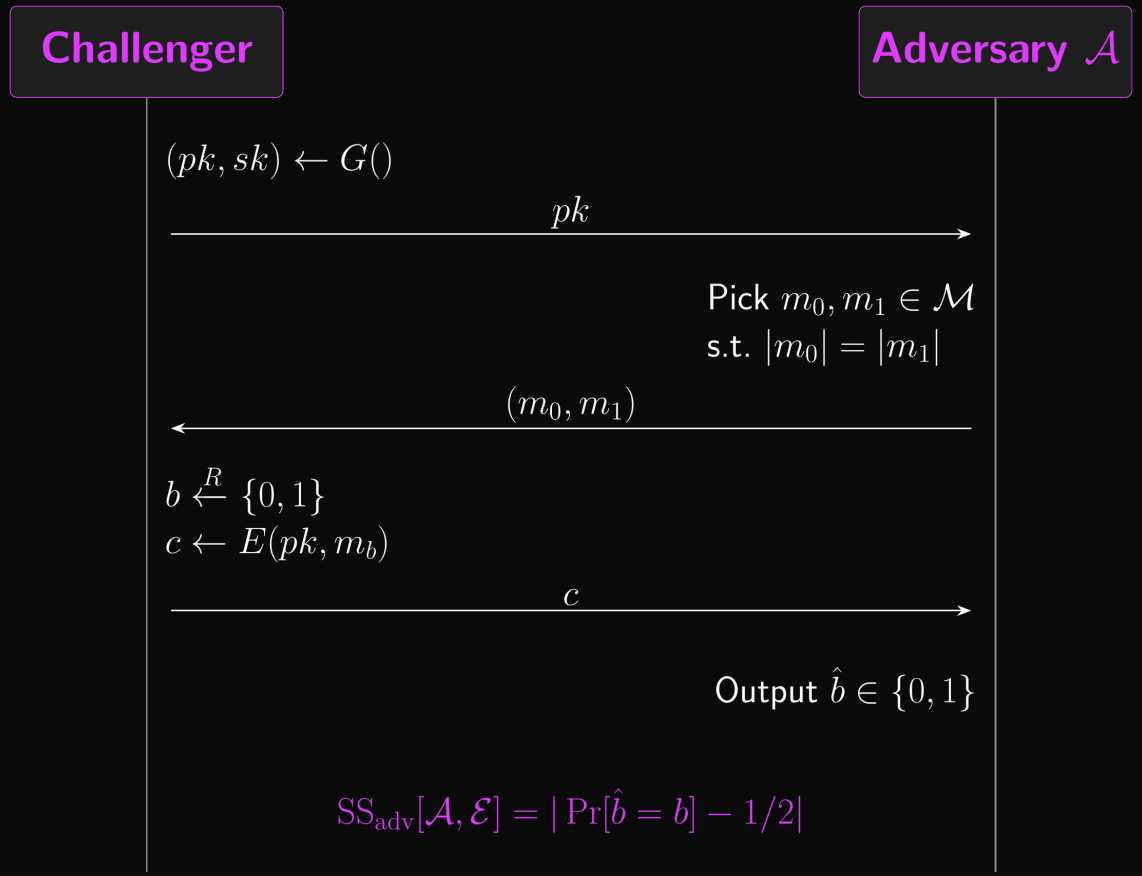
Public Key Encryption: Attack Game

- A public-key encryption scheme $\mathcal{E}$ is semantically secure if no efficient adversary can distinguish between encryptions of any two messages of the same length.
  - Let's demonstrate with an attack game with two experiments, 0 and 1!
- The challenger starts by generating public and secret keys $(pk, sk) \longleftarrow G()$
  - The challenger sends pk to the adversary $\mathcal{A}$
- $\mathcal{A}$ selects two messages of the same length, m_0 and m_1
  - $\mathcal{A}$ sends both messages to the challenger
- In experiment b, the challenger computes $c \longleftarrow E(pk, m_b)$
  - The challenger sends c to the adversary $\mathcal{A}$
- $\mathcal{A}$ outputs a guess $\hat{b}$ depending on which message they think was encrypted
- $\mathcal{A}$'s advantage is $\mathrm{SS}_{\mathrm{adv}}[\mathcal{A}, \mathcal{E}] = |\Pr[\hat{b} = b] - 1/2|$
  - If $\mathcal{E}$ is semantically secure, then $\mathrm{SS}_{\mathrm{adv}}[\mathcal{A}, \mathcal{E}]$ should be negligible for all efficient adversaries $\mathcal{A}$

Public Key Encryption: CPA for Free

- It's worth noting that the adversary has the public key
  - They can encrypt arbitrary messages and obtain ciphertexts for them!
- For public key encryption, semantic security implies security under chosen plaintext attack (CPA)
  - It's trivial to show that a Q-query adversary's advantage is equivalent to the basic version we just described
  - This isn't the case for symmetric ciphers!
- This also underlines why semantically secure public key encryption algorithms must be randomized
  - If every plaintext encrypts to a single ciphertext under a key pk...
  - $\mathcal{A}$ wins every time by simply encrypting m_0 or m_1, then comparing them with the ciphertext sent by the challenger!

Public Key Encryption: Attack Game



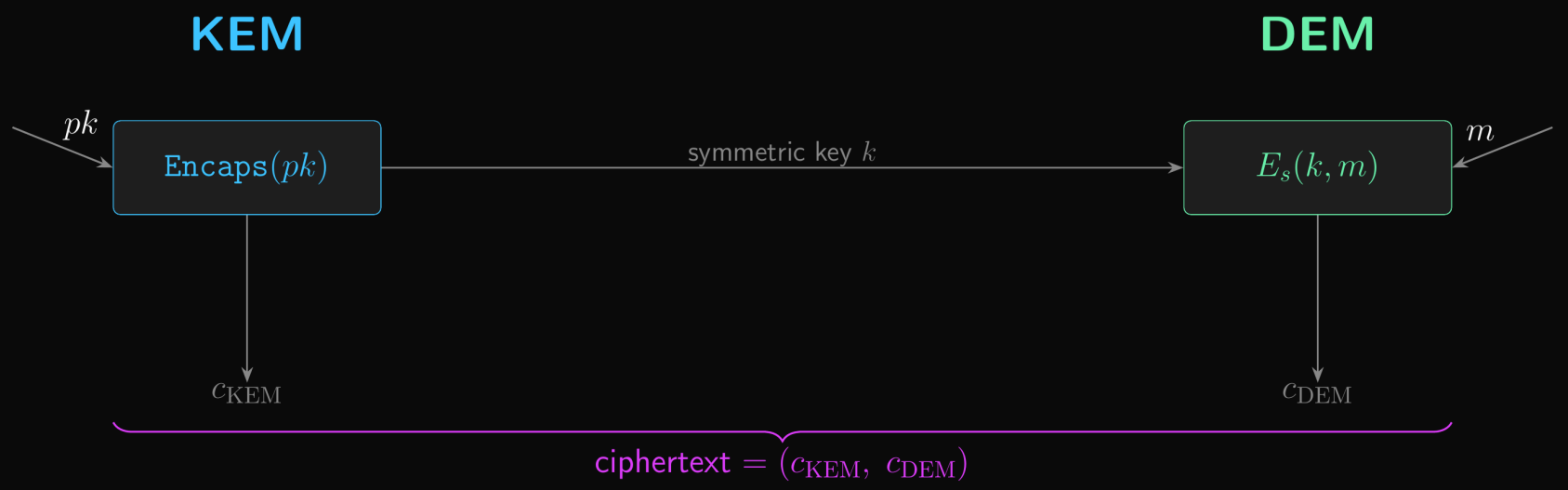
PKE from Trapdoor Functions

- We can build a public-key encryption scheme $\mathcal{E}_{\text{TDF}}$ from any TDF $\mathcal{T} = (G, F, I)$ and a hash function H
- ***Encrypt*** (pk, m):
  - $x \xleftarrow{R} \mathcal{X}$
  - $y \leftarrow F(pk, x)$
  - $k \leftarrow H(x)$
  - $c \leftarrow E_s(k, m)$ [symmetric encryption]
  - output (y, c)
- ***Decrypt*** $(sk, (y, c))$:
  - $x \leftarrow I(sk, y)$
  - $k \leftarrow H(x)$
  - $m \leftarrow D_s(k, c)$ [symmetric decryption]
  - output m
- The TDF hides x; the hash H extracts a symmetric key from x; then symmetric encryption handles the message

Hybrid Cryptosystems

- $\mathcal{E}_{\mathrm{TDF}}$ already reveals the pattern: public-key crypto handles a short secret; symmetric crypto handles the data
  - After that, symmetric encryption takes over for the bulk of the work
- This concept is formalised by the idea of a key encapsulation mechanism (KEM)
  - The KEM generates and encrypts a random symmetric key in one step
- In $\mathcal{E}_{\mathrm{TDF}}$, the (x, y) pair is exactly a KEM!
  - Instead of encrypting a message directly, we encrypt a random key (used for symmetric encryption) to send to the other party
- This gives a clean separation of concerns in a hybrid cryptosystem
  - KEM handles key distribution
  - Data encapsulation scheme (symmetric cipher) handles data encryption
- Pretty much every popular use of public key encryption follows this pattern!
  - TLS, SSH, OpenPGP and HPKE (RFC 9180)

KEM/DEM Hybrid Encryption



Decrypt: $k \leftarrow \text{Decaps}(sk, c_{\text{KEM}}), \quad m \leftarrow D_s(k, c_{\text{DEM}})$

ElGamal Encryption

One slide, one result.

ElGamal

- ElGamal is our first IND-CPA-secure public-key encryption scheme (unlike textbook RSA!)
  - Key generation: choose secret $\alpha \xleftarrow{R} \mathbb{Z}_q$, publish $pk = g^\alpha$
  - Encrypt $m \in \mathbb{G}$: pick random $\beta \xleftarrow{R} \mathbb{Z}_q$, output $(g^\beta, m \cdot (pk)^\beta)$
  - Decrypt (u, v): compute $m = v \cdot u^{-\alpha}$
- The ciphertext is randomised (fresh β each time), so it can be IND-CPA secure
- Security reduces to the decisional Diffie-Hellman (DDH) assumption
  - DDH: given $(g, g^\alpha, g^\beta, Z)$, it's hard to tell if $Z = g^{\alpha\beta}$ or random
  - DDH is a stronger assumption than CDH: DDH hard $\Rightarrow$ CDH hard $\Rightarrow$ DL hard
  - Believed to hold in suitable groups (e.g. prime-order subgroups of $\mathbb{Z}_p^*$, elliptic curves)
- ElGamal illustrates the reduction template: scheme security reduces to a hardness assumption
  - This pattern recurs in Lecture 07 when RSA-FDH security reduces to RSA in the ROM

Forward Secrecy

Protecting past sessions from future compromises.

Static vs Ephemeral Keys

- So far we've discussed key exchange using long-term (static) keys
  - If a long-term key is compromised, all past sessions using that key are exposed
  - An adversary who records ciphertexts and later obtains the key can decrypt everything
- Concrete scenario: a server's TLS private key is leaked via a backup, a breach, or a subpoena
  - An adversary who recorded encrypted traffic for the past year can now decrypt *all* of it
  - This is not hypothetical: it's exactly the threat model behind "harvest now, decrypt later"
- ***Forward secrecy***: compromise of long-term keys does not compromise past session keys
  - This requires ephemeral keys: fresh key pairs generated for each session
  - After the session ends, the ephemeral secret key is destroyed

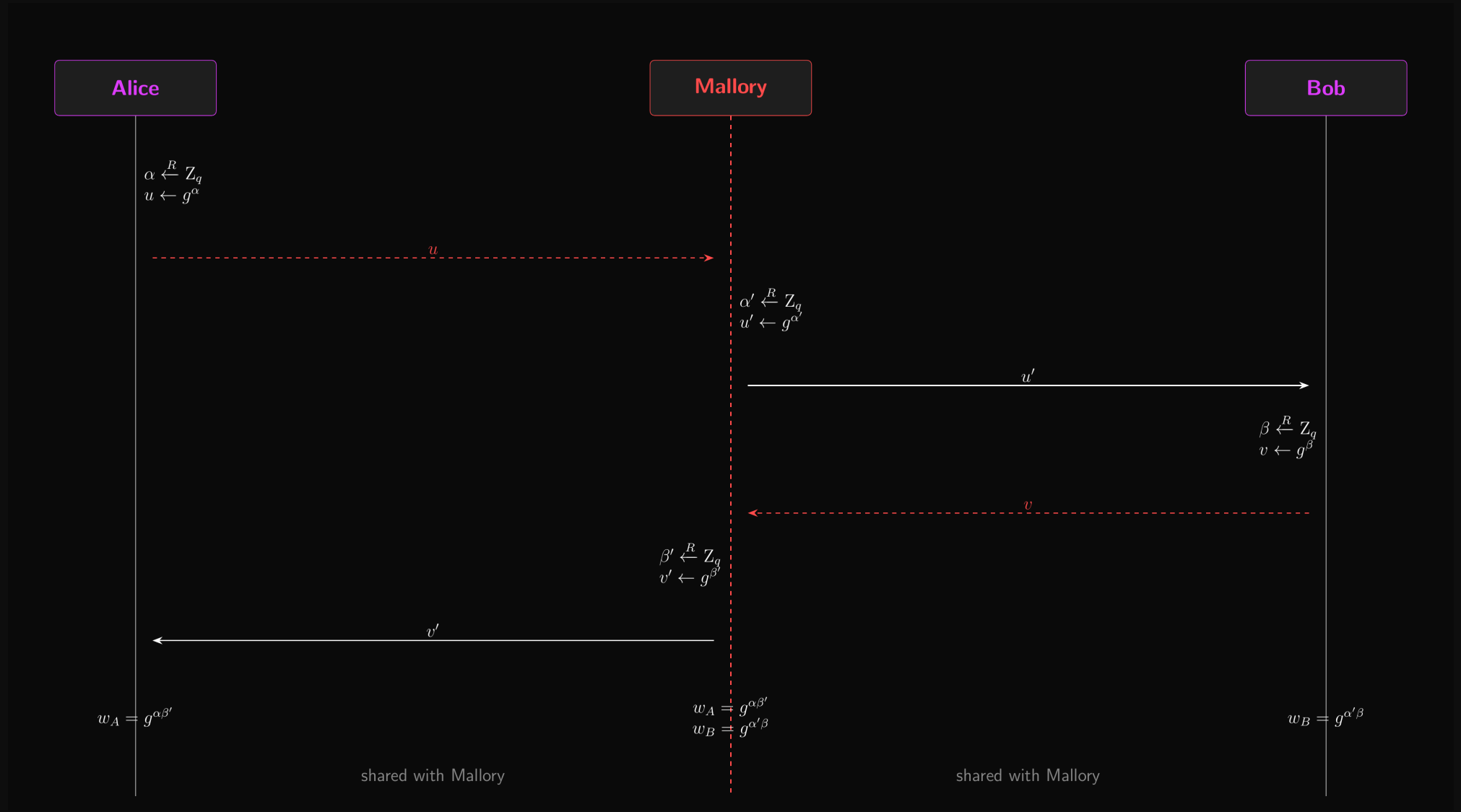
A Note on Terminology

- Different sources use different names for the same two categories of key
  - We'll use ***static*** and ***ephemeral*** throughout this module
- Common synonyms for ***static*** keys (long-lived, reused across sessions):
  - ***Long-term keys*** – the most common alternative; used in Boneh & Shoup and most RFCs
  - ***Identity keys*** – Signal protocol terminology (the "IK" in X3DH)
- Common synonyms for ***ephemeral*** keys (fresh per session, destroyed after use):
  - ***Temporary keys*** or ***transient keys*** – informal usage in some textbooks
  - ***One-time prekeys*** – Signal-specific (the "OPK" in X3DH); generated in advance, used once
- Watch out for ***session key*** – usually means the derived shared secret (e.g. K in 3DH), not the ephemeral DH key pair itself
- Some protocols also have ***semi-static*** keys: rotated periodically (e.g. weekly) but not per-session
  - Signal's "signed prekey" (SPK) is a semi-static key – refreshed regularly but reused across multiple sessions

Man-in-the-Middle

- We've very explicitly stated that these protocols are anonymous
  - And secure against eavesdropping where the assumptions hold
  - It's implied that we're not secure against active attackers!
- Both the RSA key exchange and DHKE protocols we've sketched fail instantly against a man-in-the-middle attack precisely because of that anonymity
  - All Mallory has to do is intercept all of Alice and Bob's messages
  - Mallory generates his own random secrets and continues the key exchange
  - With Alice, Mallory impersonates Bob and continues the protocol
    - Mallory now has a shared secret with Alice (who thinks he's Bob)
  - With Bob, Mallory impersonates Alice and continues the protocol
    - Mallory now has a shared secret with Bob (who thinks he's Alice)
  - Mallory can now communicate with both parties, and Alice and Bob are unaware that they've been compromised!

Man-in-the-Middle Attack



Trust

- The problem here doesn't lie with DHKE itself... we did exchange a key, after all!
  - Should we really trust public keys sent over insecure channels?
- You're trusting public keys from websites you browse right now!
  - How do you know this is safe? You need some kind of trusted third party...
- Public keys for websites are signed by a certificate authority (CA)
  - It's a cryptographic stamp of approval!
  - You can verify the signature and be sure that the public key is authentic
- Note that what we're verifying is that the person you're talking to is who they claim to be... not that they're a trustworthy person!
  - downloadmoreram.com is unlikely to give you more RAM, legit or not
- We'll look at PKI and certificates properly in Lecture 07

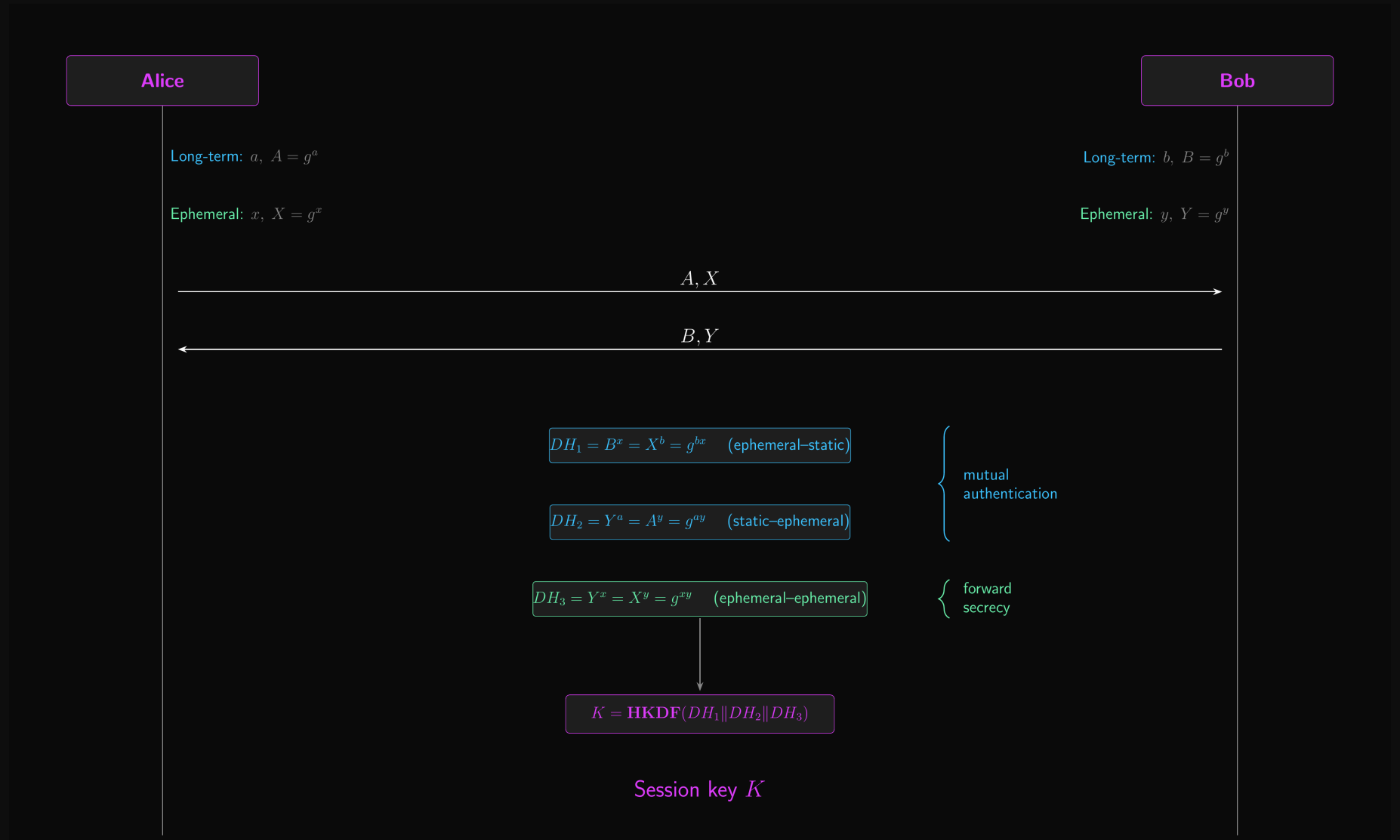
Triple Diffie-Hellman (3DH)

- Let's fix DHKE and get authenticated key exchange with ***forward secrecy***
  - Let's assume that we've already agreed system parameters g and p
  - Let's use conventional notation this time - $PUBLIC$ and $private$
- Long before we start, Alice and Bob generate long-term identity key pairs
  - Alice generates $A = g^a \mod p$ and publishes her public key A
  - Bob generates $B = g^b \mod p$ and publishes his public key B
  - The public keys may be made available through some kind of PKI
  - Their purpose is to prove that Alice and Bob are who they say they are!
- Alice and Bob generate fresh ephemeral key pairs for each session
  - Alice generates $X = g^x \mod p$ and publishes her public key X
  - Bob generates $Y = g^y \mod p$ and publishes his public key Y
  - Their purpose is to provide forward secrecy
  - Past sessions aren't at risk if this session's key is compromised!

Triple Diffie-Hellman: Session Key

- Now we have to derive a session key K using three DH exponentiations!
- Alice computes:
  - $DH_1 = B^x = (g^b)^x = g^{bx} \mod p$
  - $DH_2 = Y^a = (g^y)^a = g^{ya} = g^{ay} \mod p$
  - $DH_3 = Y^x = (g^y)^x = g^{yx} = g^{xy} \mod p$
  - $K = \mathrm{HKDF}(DH_1 \| DH_2 \| DH_3)$
- Bob computes:
  - $DH_1 = X^b = (g^x)^b = g^{xb} = g^{bx} \mod p$
  - $DH_2 = A^y = (g^a)^y = g^{ay} \mod p$
  - $DH_3 = X^y = (g^x)^y = g^{xy} \mod p$
  - $K = \mathrm{HKDF}(DH_1 \| DH_2 \| DH_3)$
- DH_3 uses only ephemeral keys – once x and y are destroyed after the session, DH_3 (and thus K) cannot be reconstructed
  - Even if long-term keys a and b later leak, past session keys can't be compromised

Triple Diffie-Hellman (3DH)



Why TLS 1.3 Mandates Ephemeral Key Exchange

- TLS 1.3 removed support for static RSA and static DH key exchange
  - Only ephemeral (EC)DH is allowed
- The reason: forward secrecy is not optional for modern protocols
  - A compromised server key should not retroactively expose years of recorded traffic
  - With static RSA, anyone who obtains the server's private key can decrypt all past sessions
  - With ephemeral DH, each session uses a fresh key pair that is destroyed after use

Digital Signatures

Authentication with public keys.

The Authentication Gap

- We've solved key exchange (DH, ECDH) and public-key encryption (ElGamal, KEM)
- But everything is anonymous: no authentication
- MACs provide message authentication, but require a shared secret key
  - Any keyholder can produce a tag
  - Bob can verify Alice's tag, but he could have produced it himself
- Can we do message authentication using only public keys?
  - Yes: digital signatures

Non-Repudiation

- Digital signatures provide a property MACs cannot: ***non-repudiation***
  - Only the signer (secret key holder) can produce a valid signature
  - Anyone with the public key can verify it
  - Once Alice signs, she cannot credibly deny it to a third party
- This is strictly stronger than MAC authentication
  - Not always desirable: sometimes deniability is a feature, not a bug
  - In law, electronic signature $\neq$ cryptographic digital signature

Signatures in Practice

- Software distribution: Microsoft signs updates with its secret key
  - Customers verify with the public key before installing
  - No shared keys, no trusted third party for end users
- Authenticated email (DKIM): domains sign outgoing emails
  - Public key published in a DNS record; recipients verify sender authenticity
- Closing the MITM loop: Alice signs her DH public value
  - Bob verifies the signature with Alice's long-term public key
  - This is exactly what certificates and TLS 1.3 do
  - Certificate authorities (CAs) provide the trust anchor

Coming Up in Lecture 07

- Formal signature scheme definition: the (G, S, V) triple
  - Mirrors the MAC structure from Lecture 04
  - Security game (UF-CMA) transfers directly
- Signature constructions: RSA-FDH, Schnorr, ECDSA
- CCA security: what happens when the adversary can submit ciphertexts for decryption?
  - RSA-OAEP closes the loop on textbook RSA's IND-CPA weakness
- Authenticated encryption: formal proof of the Encrypt-then-MAC theorem from Lecture 04

Conclusion

What did we learn?

What Did We Learn?

- We solved the key distribution problem that every symmetric primitive left open
  - Trapdoor functions and RSA: public-key key exchange from modular arithmetic
  - Diffie-Hellman: key agreement from cyclic groups and the CDH assumption
  - Elliptic curves: the same idea with smaller keys and stronger security per bit
- We introduced HKDF for turning raw shared secrets into usable keying material
  - Extract-then-expand, built from HMAC
  - Distinct from password-based KDFs (which we'll see in Lecture 08)
- We saw how public-key encryption, KEMs and hybrid cryptosystems work
  - ElGamal: IND-CPA from DDH
  - KEM/DEM: the pattern behind TLS, SSH, HPKE
- Forward secrecy and 3DH protect past sessions from future key compromise
- Digital signatures fill the authentication gap that MACs cannot

Where Do We Go from Here?

- Next lecture: formal signature definitions, CCA security, authenticated encryption and HPKE
  - The (G, S, V) triple and UF-CMA security game
  - RSA-FDH, Schnorr and ECDSA constructions
  - IND-CCA2 and RSA-OAEP
  - AEAD, AES-GCM, and HPKE (RFC 9180)

Questions?

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