ISA 562: Information Security, Theory and Practice

Lecture 2

1 Achieving IND-CPA security

1.1 Pseudorandom numbers, and stateful encryption

As we saw last time, the OTP is perfectly secure, but it forces us to generate and share a key that's as large as all of the plaintext we'll ever encrypt with it. This is a big drawback. One simple fix to this problem is to use a pseudorandom number generator (PRG). A PRG takes a short, uniformly sampled *seed*, and expands it into an arbitrarily large *pseudorandom* string. What do we mean by pseudorandom? Suppose that our PRG takes random seeds of length ℓ and expands them to be of length $n \gg \ell$. Since there are only 2^ℓ possible inputs to this PRG, there are at most 2^ℓ possible outputs. So, even though the output is a string of length n, and there are 2^n possible strings of length n, only $2^\ell \ll 2^n$ of these n-bit strings can possibly be in the range of the PRG. We haven't added any entropy. But if we are using a secure PRG, then we can prove that no polynomial-time machine can distinguish whether they've been given a pseudorandom string (resulting from choosing a random ℓ bit string and running it through the PRG), or a truly random string, chosen uniformly from $\{0,1\}^n$.

A very natural idea for encryption then follows. If two parties share a random seed of length ℓ, they can expand this as much as they'd like, and use a fresh portion of the pseudorandom string for each encryption. The only major drawback is that they have to keep track of which portions they've used so far, and which ones can still be used. (Remember, reusing the same OTP is completely insecure!) So, this gives us a *stateful* encryption scheme. This isn't ideal, but it can work in some settings. It also gives some intuition for what comes next.

1.2 Random Functions

Recall that a function $F : \{0,1\}^n \to \{0,1\}^n$ maps every n-bit input to some n-bit output. Imagine for the moment that both parties can afford to store the description of such a function $F : \{0,1\}^n \to \{0,1\}^n$, chosen uniformly at random from the set of *all* such functions. What would this keyspace look like? To help picture it, we can fix a particular way of representing such a function. We will do this with an array of size 2^n, using each slot of the array to store one of the function's output values. We order these outputs according to the ordering of the input: if $F(0) = y_0$, then we store y_0 in the first slot of the array; if $F(1) = y_1$, we store $y_1$in the 2nd slot of the array; if $F(i) = y_i$, we store y_i in the ith slot of the array. Fixing all of the bits of this array gives us the description of a single function that maps n input bits to n output bits. If we chose all of the bits of this array *at random*, this gives us a method for randomly choosing one such function. How many such functions are there? That is, if we view this as our keyspace, how large is the space? Since each output value is n bits, and we have to specify 2^n output values, the description of a function (using the above representation) requires $n2^n$ bits. Note that changing just a single bit in this representation gives us a different function. That is, every unique string of length $n2^n$ can be viewed as describing a different function. There are 2^{n2^n} strings of length $n2^n$, so this is exactly how many functions there are mapping n-bit inputs to n-bit outputs.

We turn now to using random functions for encryption. For the moment, we'll focus again on fixed length messages. Specifically, we'll assume that the message space, and ciphertext space are both $\{0,1\}^n$. As per the conversation above, we let the key space be the set of all functions mapping n bit inputs to n bit outputs. Formally, $\mathcal{K} = \{F \mid F : \{0,1\}^n \to \{0,1\}^n\}$. As we just discussed, the size of this key is prohibitively large, so this does not resolve the issue we had with using the one time pad construction. But it provides a good starting point for doing that. We claim the following encryption scheme satisfies IND-CPA security.

$\mathsf{Gen}(1^n)$: output the description of a randomly chosen function, $F : \{0,1\}^n \to \{0,1\}^n$.
$\mathsf{Enc}(F, m)$: Sample a random value $r \leftarrow \{0,1\}^n$. Let $c = F(r) \oplus m$. Output (c, r).
$\mathsf{Dec}(F, (c, r))$: Output $m = c \oplus F(r)$.

An important property of this encryption scheme is that it is not deterministic: if we encrypt the same message twice, we will choose two different values of r in each encryption (except with probability 2^{-n}), so the two resulting ciphertexts will look different from one another. As we mentioned last time, this is an essential property for any encryption scheme that meets the IND-CPA security definition. We won't prove in this class that this construction meets that definition, but the intuition stems from the proof that the OTP is secure: for large enough values of n, we are all but guaranteed that we'll never re-use the same value of r in multiple encryptions. Since F is a randomly chosen function, the adversary cannot know anything about $F(r)$, so it is a perfectly good OTP.

1.3 Pseudorandom functions

As we've already discussed, representing a truly random function would require far too much storage (and therefore runtime) for a reasonable encryption scheme. However, note that *some* functions can be represented much more simply. For example, $F(x) = 2x$ is a very simple function to store and evaluate. So it is only our insistence on supporting the full set of all possible functions on n-bits that is causing problems for us. We will instead introduce the idea of *pseudorandom functions*. This is a subset of the functions mapping n-bits to n-bits, with two properties: 1) the functions can be concisely represented, and 2) when a function is randomly chosen from this subset, the output is *indistinguishable* from random by any polynomial-time machine.

It's not obvious that pseudorandom functions (PRFs) even exist, but one of the big breakthroughs of modern cryptography is a proof that, as long as certain computational problems are hard, then PRFs do exist.[1] In practice, we instead build them heuristically, and they are commonly called *block ciphers*. The most well known are DES (Data Encryption Standard) and the newer AES (Advanced Encryption Standard). These block ciphers are single, publicly known algorithms, and the concise description of each function is simply encoded in an n-bit key. We therefore might write a particular function from the set as $F_k : \{0,1\}^n \to \{0,1\}^n$, with each possible value of k giving us a different function from the set. Since there are 2^n possible values for k, note that this set of functions is much smaller than the set of all 2^{n2^n} functions; nevertheless, if n is 256, this is still extremely large, and it is quite plausible to think that using a random function from this smaller set is indistinguishable from using a truly random function. Since the block cipher is a public algorithm, and only the key changes how the input is "scrambled", it is common to write

[1] As always, an example of such a problem is the factoring problem, but there are many others that suffice as well.

$F_k(r)$ as $F(k,r)$, treating F (e.g. AES) as a single, fixed function that maps a *pair* of n-bit inputs to a single n-bit output. Similarly, although a PRF is a set of functions, as previously described, but it is common to refer to this single, keyed function as a PRF; we should recognize that the key is what distinguishes one function in the set from every other.

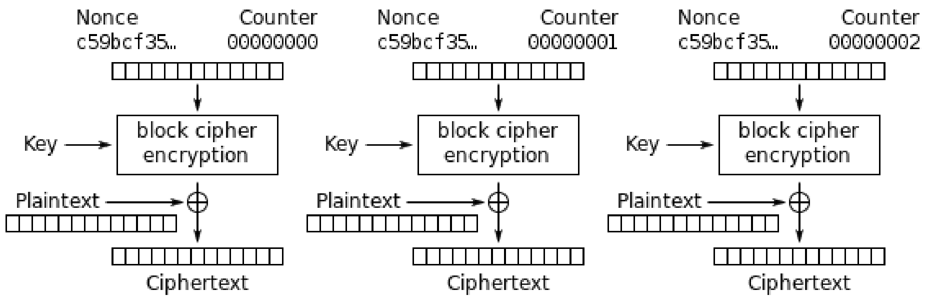
An important point that is often misunderstood is that PRFs and block ciphers are *not*, in themselves, secure encryption schemes. Recall the construction from the beginning of this lecture: we did not use the message as input to the random function. Had we done that, it is easy to see that encryption would be deterministic, and cannot possibly meet the security definition that we're aiming for. When using a PRF in place of the random function, the same issue arises, and we solve it in the same way we did before. We make it explicit here. Let F be a PRF (e.g. AES). Let the key space, message space, and ciphertext space all be $\{0,1\}^n$. Then the following encryption scheme is IND-CPA secure.

$\mathsf{Gen}(1^n)$: Randomly choose $k \leftarrow \{0,1\}^n$, and output k.
$\mathsf{Enc}(k,m)$: Sample a random value $r \leftarrow \{0,1\}^n$. Let $c = F(k,r) \oplus m$. Output (c,r).
$\mathsf{Dec}(k,(c,r))$: Output $m = c \oplus F(k,r)$.

1.4 Modes of operation

So far, we have only described an encryption scheme that supports n-bit messages, using n-bit keys. Of course, we were able to achieve this using the OTP! But, note that now we can re-use the key for multiple encryptions, which is an important distinction. In fact, this suggests a natural way to encrypt longer plaintexts: we simply need to break our plaintext up into blocks of length n (which is why PRFs are commonly called block ciphers), and encrypt each as it's own message. Extending the previous construction in the natural way, to encryption a message m that contains ℓ blocks, $m_1||\cdots||m_\ell$, we could simply choose ℓ random strings, $r_1,\ldots,r_\ell$, compute $c_i = F(k,r_i) \oplus m_i$, and output $(c_1,r_1),\ldots,(c_\ell,r_\ell)$. Note that with this approach, the ciphertext is twice as long as the original message. In practice, it is possible to do better. We now look at several different encryption methods (often called *modes of operation*) that are more commonly used.
ECB mode (Insecure!): Although it is insecure, the electronic codebook mode of operation is still frequently used, so it is good to know about. It shouldn't be used unless you're an expert in the field, you understand the application and the risk, and you're certain that it won't creep into other applications where the risk cannot be tolerated. We learn about it anyway, because it is important to know it when you see it being used. The mode is extremely simple: given message $m = m_1||\cdots||m_\ell$, compute $c_i = F(k,m_i)$, and output $(c_1,\ldots,c_\ell)$. Note that this is deterministic, so it cannot be IND-CPA secure! If the same message is sent at different times, the adversary will detect that. The adversary can also easily learn whether there are repeated blocks in the message.

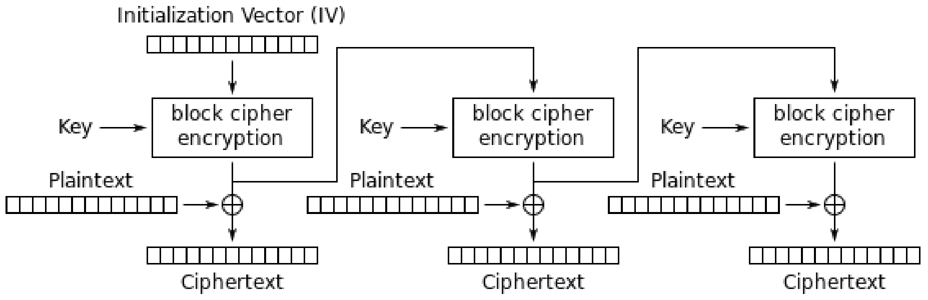
CTR mode:



Counter (CTR) mode encryption

In counter mode, we choose a random initialization vector (IV) – this is called a *nonce* in the above image – and increment this value for each block of the message. It is otherwise the same construction described previously, but because each of the ℓ inputs to the PRF can be determined by the first random IV, the remaining $\ell - 1$ inputs do not need to be sent with the ciphertext. Formally, letting $m = m_0 || \cdots || m_{\ell-1}$, $c_i = F(k, IV + i) \oplus m_i$, and we output $(IV, c_0, \ldots, c_{\ell-1})$. To decrypt, the ith block, you simply compute $m_i = F(k, IV + i) \oplus c_i$. One nice feature of this mode is that both encryption and decryption of each block can be parallelized.

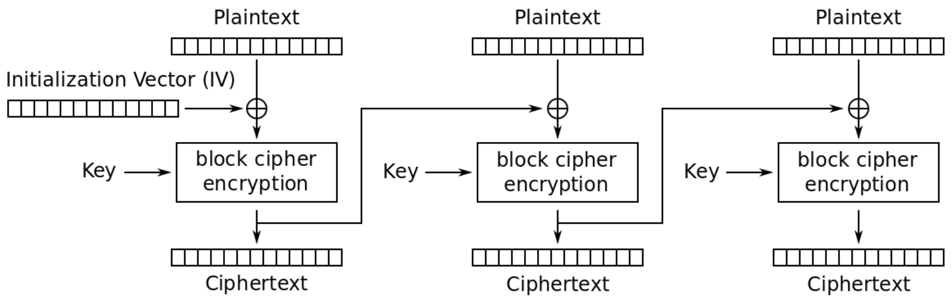
OFB mode:



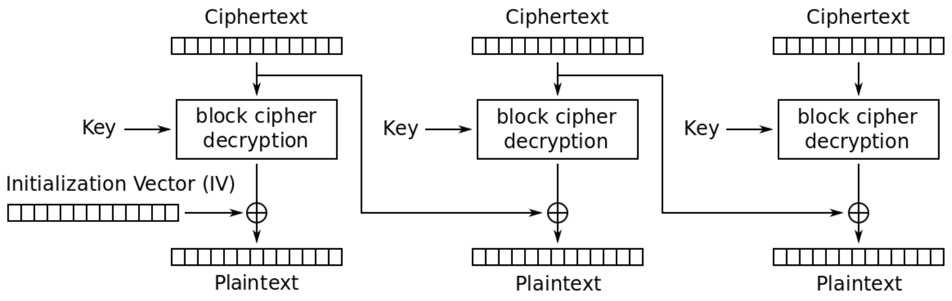
Output Feedback (OFB) mode encryption

In the output feedback mode, we derive the next OTP by running the previous one through the block cipher. That is, we start with a random IV, and let the first OTP be $r_1 = F(k, IV)$. We then compute the ith OTP, r_i as $F(k, r_{i-1})$. We use these ℓ OTPs to encrypt the plaintext via XOR. As with CTR mode, the ciphertext includes $(IV, c_1, \ldots, c_\ell)$. This is not as easily parallelized as CTR mode, though, we can compute this stream of OTPs before we have the message for encryption.

CBC mode:



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

The cipher block chaining mode the most frequently used mode of operation. We start with a random IV as a OTP for the first block of the plaintext, and the resulting value is fed through the block cipher: $c_1 = F(k, IV \oplus m_1)$. We output c_1 as part of the ciphertext, but we also use it as a OTP for the next block of plaintext, and feed that result through the cipher as well. Note that c_1 itself looks random, under the assumption that our block cipher is a good PRF. We proceed in this way for all of the remaining blocks: $c_i = F(k, c_{i-1} \oplus m_i)$, and we output $(IV, c_1, \ldots, c_\ell)$ as our final output.

To decrypt, note that we actually have to invert the PRF, so our PRF has to be a psuedorandom *permutation*, and it has to be easily invertible given the secret key. (In practice, all block ciphers are designed with this property.) To recover the first block of the message, we compute $F^{-1}(k, c_1) = m_1 \oplus IV$, and the use the IV to recover m_1. To recover the m_i, we compute $m_i = c_{i-1} \oplus F^{-1}(k, c_i)$. Note that we can decrypt the ith block using only 2 blocks of the ciphertext, so decryption can be parallelized.

One reason CBC became popular is that it *appears* to give some notion of tamper resilience: if someone changes a bit of the ciphertext, intuitively it would seem that this would greatly change the value of the resulting plaintext upon decryption. Unfortunately, this can't be formalized, and in fact there are some pretty bad attacks on this mode if we allow the adversary to observe what happens when we try to decrypt ciphertexts that he has manipulated, as we will soon see. We now know that authentication has to be handled carefully and correctly; ad hoc attempts do not suffice.

Messages of arbitrary length: Note that we have until now assumed that our plaintexts are block aligned, but we will frequently have messages whose lengths are not multiples of our block size. For CTR and OFB mode, we can generate our random pads for encryption, and just truncate the end to match the length of the message. For CBC, this does not work, since we need the entire last block in order to decrypt.

IV reuse: Coming up with a good source of randomness is not easy, so, rather than generating a fresh IV every time, it is common for people to maintain the state of the last encryption, and to use the IV generated for the last block of the prior encryption as the IV for the first block in the next encryption. For example, with CTR mode, if the inputs to the PRF during the prior encryption were $(IV, IV+1, \ldots, IV+\ell-1)$, then in the next encryption you could use $IV+\ell$ as the new IV. In OFB mode, if the OTPs from the previous encryption are $(IV_1, IV_2 = F(k, IV_1), \ldots, IV_\ell = F(k, IV_{\ell-1}))$, then in the next encryption we can use IV_ℓ as the IV, and $F(k, IV_\ell)$ as the first OTP.

It is important to note that IV reuse (or, stateful encryption, as it is sometimes called) is *not* secure in CBC mode. Consider the following attack. Suppose the adversary sees a ciphertext (IV, c_1, c_2, c_3), and he happens to know that the first block of the corresponding plaintext, m_1, is either m_1^0 or m_1^1. If the server holding the key is reusing IVs, then the IV for the next encryption is c_3. If the adversary can get the server to encrypt $IV \oplus c_3 \oplus m_1^0$, he will receive $c_4 = F(k, IV \oplus m_1^0)$. If $m_1 = m_1^0$, then this is exactly the value of c_1! Therefore, $c_1 = c_4$, then $m_1 = m_1^0$, and if $c_1 \neq c_4$, then he knows that $m_1 = m_1^1$.

1.5 Padding oracle attacks

We demonstrate an attack on CBC mode encryption, which demonstrates the need for authenticating our communication, including our ciphertexts. (Authentication is the act of proving that a message came from the party that shares your key, and that it has not been altered from the original. We will deal with the topic of authentication next.) Recall that in CBC mode, our plaintexts have to be padded to be a multiple of the block length. The PKCS7 standard describes the following method for padding, so that the padding can be easily removed at the receiver's end. If n is our block length, and L is the number of bytes in n (i.e. $L = n/8$), let b be the number of bytes needed to pad the message out to L bytes. Then, in each of those b bytes, you encode the value of b in binary. So, if the last block of your message is seven bytes short of L, you would repeat 00000111 in the last 7 bytes of your last block. During decryption, Dec is modified to look for and remove the padding: it first performs decryption as previously described for CBC mode, and then analyzes the last byte of the last block. If that value is b, then it verifies that the last b bytes of the last block *all* contain the value b. If they don't, then Dec outputs a failure symbol and terminates.

Suppose an adversary sees an encryption of a message $m = m_1||m_2$, with the form (IV, c_1, c_2), and he'd like to recover m_2. Suppose he has access to a server that will attempt to decrypt whatever message the adversary sends, and will give some indication if the decryption procedure fails. For the attack to work, the adversary does not need anything more than an indication of decryption failure. For example, it suffices if the server terminates the connection, or sends an error message. Recall that in CBC mode, $c_1 = F(k, m_1 \oplus IV)$, and $c_2 = F(k, m_2 \oplus c_1)$. It follows that that $m_2 = c_1 \oplus F^{-1}(k, c_2)$. The adversary knows nothing about $F^{-1}(k, c_2)$, of course, but he does know c_1. By flipping the ith bit of c_1, note that he exactly flips the ith bit of m_2, and nothing else. This observation is what leads to an attack.

To start, the adversary will submit a bunch of modified ciphertexts to the server to learn the length of the padding, b. To do this, he modifies the first byte of c_1, leaves IV and c_2 as they were, and submits the resulting ciphertext. Note that if the padding did not include the first byte, then this modifies the plaintext value, but does not cause any changes with in the padding: there are still b repetitions of the byte containing the value b. Decryption will not cause any error. The adversary repeats this, modifying the second byte, then the third, and so forth, and submitting each modified ciphertext. Eventually, when the adversary has modified byte $L - b + 1$, which is the first byte of padding, m_2 will contain b in the last $b - 1$ bytes, but some value *other* than b in the byte just prior to that (i.e. in byte $L - b + 1$). This will lead to a decryption error, and the adversary will have recovered the value of b.

The next step is to start recovering bytes of m_2, one byte at a time, starting with byte $L - b$, which is the byte just prior to the padding. The idea here is a bit more subtle. The adversary will try to modify the plaintext in a particular way, again by modifying bits of c_1. Note that, knowing the value of b, one thing the adversary can do is replace the content of the last b bytes of m_2 with the value $b + 1$. To do that, he could compute

$$\hat{c}_1 = c_1 \oplus (\underbrace{0, 0, \ldots, 0,}_{L-b \text{ bytes}} \underbrace{b \oplus b + 1, \ldots, b \oplus b + 1}_{b \text{ bytes}}).$$

If he submitted $(IV, \hat{c}_1, c_2)$ for decryption, note that the first $L-b$ bytes of m_2 would be unchanged, whereas in the last b bytes, the content of each byte would be $b \oplus (b \oplus b + 1) = b + 1$. Now, to figure out the content of the byte just prior to the padding, the adversary will submit a bunch of modified ciphertexts, each looking similar to $\hat{c}_1$, but with a change in byte $L - b$. Suppose the content that byte is B. If the adversary modifies the content of that byte such that it contains the value $b + 1$, the modified m_2 would have correct padding: it has b copies of $b + 1$ because of the way $\hat{c}_1$ is constructed in the last b bytes, and because this new modification to byte $L - b$ adds one more copy of $b + 1$, decryption can proceed without failure. On the other hand, if a modification to byte $L - b$ results in anything other than the value $b + 1$, then the padding is *not* well formed, and a decryption error should result. This is because the last byte of m_2 contains $b + 1$, so the decryption procedure expects $b + 1$ copies of that value, which should include a copy in byte $L - b$. We therefore define 256 different ciphertexts, each a different variant of $\hat{c}_1$:

$$\Delta_i = c_1 \oplus (\underbrace{0, 0, \ldots, 0,}_{L-b-1 \text{ bytes}} i, \underbrace{b \oplus b + 1, \ldots, b \oplus b + 1}_{b \text{ bytes}}).$$

When the adversary submits $(IV, \Delta_i \oplus c_1, c_2)$, the plaintext m_2 is unchanged in the first $L - b - 1$, it contains $B \oplus i$ in byte $L - b$, and it contains $b + 1$ in the last b bytes. If $i = B \oplus b + 1$, then decryption will continue without failure, whereas if i is any other value, it will fail due to invalid padding. The adversary submits all 256 ciphertexts, and observes which one does not result in a failure. Supposing $c_1 \oplus \Delta_i$ did not lead to a failure. From the value of i, he computes $B = b + 1 \oplus i$ to recover B, and then moves on to the prior byte in a similar manner. (Note that because byte $L - b$ of m_2 contains value B, the manipulation he uses in byte $L - b$ when recovering byte $L - b - 1$ should use an XOR value of $B \oplus b + 2$, rather than $b \oplus b + 2$.)

It is worth asking where we went wrong, given that we claim to have a proof that CBC mode encryption is secure. The answer is that our IND-CPA security definition (indistinguishability under chosen plaintext attack) did not consider the ability of the adversary to request decryptions of ciphertexts of his choosing. Indeed, there is a stronger security definition, called IND-CCA (indistinguishability under chosen ciphertext attack) which does take this capability into account. None of the modes that we discussed would meet this stronger security definition.

We won't go into the proper way to define such security, but a crucial component for achieving such security is message authentication. It is an important lesson that encryption does not provide authentication: clever manipulation of ciphertexts can have devastating impacts on the plaintext values. We will talk about message authentication in the next lecture.