Example scenarios:
   - Alice ----(public network)----> Bob
   - Alice -------->(disk)----------> Alice

Overall goals
   * CONFIDENTIALITY: Keep other people from reading Alice's messages/data
   * INTEGRITY: Keep other people from tampering with Alice's messages/data
   * AUTHENTICITY: Ensure that data purportedly from Alice really is from Alice

At the heart of cryptography is the notion of randomness. To understand the kind of randomness that cryptography tries to achieve, it is useful to first view it within a conceptual, idealized model of "random functions". Recall that a function F is just any way to map from some set X to some (possibly different, possibly the same) set Y. (In case "function" here sounds too much like continuous, differential functions like a line or polygon, just think of a "function" here as a hash table mapping X to Y.)

Consider the set of ALL possible functions mapping X to Y. For even reasonably large X & Y, there are astronomically many functions between them.

Now suppose that out of all possible functions between X and Y, you and I choose one uniformly at random (i.e., each function has equal probability of getting chosen), and that no one but us knows which out of those astronomically many functions we chose.

What kinds of properties does this function chosen uniformly at random have?

- For any given x \in X, the probability that F(x) = y is 1/|Y|. That is, each output is chosen uniformly at random (of course, if Y consisted of a single value, then someone can easily guess the output without knowing F!).
- The slightest change to a given input can result in a completely different output. Put another way, if x' is the same as x, but with as little as one single bit changed, then the probability that any two bits of F(x) and F(x') match is 1/2. This is because the function was chosen at random: again, if you think of a function as a hash table, then if we chose a hash table at random, there may be no connection whatsoever between F(x) and F(x'), no matter how close x and x' may be.

An important variant of this is a random "permutation". In this setting, F maps from an input set X into the same output set X, but in a random, one-to-one fashion.

Another important property that we will want from a random function is that it is one-way: given F(x), it is impossible to determine x without resorting to a "brute force attack":

```
- Given y
- do
-       choose an x
- while ( F(x) != y );
- return x;
```

If the set of possible inputs is X, then this will take as many as |X| steps. So long as X is very large, then a brute force attack will not be feasible. On the other hand, if the set of possible inputs happens to be small (e.g., if the attacker knew that the only possible inputs were "attack in the morning" or "attack in the evening"), then a brute force attack would be very simple. Our goal will be to provide confidentiality even if Alice wishes to send one of only a few messages.

Why is such a function going to be useful? Let's consider our scenario where Alice was writing a file to disk without wanting anyone else to read it. Instead of writing x to disk, she could write F(x) to disk, instead. This will appear to be a random value; one can only recover it with a brute force attack.

But that's the problem: not even Alice can recover the file without a brute force attack! Shredding a document is a one-way function---and it goes a long way towards achieving confidentiality---but that's not a very useful way to store one's documents.

In other words, one-way functions alone aren't going to be enough to meet all of our goals. Often, for a given F(x), we will want some method by which we can *efficiently* recover x. But of course, if *anyone* could invert F, then anyone could recover Alice's stored data F(x).

This brings us to the notion of a "one-way trapdoor function". The idea is that such a function would take two values: an input x (as before), and a key k. Suppose Eve sees an output value F(k,x):

- If she does not know k, then she cannot efficiently recover x. That is, F(k,.) appears to be a one-way function.
- However, if she does know k, then she can efficiently recover x.

Some good news: if we had a one-way trapdoor function F(key, message), then we can solve our confidentiality problem! The protocol would work as follows:

- Alice and Bob agree on a random, large key k, and both agree to keep it secret.
- Alice wants to send message m; she computes F(k,m) and sends it over the public network to Bob.
- Because Bob knows k, he can efficiently recover m from F(k,m).

Eve obtains F(k,m), but since she doesn't know k, she cannot efficiently recover m (she can at best perform a brute-force attack).

These are just some high-level concepts: we can't possibly store all possible random functions between all binary strings, and in fact, we don't even know if random one-way trapdoor functions exist! Instead, we have functions that look and feel like these kinds of functions.

These are called pseudorandom functions. They are made up of purely deterministic steps, but they approximate many of the same properties of random functions (e.g., changing one bit in the input changes each bit in the output with probability approximately 1/2).

Creating really good pseudorandom, one-way trapdoor functions is extremely difficult, and involves some very deep results in mathematics and cryptography. Our goal in this class is not to dig into these in depth (for that, you are highly encouraged to take courses in cryptography). Instead, we are going to be investigating how to use these.

Because, you see, cryptography is not a fool-proof solution to all security problems.

- It can be implemented incorrectly
- It can be USED incorrectly

An important distinction here is between MECHANISM and PROTOCOL. A protocol is, broadly speaking, an algorithm involving more than one principal (person, program, device, etc.) communicating with one another. (A protocol that consists of only one party is an algorithm.) Protocols make use of mechanisms like the random functions we've been talking about thus far. The protocol might take a perfectly good mechanism and make it worthless.

Consider our example once more with Alice and Bob communicating over the private network: what if she shared the same key k with everyone she communicated with?---any message she sent would be visible to everyone who had this key. What if she and Bob chose a very small key (k = 3)?---a brute force attack would only take a few iterations. Note that these problems arose

because Alice misused the cryptography available to her, not because it was broken.

But, of course, cryptographic mechanisms are occasionally broken. Some that were once in wide use (e.g., DES, SHA-1, MD5) were eventually found to be insufficiently secure in the face of new types of attacks.

To briefly summarize and motivate the remainder of this document, the best defenses against compromised crypto are to:

(1) Understand the crypto you use.
- In what settings is it secure?
- What kinds of keys are secure?
- How long are data protected with a given crypto scheme/key size?

(2) Stay up to date with new mechanisms and attacks
- Is <favorite variant of encryption> still secure?
- Is <your application's key size> still sufficient?

In other words, this document should at best be your FIRST step towards learning about and using cryptography. To remain safe, you must remain informed.

==========

SYMMETRIC KEY CRYPTO

BLACK BOX = BLOCK CIPHER:

A pseudorandom one-way trapdoor function, which operates over fixed-sized "blocks"

Key k
E(k,m) = c: "encryption", outputs "ciphertext"
D(k,c) = m: "decryption", outputs the original "plaintext" message

A defining property of a block cipher that the size of the input messages is equal to the size of the output messages. This is called the "block size".

(Because the input size is equal to the output size, and because the function is invertible, this is an example of a "pseudorandom permutation".)

The crucial CORRECTNESS property is that for any k and m, we have D(k, E(k,m)) = m.

The crucial SECURITY property is that, once the key is fixed, the resulting function F(k,.) is indistinguishable from a function chosen uniformly at random from all possible functions between block-sized binary strings. This notion of randomness is captured by the notions of:

CONFUSION: Each bit of the ciphertext should depend on multiple parts of the key.

DIFFUSION: Flipping a bit in the message should result in multiple bits changing in the ciphertext (and vice versa). Ideally, *every* bit in the ciphertext would change with 50% probability.

There are many refinements of confusion and diffusion that are very important in evaluating the quality and properties of a given block

cipher, but for the sake of using them, you can think of it as "Slight (even one-bit) changes in the key or the message result in significant changes in the ciphertext."

Tying this back to our discussion above, Alice and Bob's protocol becomes:

- Alice and Bob agree on a key k
- Alice wants to send message m; she computes c = E(k,m) and sends c
- Bob obtains c and computes D(k,c) = m to recover the original message.

WHAT IS KEPT SECRET?

The ONLY thing about a block cipher that is kept secret is the shared key. The encryption and decryption algorithms ought to be known publicly, as per Kerkhoff's principle (more simply stated as "security through obscurity is bad").

SOME EXAMPLE BLOCK CIPHERS

	block size	key size
3DES	64	168
AES	128	128, 192, or 256

3DES is slower and less secure than AES, but it is still used in practice. There is no compelling reason to use anything but AES.

USING BLOCK CIPHERS:

Fresh out of the box, block ciphers have two shortcomings that we will talk about overcoming.

--The first problem with block ciphers--
First, suppose Alice repeatedly sends one of only a few messages to Bob. Each time she sends some message m, Eve will see the same ciphertext E(k,m). This may be enough for Eve to infer something about Alice and Bob's communication: suppose that every time she sees ciphertext c, the next day the price of oil goes up, and every time she sees ciphertext c', the price of oil goes down---even without recovering the messages themselves, Eve was able to infer some very important information!

Note that this problem wouldn't have arisen had every message been random. The trick is then to make messages random by including a NONCE into the message. (Nonces are sometimes referred to as "initialization vectors".) We will see precisely how to incorporate initialization vectors into block ciphers, but for the time being, let's just think of both encryption and decryption as taking three inputs: a key, a message, and an IV.

There are fundamentally two different ways to choose initialization vectors:

(1) Each could be RANDOM. For example, with each message, Alice sends, she could choose a random IV r, compute

$$c = E(k, m \text{ xor } r)$$

and send both r and c to Bob.

Bob would then recover m by computing

m = r xor D(k,c).

(2) Each could be UNIQUE. For example, when sending the i'th message, Alice could compute c = E(k,m xor i) and send c to Bob.

So long as Bob knows it's the i'th message, he can recover m by computing i xor D(k,c).

... Well, almost ...

As we will see later, simply using incrementing counters can be harmful when combined with so-called "modes" of encryption. We will evaluate this more later, but for now, here is the rule: if you are going to use a "unique" IV, then use

IV' = E(k, IV)

for both encrypting and decrypting. This ultimately results in a random IV' (recall that even a slight change in IV, like incrementing it, will result in a significantly different IV').

The main benefit of unique initialization vectors is that Alice doesn't have to use as much bandwidth as random ones. But it requires in-order delivery (the i'th message that Alice sends must be the i'th message that Bob receives).

Protocols that ensure in-order delivery, such as SSL/TLS, use unique nonces, while protocols that do not ensure in-order delivery, such as IPsec, use random nonces.

--The second problem with block ciphers--
The other problem with block ciphers is that they only operate over small blocks of data. Using AES's block cipher alone, we would only be able to encrypt messages up to 128 bits long!

There are multiple "modes" of encryption using block ciphers. These operate by breaking an arbitrarily sized input into segments of length equal to the block size (e.g., 128 bits for AES), and iteratively applying the block cipher.

The modes have different guarantees with respect to security and performance, and as a result, crypto libraries export many different modes. Following our principle of knowing what your libraries give you, it is important to understand their trade-offs (and which modes to avoid altogether!).

BLOCK CIPHER MODES

--Electronic code book mode--
One of the most natural ways to iteratively apply a block cipher is to simply encrypt/decrypt each block-sized segment. Let m[i] denote the i'th segment of the plaintext (e.g., for AES, m[1] would be the first 128 bits, m[2] would be the next 128 bits, etc.). Let c[i] denote the i'th segment of the ciphertext.

Electronic Code Book (ECB):

Encryption:
Input: plaintext m, key k
c[i] = E(k, m[i])

```
   Decryption:
      Input: ciphertext c, key k
      m[i] = E(k, c[i])
```

Can you spot the problem here?

If two separate segments are equal, m[i] = m[j], then Eve can detect this by noting that c[i] = c[j]. This may seem innocuous, but it leaks information, which in some settings Eve may be able to exploit to infer something about Alice and Bob's conversation.

As a result, you are highly discouraged from ever using ECB.

This was the same issue that led us to initialization vectors; the rest of the modes avoid this issue by incorporating IVs in one way or another.

--Cipher-block chaining mode--
Let's keep the basic notion of using a different block cipher for each block of data. The idea is to introduce randomness at each step. Recall that each iteration of the block cipher either needs a RANDOM IV or a UNIQUE IV. This mode generates a random one at each step.

How do we generate a random IV multiple times? Well, fortunately, there is already (pseudo)randomness built into the system: the output of every call to E() is pseudorandom. So in CBC, we simply use the output of one block's E() as the IV into the next block's E():

```
   Cipher-Block Chaining (CBC)

   Encryption:
      Input: plaintext m, key k, initialization vector IV
      c[0] = IV
      c[i] = E(k, m[i] xor c[i-1]), for i >= 1

   Decryption:
      Input: ciphertext c, key k, initialization vector IV
      m[i] = D(k, c[i]) xor c[i-1]
```

Let's talk security: the issue that we faced with ECB---wherein m[i] = m[j] resulted in c[i] = c[j]---does not apply to CBC. This is because the inputs to E() at step j are, with CBC, extremely likely to be different than the inputs to E() at step i.

As for performance, note that encryption in CBC mode is not parallelizable: computing ciphertext block c[i] requires knowing the previous ciphertext block c[i-1]. On the other hand, decryption in CBC mode can be parallelized: recovering m[i] does not require knowing m[i-1]. Sure, it requires knowing c[i-1], but all of the ciphertext blocks are given as input.

There are other modes that generate random IVs for each application of the block cipher. The differences between them are relatively minor (and easier to identify when viewed diagrammatically: I recommend looking up the block diagrams online). For your edification:

```
      - Propagating cipher-block chaining (PCBC)
         Encryption:
            c[0] = IV
            c[i] = E(k, m[i] xor m[i-1] xor c[i-1])

         Decryption:
            m[0] = IV
            m[i] = D(k, c[i]) xor m[i-1] xor c[i-1]
```

```
      - Output feedback (OFB)
         Encryption:
            p[0] = c[0] = IV
            p[i] = E(k, p[i-1])
            c[i] = p[i] xor m[i]

         Decryption:
            p[0] = c[0] = IV
            p[i] = E(k, p[i-1])
            m[i] = p[i] xor c[i]

      - Cipher feedback (CFB)
         Encryption:
            c[0] = IV
            c[i] = E(k, c[i-1]) xor m[i]

         Decryption:
            m[0] = IV
            m[i] = E(k, c[i-1]) xor c[i]
```

An interesting feature of OFB and CFB is that neither make use of the block cipher's decryption function D(). What distinguishes OFB from the others is that encryption and decryption are identical.

Some permit faster implementation in hardware, usually at the cost of not allowing parallelizable decryption (from the descriptions above, a useful exercise is to identify which permit parallelizable encryption or decryption). Among all of these, CBC is by far the most commonly used.

--Counter mode--
Let us now consider a different mode wherein the IVs used at each iteration are not random but UNIQUE. We will ensure uniqueness by incrementing the IV at each iteration, rather than using the output from the previous iteration. This means that we don't need to wait for the computation of c[i-1] to compute c[i].

```
   Counter mode (CTR)

   Encryption:
      Input: plaintext m, key k, initialization vector IV
      c[0] = IV
      c[i] = E(k, IV+i) xor m[i]

   Decryption:
      Input: ciphertext c, key k, initialization vector IV (= c[0])
      m[i] = E(k, IV+i) xor c[i]
```

CTR mode allows parallelizable encryption and decryption.

Moreover, note that it does so by using *only* E()!

To understand how it derives its security, let us take a moment to discuss "one-time pads"

ONE-TIME PADS

A one-time pad is a very simple mode of encryption and decryption that has some very strong security properties. It also is a great example of how misusing a crypto system can completely violate all of its security properties.

```
   One-Time Pad (OTP)
```

```
   Encryption:
      Input: plaintext m, key k OF SIZE AT LEAST |m|
      c = m xor k

   Decryption:
      Input: plaintext m, key k
      m = c xor k
```

OTP is not a mode, nor does it make use of block ciphers (or even pseudorandom functions). It is far simpler than that, but also far more limited in its utility.

First, the bad news: a one-time pad really lives up to the name, in that a given key must only be used ONCE. To see why, suppose Alice reused the same key k to encrypt two different messages m and m', yielding ciphertexts c and c'. Eve could infer information by computing c xor c' = (m xor k) xor (m' xor k) = m xor m'. It turns out that xor'ing two human-language messages together can leak information. This was the basis of the Venona project, a US-led initiative during the Cold War; Soviets were incorrectly using one-time pad keys twice, which allowed the Venona project to recover a significant portion of the plaintext messages.

Now the good news: if the key is used only once, OTPs offer perfect secrecy. That is to say, OTPs are not even susceptible to brute-force attacks! To see this, let's consider a really simple one-bit message. Since OTP KEYS MUST BE AS LONG AS THE PLAINTEXT MESSAGE, the key must also be a single bit. Eve sees one of two values, either a zero or a one. Let's say that she sees a zero, and let's try to brute force all of the keys.

- c = 0 && k = 0 => the message was a zero
- c = 0 && k = 1 => the message was a one

If the key is chosen at random, then k takes on either 0 or 1, both with probability 1/2. Thus the probability is also 1/2 that the message was a 0 or a 1. That is to say, so long as:

(a) the key is random,
(b) the same length as the message, and
(c) not used more than once,

then Eve can learn absolutely nothing about the message: ANY message of length |m| could have been the original message.

So why not just use OTPs? The problem is that the key size must be precisely as long as the message: establishing keys long enough to, say, transfer large files would be impractical. This is an *inherent* limitation of perfect secrecy.

Instead of *perfect secrecy*, the block cipher-based approaches we are talking about provide what is called "computational secrecy": instead of total secrecy against computationally unbounded attackers, it provides secrecy with high probability against bounded attackers.

In other words, we are making a trade-off: we are accepting a weaker form of security, but in return are opening ourselves up to much more efficient schemes.

BACK TO CTR MODE

Now take another look at CTR mode; each iteration i is very similar to OTP, where the message is m[i] and the key is E(k, IV + i). This is the basis of CTR mode's security, but as you can see, it depends heavily on the properties of the block cipher, E().

Once again, I highly encourage you to take cryptography courses to understand more about the properties of block ciphers, how they are constructed, why they take on the key and message sizes they do, and also to understand other modes of symmetric key encryption.

CTR AND COUNTER-BASED NONCES

When we introduced the notion of unique IVs, we mentioned that we should actually use the encryption of the counter as the IV. Now we can see why.

Suppose that we were sending a message M that was two blocks large. Applying CTR mode would result in a ciphertext of:

```
C[0] = E(k, IV + 0) xor M[0]
C[1] = E(k, IV + 1) xor M[1]
```

Now suppose that we just used an incrementing IV, so the next time we encrypted something, we used IV' = IV+1 as the nonce. Now suppose that we send a single-block-sized message M', and that it just so happens that M'[0] = M[1]. Then we would get:

```
C'[0] = E(k, IV' + 0) xor M'[0]
      = E(k, (IV+1) + 0) xor M'[0]
      = E(k, IV + 1) xor M[1]
      = C[1]
```

The problem was that we were ultimately conflating our IV with the internal state of the CTR mode's counter. By instead using IV' = E(k, IV) at each step, we can be sure that there is with very high probability no connection between the IV counter and CTR's counter.

SYMMETRIC KEY CRYPTO PROPERTIES

The techniques covered thus far provide a fascinating property: they allow two principals, Alice and Bob, to communicate over a public network without letting an eavesdropper Eve discern anything about their true, plaintext messages (that is, unless Alice and Bob use ECB or reuse OTP keys).

Moreover, these cryptographic mechanisms are extremely fast in practice. This is generally true of symmetric key crypto. Thus, when encrypting large amounts of data, symmetric key crypto is the way to go.

On the other hand, there are two main issues with these symmetric key crypto schemes. First, they *only* protect against eavesdropping attacks: there is nothing to stop Eve from modifying the messages in transit, and yet these schemes provide no way for Bob to detect whether or not this happened.

Second, because they require the same key for encrypting and decrypting, they require every pair of communicating parties to establish a pairwise key. Without already having a shared key, how can Alice and Bob establish a key without Eve figuring out what it is?

We will address both of these issues in turn. To do so, we will need to introduce new black boxes beyond the block ciphers we have covered thus far.

==========

MESSAGE INTEGRITY

Let's continue with the scenario in which Alice is communicating with Bob over a public network. The next problem we will tackle involves an attacker, Tammy, who intercepts the messages that Alice sends to Bob and tampers with them. Without locking down the entire network (which is impossible in practice), our goal is to allow Bob to DETECT whether anyone tampered with Alice's original transmitted message.

This goal is completely orthogonal to eavesdropping, so for the time being, let us assume that Alice is sending her message in the clear (i.e., not encrypted at all), and that we are not worried about an eavesdropping attack. Keeping the problems of confidentiality and integrity as separate problems allows us to apply them to a broader set of applications: Alice might be sending Bob an advertisement; she doesn't care if everyone can see it (she'd love for them to!), she just wants to make sure that Tammy can't slip in her own ad without Bob realizing it.

(We will see at the end of this section how to combine authenticity and encryption.)

MESSAGE AUTHENTICATION CODES

The overall solution to this problem, known as MACs (message authentication codes), consists of two algorithms:

(1) Alice runs a SIGNING function S

- Inputs: key k, message m
- Outputs: tag t

She then sends m and t to Bob.

(2) Bob runs a VERIFICATION function V

- Inputs: key k, message m, tag t
- Outputs: "yes" if Alice sent (m,t),
  "no" otherwise

A cryptographic mechanism that provides both of these functions is referred to as a MAC. There are many different ways to implement a MAC, and many different kinds are used in practice. We will cover a few, but before we do that, let us reason about what it means for a MAC to be SECURE.

SECURE MACs

Recall that, when defining the security of a system, we have to define two things: the attacker's powers (what an attacker can and cannot do), and the attacker's goals. If the attacker can achieve his or her goals using the assumed powers, then we say that the system is insecure.

The best crypto systems allow attackers an incredible amount of power, while giving the attacker a very meager set of goals. If the system can be proven secure under those assumptions, then it is likely to be applicable in a broad set of domains.

We say that a MAC is secure if:

Attacker's powers: chosen plaintext attack
- The attacker can issue messages m_1, m_2, ... and get in return the corresponding tags t_1, t_2, ...

Attacker's goal: EXISTENTIAL FORGERY

- After seeing message/tag pairs (m_1, t_1), (m_2, t_2), ...
  the goal of the attacker is to produce a message/tag
  pair (m', t') such that

    (a) (m', t') is VALID, that is, V(k, m', t') = "yes"
    (b) (m', t') != (m_i, t_i) for any i (i.e., it is new)

We have allowed the attacker a seemingly easy goal: we are not asking the attacker to be able to forge a tag for *any* given message; nor are we even asking for the m' he creates to even make any sense! It could be complete gibberish, but so long as it is valid, then Bob will think that Alice intentionally spoke gibberish to him.

We will be discussing MACs that are secure by this definition, but before we do that, there is an important property to mention:

THIS PROBLEM IS IMPOSSIBLE TO SOLVE WITHOUT ALICE AND BOB SHARING A SECRET KEY.

Consider the alternative: let's say that we used a simple checksum algorithm like CRC to compute our tags. That is, suppose that tag t = CRC(m). Because CRC is a publicly available function that requires no secret inputs, anyone can compute CRC on any message. As a result, it is trivial for Tammy to produce an existential forgery: she can use any message m and compute t = CRC(m).

So in all of these schemes, we are going to continue using shared secret keys, just as we did with symmetric key encryption. In fact, the similarities run much deeper than that, thanks to the following result:

ANY PSEUDORANDOM FUNCTION (like our block ciphers) YIELDS A SECURE MAC, AS WELL

"SMALL" MACS (no one really calls it this)

To see this result, let's start with a toy example: where we only wish to ensure the integrity of messages that are only as large as a block size (e.g., 128 bits for AES). We can construct a MAC as follows:

* S(k,m) = E(k,m) -- this is our tag t

* V(k,m,t) = "yes" if t = E(k,m), "no" otherwise

Why is this a secure MAC? Let's reason through it:

- E(k,m) looks like a random function to the attacker
- So when the attacker sees E(k,m_1), E(k,m_2), ...
- These look like random values
- How is the attacker supposed to guess a random value?
- If, like in AES, the tags are 128 bits long, then the attacker would only have a 1 in 2^128 chance of guessing.

The problem is exactly the same as the problem with block ciphers for encryption: used alone, they can only be used for small messages. So how can we generate secure MACs for bigger messages?

"BIG" MACS (no one really calls it this)

Because the problem is so similar, it should come as no surprise that some of the solutions look very similar, too. In fact, we basically just apply CBC mode encryption to get a secure MAC for larger messages! There are, however, two very important differences. Let's look at the construction:

```
   Encrypted CBC (ECBC)

   Signing:
      Input: message m, key k, key k'
      - Break m up into block-size-long segments: m[1], ..., m[n]
      - Apply CBC using key k and no initialization vector
      - BUT do NOT output the intermediate
          values (c[1], ..., c[n-1]).
      - Take only the final ciphertext output, c[n]
      - The tag is E(k', c[n])

   Verifying
      Input: message m, key k, key k', tag t
      - Simply run the Signing algorithm with inputs (m, k, k') and
        return "yes" if it outputs t, return "no" otherwise.
```

The two differences with the original CBC mode are:

(1) In CBC, the size of the output was the same as the size of the input, but ECBC outputs only a single block. This is because we simply do not need to return anything more: our attacker Tammy still has only a 1 in 2^128 chance to produce an existential forgery (for AES, where the block size is 128). Because we send the tag along with the message, it is best to keep it as small as possible while still keeping Tammy's chances of attacking astronomically low. The output of CBC had to be much larger because Bob was recovering the original message from CBC's output. With a MAC, we are not trying to recover the message, just to make it very unlikely that anyone else can undetectably tamper with it.

(2) We used two keys! Consider the alternative: what if we had simply output the final ciphertext block, c[n]? This makes it possible for an attacker who can launch a chosen plaintext attack to produce an existential forgery.

* Attacker asks for the tag for message m = m[1]...m[n]
* Gets corresponding tag t = c[n]
* The attacker then generates a single-block message m'
* And then asks for the tag for message c[n] xor m'
* He gets the corresponding tag t'
* t' is also a valid tag for m concatenated with m'

(It would be very good practice for you to work out this example on your own.)

These differences are important for understanding what is going on behind the scenes, and in understanding why you need two keys when using ECBC. But ECBC and CBC also share many similarities: in particular, they must be computed sequentially.

There are also MACs that are rather similar to CTR modes, aptly named Parallel MAC (PMAC), which permit parallel computation. But we will not cover that here.

HASH MACS

Another way to construct MACs does not use block ciphers at all, but rather a different primitive altogether. This brings us to our second black box, the hash function.

BLACK BOX = HASH FUNCTION

A psuedorandom one-way function that does not take a key.

H(m) = h

* The input m is called the "pre-image" and comes from an arbitrarily large set of inputs.

* The output is called the "digest" or simply the "hash", which is typically much smaller, such as 256 bits.

While there are many hash functions, there are two important properties that make a hash function cryptographically secure:

(1) Pre-image resistant

Given H(m) it is hard to find m

(2) Collision resistant

Given H(m), it is hard to find m' such that
* m' != m and yet
* H(m) = H(m')

In fact, cryptographically secure hash functions provide an even stronger property than collision resistance: for two messages m != m' and for a given bit index i, the probability that H(m) and H(m') have the same value for bit i is 1/2.

The nice thing about hash functions is that they are typically very efficient to compute.

In practice, hash functions themselves are also computed iteratively, similar to the modes in block ciphers, but we will not cover that here.

EXAMPLE HASH FUNCTIONS

MD5 outputs 128 bits, and is still commonly used, despite collision attacks discovered in 2004 (and more attacks since).

There is a large family of hash functions known as SHA (SHA-1, SHA-256, SHA-512, etc.). SHA-1 outputs 160 bits, and it is also commonly used, although there exist theoretical attacks on it. SHA-256, which outputs 256 bits, is the recommended hash algorithm for most settings.

HMACS

Now let us see how to apply this secure hash function black box to obtain a secure MAC.

From now on, let "||" denote concatenation.

Hash-MAC (HMAC)

Signing:
Input: key k, message m
- opad = 0x5c5c5c...
- ipad = 0x363636...
- S(k, m) = H((k xor opad) || H((k xor ipad) || m))
- In fact, you can just return the last 128 bits as the tag t

Verification consists of simply rerunning the Signing algorithm and verifying that it is equal to the tag provided.

This can be viewed as applying a MAC (the outer H()) on a message digest (the inner H()).

WHAT IS USED IN PRACTICE?

These two MACs are both used extensively in practice. HMAC is the predominant MAC used online, while ECBC is commonly used by banking institutions.

==========

AUTHENTICATED ENCRYPTION

We have thus far solved two problems SEPARATELY:

(a) Protection against eavesdroppers
(b) Detection of tampering

Now let us turn to how we can solve both of these together. Let us return once again to our scenario where Alice and Bob are communicating over a public network. They wish to remain resilient to an eavesdropper, and thus must encrypt their data before sending it over the network. They also wish to detect tampering, and thus must send a tag, as well.

There are fundamentally three different ways we could do this:

(1) Alice sends Encrypt(m || MAC(m))
- This is referred to as "MAC then encrypt"
- Used by SSL

(2) Alice sends Encrypt(m) || MAC(Encrypt(m))
- This is referred to as "Encrypt then MAC"
- Used by IPsec

(3) Alice sends Encrypt(m) || MAC(m)
- Used by SSH

So, what is the right one to use?

First, keep in mind that a MAC might leak information about the message: that was not part of our definition of security, so we do not have any explicit protection against it happening. As a result, (3) may not be secure, and thus should not be used.

Second, although the details are beyond the scope of this document, it is worth noting that there are combinations of encryption and MAC algorithms that render (1) insecure.

The most generally applicable approach is (2): "Encrypt then MAC". Let's reason as to why this is the case:

- Intuitively, Encrypt(m) reveals nothing about the message
- MAC ensures that no tampering has been done to the ciphertext
- At worst, MAC could leak information about the ciphertext, but that does not leak information about the original plaintext.

There are various standards that describe specific combinations and orderings of encryption and MAC algorithms. For example, 802.11i uses a standard called CCM, which applies ECBC MAC and then CTR mode encryption.

We will not be covering these standards in depth in this class, but it touches on a very important concept:

HOW TO BE A GOOD CRYPTOGRAPHY PRACTITIONER

In addition to the broader advice of staying up-to-date and informed of what mechanisms and key sizes are considered secure, it is important to

FOLLOW THE STANDARDS.

We will be discussing a few standards, which span topics such as how to choose keys, how to set initialization vectors, which combinations of encryption and MAC algorithms to use, etc. As future practitioners, it it important that you follow these to a tee.

Another important point that I have mentioned in class is that you should not take it upon yourself to design your own cryptographic mechanisms nor to implement them yourself. USE THE EXISTING LIBRARIES AS SPECIFIED.

==========

KEY EXCHANGE

There is still one major problem with symmetric key crypto that we have yet to address: How do Alice and Bob establish their key in the first place?

A natural solution to this problem is to establish these keys "physical channel", such as meeting in person. But this would not scale to meet the demands of the Internet (we would all have to visit Amazon just to set up a key to allow us to securely purchase from their website).

Let us try to tackle this problem by first assuming an attacker that can only eavesdrop (she cannot tamper). In this setting, we have one of the most beautiful results from cryptography.

DIFFIE-HELMAN KEY EXCHANGE

(1) Alice choses:
- a random, large prime p
- a positive integer 1 < g < p that is a "generator" (a.k.a. "primitive root") for prime p. This means that for all k = 1, ..., p-1, there exists an integer i such that k = g^i mod p. For example, 3 is a generator for 5: 3^1 mod 5 = 3, 3^2 mod 5 = 4, 3^3 mod 5 = 2, and 3^4 mod 5 = 1.)

She sends both p and g to Bob.

(2a) Alice chooses a random x and computes A = g^x mod p
(2b) Bob chooses a random y and computes B = g^y mod p

(3a) Alice sends Bob A
(3b) Bob sends Alice B

(4a) Alice computes K = B^x mod p
(4b) Bob computes K = A^y mod p

(5) Alice and Bob's shared secret key is K = g^(xy) mod p

(Work this out to see why Alice and Bob end up with the same K.)

Note that Alice and Bob had *no* shared information prior to running this protocol. Note further that, with our passive eavesdropper, Eve is able to overhear *all* communication between Alice and Bob. It may seem surprising (if this is the first time you are seeing this, it *should* seem surprising) that this scheme derives any security whatsoever.

To reason about the security, let's start by looking at what Eve gets to see:

- The prime p and the integer g
- g^x mod p from Alice
- g^y mod p from Bob

So why can't Eve compute the secret key g^(xy) mod p? First, note that simply multiplying g^x * g^y will result in g^(x+y). So that doesn't do the trick.

If we weren't operating over "mod p", then Eve could simply compute log_g(g^x) to obtain x, and then simply compute (g^y)^x.

But it is precisely because we are operating over "mod p" that she cannot do this. This is due to what is known as the DISCRETE LOG PROBLEM, which basically states that taking logarithms modulo some prime is hard. More specifically:

- given g and p
- given g^x mod p
- it is hard to compute x

This is no less difficult for Alice or Bob: Alice was not able to infer Bob's choice of y given B = g^y, but the key was that she didn't *have* to: she simply raised this value to her secret value x, and vice versa for Bob.

There are more details to what makes this problem hard, and in fact it boils down to careful consideration for the values of p and g. These are detailed in undergraduate crypto courses.

The details to which we have covered Diffie-Helman key exchange are intended to serve as a proof of existence: it is possible to exchange keys online in an adversarial setting.

However, recall that our adversarial setting was rather weak: Alice and Bob were only dealing with a passive eavesdropper. An active man in the middle could easily violate the correctness of this protocol. As an exercise, describe a protocol that an attacker could use to make Alice and Bob think that they have established a pairwise key with one another, but in reality, they are both communicating through the attacker.

What we really want is a way to ensure authenticity---that Alice and Bob can be sure that they have established a key with one another. This is known as AUTHENTICATED KEY EXCHANGE.

It is impossible to solve this problem without sharing SOME information in advance.

To solve this, we will be making use of our third and final black box: asymmetric key crypto.