Week 2

Nadav Kohen

June 29, 2025

Now that we have introduced some of the basic constructions and vocabulary from public key cryptography, our goal this week is to begin our path towards proving that protocols are secure. This will require us to practice mathematical proofs, and it will also require definitions that model our notions of security, which will in turn require some basic probability theory. We will tackle some simpler topics this week in hopes of studying the security of Schnorr signatures next week.

To prove a statement P is true, assume the opposite, i.e., ~P, and then show that this assumption leads to a contradiction.
Thus, ~P => contradiction implies P must be true.

Perhaps the most prevalent proof technique in cryptography is proof by reduction, which is itself a special case of proof by contrapositive. In logic, the contrapositive of an implication $P \Rightarrow Q$ is the equivalent implication $\neg Q \Rightarrow \neg P$, where $\neg$ denotes negation (i.e., $\neg Q$ is read "not Q"). These implications are logically equivalent because each of them is true if and only if P is false or Q is true. Hence, if one is trying to prove some conclusion, Q, from some assumption, P, proof by contrapostive is the method of assuming that the conclusion is false and arguing that this forces the assumptions to be false. In cryptography, we often want to show that if we assume that problem X is hard to solve, then another problem, Y, is also hard to solve. A proof by reduction in this context is a method of proving such claims by showing that if Y has an efficient solution, then we can use this solution as a resource to construct an efficient solution for X.

Proof Method	What You Want to Prove	What You Assume	What You Show	Logical Form	Typical Use Case
Contradiction	A statement P	$\neg P$ (the opposite)	That $\neg P$ leads to a contradiction	$\neg P \Rightarrow \text{False} \Rightarrow P$	Non-existence, irrationality, uniqueness proofs
Contrapositive	$P \Rightarrow Q$	$\neg Q$	Derive $\neg P$	$\neg Q \Rightarrow \neg P$	Logical implications, number theory
Reduction to Absurdity	A statement P	$\neg P$	Leads to absurd/ impossible result (contradiction)	Same as contradiction	Often used interchangeably with contradiction

Discrete Logarithm (DLP) Assumption

Let G be a cyclic group of prime order q, and let $g \in G$ be a generator. Given g and $h = g^x \in G$, the **Discrete Logarithm Problem (DLP)** is to find $x \in \mathbb{Z}_q$.

One-More Discrete Logarithm (OMDL) Assumption

Let $g \in G$ be a generator of a cyclic group of prime order q. Let $\mathcal{A}$ be a PPT adversary with access to:

- a challenge oracle $\mathcal{O}_{\text{Chal}}$: returns random elements $h_i \in G$
- a discrete log oracle $\mathcal{O}_{\text{DL}}$: returns $\log_g(h)$

The challenger gives $k+1$ random group elements $h_1, \ldots, h_{k+1}$. The adversary may make at most k queries to $\mathcal{O}_{\text{DL}}$.

Goal: Compute $x_i \in \mathbb{Z}_q$ such that $g^{x_i} = h_i$ for all $i = 1, \ldots, k+1$.

Let's take a moment to understand this through a concrete example:

Recall that the Discrete Log (DL) assumption for a group $\mathbb{G}$ is that it is hard to compute x given g^x as input, where g is the generator for $\mathbb{G}$. A related assumption, known as the One More Discrete Log (OMDL) assumption for $\mathbb{G}$, states that if a program is given some (positive) number, k, of random group elements, $g^{x_1}, g^{x_2}, \ldots, g^{x_k}$, and it is able to ask for and receive discrete logarithms for any $k-1$ group elements, then computing all of the values $x_1, x_2, \ldots, x_k$ is still hard. We can use reduction to see that if the OMDL assumption holds for a group, then the DL assumption holds. Specifically, this proof by reduction requires us to show that if the DL problem is easy to solve, then it will be easy to solve the OMDL problem. But of course, if we can compute discrete logarithms in a group with non-negligible probability (i.e., DL problem is easy), then we can trivially solve the OMDL problem by asking for the discrete logs for $g^{x_1}, \ldots, g^{x_{k-1}}$ and then using our DL-solver to directly compute the kth discrete log with non-negligible probability.

Exercise 1. *Let $\mathbb{G}$ be an abelian group with an element, $g \in \mathbb{G}$, of order n. A Schnorr signature of a message, m, by a private key, x, is a pair (s, R) such that $s = k + H(R, m) \cdot x \in \mathbb{Z}/n\mathbb{Z}$ and $R = g^k \in \mathbb{G}$ is the nonce (k is a random element of $\mathbb{Z}/n\mathbb{Z}$ generated for this signature). Intuitively, this works because s commits to the message using a hash, uses the private key, but it does not reveal the private key because of the random value k. A Schnorr signature can be verified by checking the equation $g^s == R \cdot X^{H(R,m)}$, where $X = g^x$ is the signer's public key.*

Use proof by reduction to show that if we assume that the discrete log problem is hard in $\mathbb{G}$, then we can conclude that it is hard to forge two Schnorr signatures for the same nonce and key, that is, if we are given a public key $X = g^x$ and nonce R (without being given x or k) it is hard to compute two different messages m_1 and m_2 and values s_1 and s_2 such that (s_1, R) and (s_2, R) are valid signatures for X of m_1 and m_2, respectively.

This will later be an important part of the proof of the stronger statement that if the discrete

log problem is hard, then Schnorr signatures are unforgable in a much stronger sense.

In the above exercise, we are working with an informal notion of what it means for something to be "hard" and for something to be "easy," and we will now want to make this more precise.

In modern cryptography, when we say something like "Computing the DL is hard", we formalize this as a game between adversary and the challenger

In cryptographic assumption definitions, we make these kinds of statements precise and functional by formalizing them as attack games. In an attack game, we model the adversary as a program that interacts with another program, the challenger (aka the environment, aka the context). In this case, we have to define a very simple game where the adversary is challenged to compute the discrete log of a group element (where g is a fixed generator), as depicted in the attack game $\mathsf{DL}^{\mathcal{A}}(\lambda)$ in Figure 1. Note that this game is parameterized by λ, which is the key size. We take this opportunity to introduce two other common cryptographic assumptions relating to last week's Diffie-Hellman key exchange. Recall that in the Diffie-Hellman key exchange, each party has a public key, $X = g^x$ and $Y = g^y$, and then their shared secret is $Z = g^{x \cdot y}$. The Computational Diffie-Hellman (CDH) assumption is that it is "hard" to compute Z given only X and Y, this is shown in the attack game $\mathsf{CDH}^{\mathcal{A}}(\lambda)$ in Figure 1. Lastly, there is a variation of the CDH assumption called the Decisional Diffie-Hellman (DDH) assumption, which states that is it "hard" to distinguish between a Diffie-Hellman shared secret and a truly random group element (given access to the public keys X and Y); this is made precise in the attack game $\mathsf{DDH}^{\mathcal{A}}(\lambda)$ in Figure 1.

One who tries to break the system

the one who sets up the challenge, usually by sampling random secrets and computing public values

For each of these games, we define the advantage an adversary program, $\mathcal{A}$, has to be

$$\mathsf{Adv}^{\mathrm{dl}}_{\mathcal{A}}(\lambda) := \Pr\left[\mathsf{DL}^{\mathcal{A}}(\lambda) = \text{true}\right],$$

$$\mathsf{Adv}^{\mathrm{cdh}}_{\mathcal{A}}(\lambda) := \Pr\left[\mathsf{CDH}^{\mathcal{A}}(\lambda) = \text{true}\right],$$

$$\mathsf{Adv}^{\mathrm{ddh}}_{\mathcal{A}}(\lambda) := \Pr\left[\mathsf{DDH}^{\mathcal{A}}(\lambda) = \text{true}\right] - \frac{1}{2},$$

Because even the random guess would be right 50% of the time, the adversary must do better than that for the advantage to be meaningful

where $\Pr[\cdot]$ is the probability function, which we will discuss in more detail later. Note that

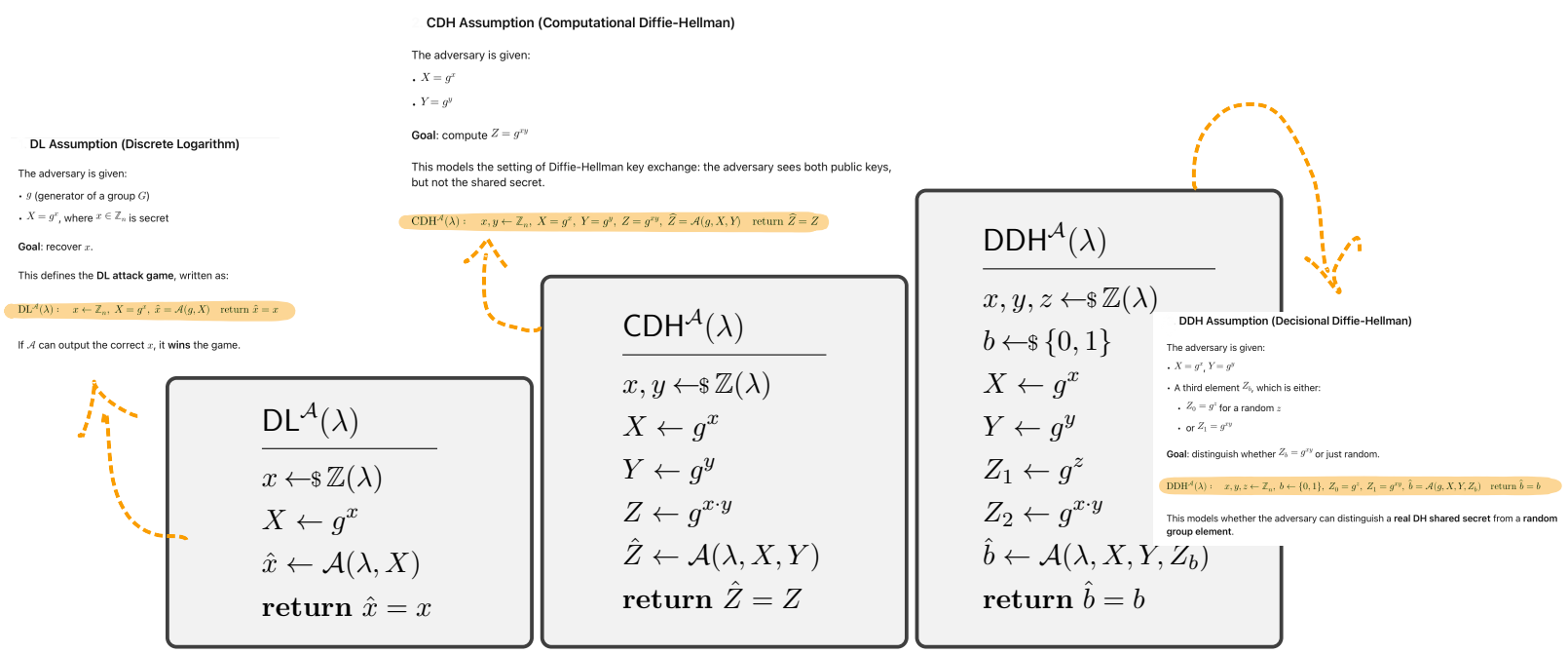

Figure 1: Attack games for the DL, CDH, and DDH cryptographic assumptions. ($\mathbb{Z}(\lambda)$ is $\mathbb{Z}_p$ for a prime, $p > 2^\lambda$, which makes λ the key size, g is a fixed generator for the group $\mathbb{G}(\lambda)$ of size p, and the symbol \$ means chosen uniformly at random).

because an adversary that just picks its value randomly wins the $\mathsf{DDH}^{\mathcal{A}}(\lambda)$ game half of the time, we define the advantage an adversary has to be how much better than probability $\frac{1}{2}$ it can do. Finally, we can make our assumptions formal and complete by saying that the DL assumption holds for a family of groups, $\mathbb{G}(\lambda)$, if for all programs, $\mathcal{A}$, the value $\mathsf{Adv}^{\mathrm{dl}}_{\mathcal{A}}(\lambda)$ is negligible. Similarly, we say that the CDH or DDH assumptions hold if $\mathsf{Adv}^{\mathrm{cdh}}_{\mathcal{A}}(\lambda)$ or $\mathsf{Adv}^{\mathrm{ddh}}_{\mathcal{A}}(\lambda)$ are negligible for all programs, $\mathcal{A}$. We will formalize what it means for a function of λ to be negligible next week, but for now just think of it as a value that becomes small exponentially fast as λ increases.

Exercise 2. *Use proof by reduction and the precise definitions above to show that*

(a) If the DDH assumption holds, then the CDH assumption holds.

(b) If The CDH assumption holds, then the DL assumption holds.

In order to solve this part, you will need two facts that we will prove later:

(1) If you have two independent events A and B, then $Pr[A \text{ and } B] = Pr[A] \cdot Pr[B]$.

(2) If $F(\lambda)$ is not negligible, then $F(\lambda)^2$ is also not negligible.

By definition of negligibility, $F(\lambda)$ is negligible if $\forall c > 0\ \exists \lambda_0 \in \mathbb{N}$ such that $\forall \lambda > \lambda_0$, $F(\lambda) < \frac{1}{\lambda^c}$. Since $F(\lambda)$ non-negligible $\Rightarrow F(\lambda) \geq \frac{1}{\lambda^c}$ ---- ①

① $^2 \Rightarrow F(\lambda)^2 \geq \frac{1}{\lambda^{2c}}$ This inequality holds for infinitely many λ. Since $\frac{1}{\lambda^{2c}}$ is still inverse-polynomial, $F(\lambda^2)$ is also non-negligible.

Let us now pivot and use the context of symmetric-key encryption to study what security definitions look like in a simpler context without public keys so that we can prove some more substantial facts. We will return to use the above cryptographic assumptions next week.

$C = m \oplus k$
$m = c \oplus k = (m \oplus k) \oplus k = m$ DECRYPTION UNDOES ENCRYPTION IN ONE-TIME PAD

Recall from Reading 4 of last week that one example of a symmetric cipher is the one-time pad, where our keys, messages, and ciphertexts are all λ-bit binary strings, our encryption function is $E(k, m) = k \oplus m$ and our decryption function is $D(k, c) = k \oplus c$, where $\oplus$ is bitwise xor. We want to precisely make the claim that the one-time pad is a secure encryption function, but what does this mean? One phrasing we might try is that for any two messages, m_1 and m_2, no adversary (who does not know k) can distinguish whether a ciphertext, c, corresponds to an encryption of m_1 or an encryption of m_2. But this leaves a few questions unanswered: What is an adversary? Which messages are being used? Who choses the messages and who encrypts them?

1. Who chooses m_1, m_2?
2. How is the key k chosen?
3. What does the adversary actually see?
4. How do we measure success or failure?

Why "security" must be an experiment

A statement like

"No adversary can tell whether a ciphertext encrypts m_1 or m_2"

is meaningless until you specify exactly:

As with cryptographic assumptions, in security definitions we make these kinds of statements precise and functional by formalizing them as attack games. In this case, we have to define a game where the adversary is challenged to distinguish what message a ciphertext corresponds to. In order to have the strongest possible notion of security, we give the adversary as much information and control as we can in the game, so we will let the adversary pick the messages m_1 and m_2, the adversary will then give these messages to the challenger and the challenger will generate a random key, k, choose a random message to encrypt, and return this encryption to the adversary. Finally, the adversary will have to guess whether the ciphertext it was given is an encryption of m_1 or of m_2. The adversary wins this game if its guess is correct, and loses otherwise. A succinct definition of this game is given by the $\mathsf{DistinguishCipher}^{\mathcal{A}}(\lambda)$ program in Figure 2.

Since an adversary that just picks all of its values randomly wins the $\mathsf{DistinguishCipher}^{\mathcal{A}}(\lambda)$

In cryptography, when we claim that a system is "secure," we're not making a vague statement. Instead, we define a precise probabilistic experiment called an attack game which simulates how an adversary might try to break the system. This experiment is implemented as a program (a formal game definition), with the following parts:

1. Inputs chosen randomly (e.g., secret keys, coins, or challenge bits).
2. Inputs chosen by the adversary (e.g., chosen messages).
3. An interaction between the adversary and the system.
4. A win condition (e.g., the adversary correctly guesses which message was encrypted).

The final output of the program is typically a Boolean:

- true if the adversary succeeded (e.g. guessed the correct bit),
- false if they failed.

Because security is defined in terms of Probability
$P_r[\text{adversary wins}]$ = what we want to be negligible

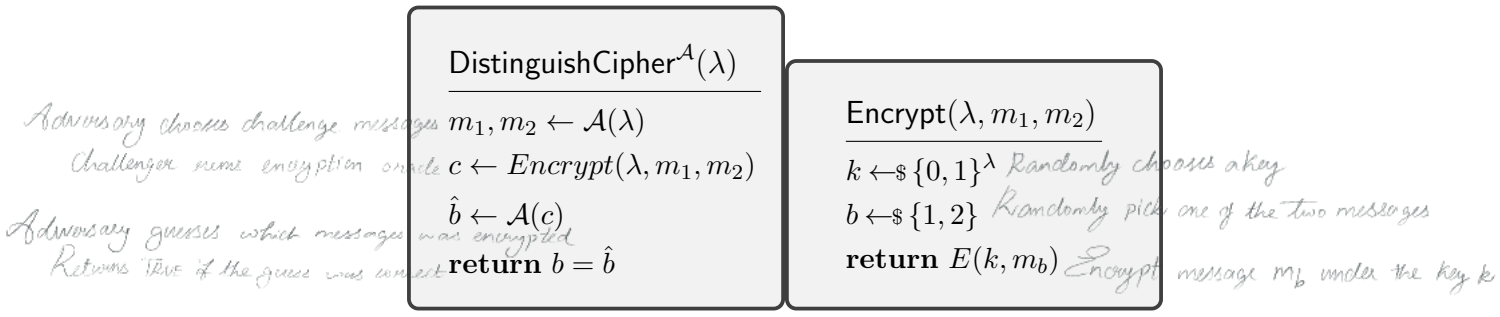

Figure 2: Encryption Distinguishing game ($ means chosen uniformly at random)

game half of the time, we define the advantage an adversary, $\mathcal{A}$, has in this game to be

$$\mathsf{Adv}^{\text{distinguishcipher}}_{\mathcal{A}}(\lambda) := \Pr\left[\mathsf{DistinguishCipher}^{\mathcal{A}}(\lambda) = \text{true}\right] - \frac{1}{2}.$$

Finally, we can define E to be secure if for all adversary programs $\mathcal{A}$, the advantage of $\mathcal{A}$ is negligible.

In order to work with security definitions such as these, we will need to develop a little bit of background in probability so that we can properly reason about advantages. All of our sets (for example, of possible keys or of possible messages) are finite, so we will not need to introduce very much of the general theory. Whenever discussing probabilities there will implicitly be a (finite) set of possible states/outcomes, Ω, and subsets of this state space are known as events. For example, if we are rolling a 6-sided die, we could model this situation with $\Omega = \{1, 2, 3, 4, 5, 6\}$ and the event that you roll an even number is given by the subset $\{2, 4, 6\}$. For our purposes, given a state space, Ω, the function $\Pr[E]$ that takes events as input is defined as follows:

1. $\Pr[E] \geq 0$ for all events $E \subseteq \Omega$. That is, probability is not negative.
2. $\Pr[\Omega] = 1$. That is, the probability that one of our outcomes occurs is 1.
3. $\Pr[E] = \sum_{x \in E} \Pr[\{x\}]$. In other words, every individual state has a probability and the probability of an event is the sum of the probabilities of its elements.

Uniform distribution means every $x \in \Omega$ gets probability $1/|\Omega|$

Thus, if we assume our 6-sided die has each of its outcomes occurring with probability $\frac{1}{6}$ (which satisfies our probability rules), then the probability of rolling an even number is $\frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{3}{6} = \frac{1}{2}$, as we would expect. In general, we call the assignments of probabilities to individual states a *probability distribution* (assuming all the values are non-negative and add up to 1). For example, if we have a set of N outcomes, Ω, then the *uniform* probability distribution on Ω is the assignment of $\Pr[x] = \frac{1}{N}$ for every $x \in \Omega$; when working with uniform distributions, computing probabilities is always the same as counting the number of states in an event (and then dividing by N).

Exercise 3. *Suppose there is a room with* 30 *people, and assume that the probabilities of all assignments of one of the* 365 *possible birthdays to each of the* 30 *people are equal (this is of course not true, but assuming a uniform distribution will make things simpler). Compute the probability that no two people share a birthday, and then compute the probability that at least two people share a birthday.*
(If instead of people and birthdays we think of inputs and hash values, this "birthday problem" shows that finding hash collisions is easier than finding hash pre-images).

If you would like an additional resource on probability that moves through some of the above topics a bit more slowly and with far more examples, see the following optional reading:

Reading 1 (Optional)**.** *If you find yourself wanting more probability practice, read about the axioms of probability and conditional probability in:*
Ross - First Course in Probability (5th ed.) - pages 25-39, 67-89.

Exercise 4. *Explain why the following is true using the definition of probability:*

$$Pr\left[\bigcup_{i=1}^{n} E_i\right] \leq \sum_{i=1}^{n} Pr[E_i].$$

When are these two values equal? (Hint: first try the case where $n = 2$).

In case this notation is not familiar to you, $\bigcup_{i=1}^{n} E_i = E_1 \cup E_2 \cup \cdots \cup E_n$ *refers to the union of all of the sets* E_1 *through* E_n*, and* $\sum_{i=1}^{n} Pr[E_i] = Pr[E_1] + Pr[E_2] + \cdots + Pr[E_n]$ *refers to the sum of all of the probabilities* $Pr[E_1]$ *through* $Pr[E_n]$*. You can think of both of these as essentially for loops.*

An important notion in probability, that we have already used, is when two events are *independent.* Intuitively this means that two events are uncorrelated. In order to capture this notion precisely, we first introduce the notion of *conditional probability*:

$$\Pr[E_1 \mid E_2] := \frac{\Pr[E_1 \cap E_2]}{\Pr[E_2]}. \qquad \Pr[E_2] > 0$$

This formula is read "The probability of E_1 occurs *given* that E_2 occurs is equal to the probability that E_1 and E_2 both occur divided by the probability that E_2 occurs." This makes sense in our definition of probability because the information that E_2 occurs essentially is the same as replacing our original set of possible outcomes, Ω, with E_2 instead, and re-interpreting the event E_1 as the set of states in E_1 that are also in our new set of possible outcomes, E_2, throwing away the rest of the states.

An important and immediate result is that $\Pr[E_1 \cap E_2] = \Pr[E_1 \mid E_2] \cdot \Pr[E_2]$. This observation allows us to compute probabilities by breaking a state space up into cases. Suppose that the state space has a partition into two pieces: $\Omega = A \cup B$ and there is no intersection between these pieces ($A \cap B = \emptyset$). Then we can compute the probability of any event, E, by splitting into the A and B cases as follows,

$$\begin{aligned}\Pr[E] &= \Pr[E \cap A] + \Pr[E \cap B] \\ &= \Pr[E \mid A] \cdot \Pr[A] + \Pr[E \mid B] \cdot \Pr[B].\end{aligned}$$

And more generally, it similarly follows that for any partition $\Omega = \cup_{i=1}^{n} A_i$, we have

$$\Pr[E] = \sum_{i=1}^{n} \Pr[E \cap A_i] = \sum_{i=1}^{n} \Pr[E \mid A_i] \cdot \Pr[A_i].$$

One way to think about this equation is that it says that you can always compute a probability, $\Pr[E]$, as a weighted average of the conditional probabilities that E occurs over some partition of the state space, where the weights used are the probabilities of the partition elements.

Now that we have conditional probabilities, we can define what it means for two events to be independent: We say that E_1 and E_2 are *independent events* if $\Pr[E_1 \mid E_2] = \Pr[E_1]$. (and equivalently, $\Pr[E_2 \mid E_1] = \Pr[E_2]$)

Knowing that E_2 happens gives no information about whether E_1 happens

Exercise 5. *Using the definitions above, show that two events, E_1 and E_2, are independent if and only if $P(E_1 \cap E_2) = P(E_1) \cdot P(E_2)$. This is the usual textbook definition of independent events, but this sometimes hides the intuition behind what this has to do with independence.*

Exercise 6. *We now have all of the tools we need to prove that the one-time pad is secure!*

(a) Let us begin with the example of the single-bit encryption case (where $\lambda = 1$). Show that for all adversary programs, $\mathcal{A}$, it is always true that $\mathsf{Adv}_{\mathcal{A}}^{\text{distinguishcipher}}(\lambda) = 0$.

(Hint: let $\Omega = \mathcal{K} \times \{1,2\} \times R$ be the set of triples with one element from $\mathcal{K} = \{0,1\}$, another bit element from $\{1,2\}$, and an element of some set, R, which represents the randomness used by our adversary (these are sources of randomness). Show that if m_1 and m_2 are any pair of messages, then $Pr[\mathcal{A}(c) = b] = \frac{1}{2}$, where $c = E(k, m_b) = k \oplus m_b$ and $\mathcal{A}$ is any program, by breaking into the four cases $(k,b) = (0,1), (0,2), (1,1), (1,2)$ and then using the partition of Ω into sets where k and b are fixed to compute $Pr[\mathcal{A}(c) = b]$. Note that you will also have to treat separately the case that $m_1 = m_2$ and the case that $m_1 \neq m_2$).

(Extra hint if you get stuck near the end: Remember $Pr[\mathcal{A}(c) = 1] + Pr[\mathcal{A}(c) = 2] = 1$ for any fixed value of c since the probability that $\mathcal{A}$ returns an output must be 1).

(b) *Using the same general argument as in the previous part, show that* $\mathsf{Adv}^{\text{distinguishcipher}}_{\mathcal{A}}(\lambda) = 0$ *for all* λ *and for all adversaries,* $\mathcal{A}$.

Since k⊕m is independent of m, we learn zero about m from c, even with infinite computing power

The general idea here is that since $k \oplus m$ is independent of m when k is uniformly random, no program can use $k \oplus m$ as input to gain any information at all. This is actually much stronger than our requirement for security; having $\mathsf{Adv}^{\text{distinguishcipher}}_{\mathcal{A}}(\lambda) = 0$ instead of just having it be negligible, is called *perfect secrecy* which was a notion developed by Shannon as he was inventing information theory. Unfortunately, Shannon also proved a theorem showing that if we require perfect secrecy, then our key sizes must always be equal to our message sizes (that is, $|\mathcal{K}| = |\mathcal{M}|$). This is in stark contrast to the world we are used to, where fixed key sizes can be used to encrypt very large messages. This is often accomplished by stretching a key by using it as the seed to a pseudo-random generator and then using the output of this pseudo-random generator to do encryption. Our final goal this week is to put together much of what we have learned so far to prove that this is secure! This accomplishment demonstrates the power of using negligible advantages instead of advantages equal to 0 in security definitions.

'STRECH' a small amount of randomness into longer output that is computationally indistinguishable from truely random

A *pseudo-random generator* (PRG), G, is an efficient deterministic algorithm that takes as input an input from $\mathcal{S} = \{0,1\}^\ell$ and outputs an element from $\mathcal{R} = \{0,1\}^L$ where $L > \ell$ (for our purposes this week, we can take $L = \ell + C$ for some large constant C). We often call the elements of $\mathcal{S}$ seeds.

Exercise 7. *Using the security definitions we have seen so far as a model, define an attack game that calls on an adversary to distinguish between an output of* G *and a truly random element. Define the advantage an adversary has in this game, and define what it means for a PRG,* G*, to be secure in terms of advantages.*

We define the *stream cipher* associated to to a PRG, G, to be the functions (E_G, D_G) given by $E_G(k,m) = G(k) \oplus m$ and $D_G(k,c) = G(k) \oplus c$, which is like the one-time pad except that we first stretch our key using G

If G is a secure PRG, then this encryption is computationally secure: no efficient adversary can tell whether it's encrypting m1 or m2.

We can simulate the one-time pad using a PRG:
Instead of using a truly random key of length L, Use a short seed $s \in \{0,1\}^\ell$,
Compute $G(s) \in \{0,1\}^L$,
Encrypt message $m \in \{0,1\}^L$ via: $c = m \oplus G(s)$

i.e., To securely send a 1000 bit message, our key must be 1000 bits long, and it can be used only once. But this is highly impractical for real-world sending GBs of data over the internet is not applicable. So PERFECT SECRECY IS NOT USABLE IN PRACTICE

Exercise 8. *Using the definition of a secure cipher given in Figure 2, and your definition of a secure PRG from the previous exercise, use proof by reduction to show that if G is a secure PRG, then E_G is a secure encryption function. You may use the fact, that we will prove next week, that if $F(\lambda)$ is not negligible, then $\frac{F(\lambda)}{2}$ is also not negligible.*

Hint: Given a program $\mathcal{A}$ that has a non-negligible advantage against the game in Figure 2 for E_G, construct an adversary program $A'(L, r)$ that plays your game from the previous exercise by using r as a one-time pad key to encrypt a random message given by $A(L)$. To argue that A' has a non-negligible advantage, use the fact that

$$Pr[\mathcal{A}' \text{ wins}] = \frac{1}{2} \cdot Pr[\mathcal{A}' \text{ wins} \mid r \text{ is random}] + \frac{1}{2} \cdot Pr[\mathcal{A}' \text{ wins} \mid r = G(s)] .$$

Reading 2. *Once you have attempted Exercises 7 and 8, you may check your work by reading the following text that uses different (but equivalent) attack games and notation to show the same result.*

Boneh and Shoup - Sections 3.1 and 3.2.