Notes #4 (for Lecture 7)

We are considering the encryption setting where A and B share a random n-bit key k, and they wish to securely communicate over an insecure channel. We will only consider here the "uni-direction conversation" case where A has a long message to communicate, small pieces at a time, to B.

In fact, this notion of a shared-private-key cryptosystem (or *session protocol*) should really be viewed as a *case study* of issues involved in defining and obtaining secure session protocols. Such a system may be used as a component of a larger, more practical, system. In any case, however, the ideas that we discuss here will certainly be important in any reasonable cryptographic system that one devises.

We consider for now one example of using our notion of a session protocol in a larger context. Let us say that users A and B want to have a "2-way conversation". For example, A talks for a while, ending with the word "over"; then B talks, ending with "over"; then A talks, etc. One way of doing this securely is as follows. A and B initially share two n-bit randomly chosen keys k_1 and k_2. They use k_1 for A to encrypt pieces to B as in our notion of session protocol, and they similarly use k_2 for B to encrypt pieces to A. Of course, these two sessions are interspersed in a complicated way, and we want to make sure that our notion of security for a session protocol is strong enough that the resulting "2-way conversation" protocol is secure. This makes it important, for example, that the adversary be able to choose the input message himself. More generally, we want our notion of security for a session protocol to be strong enough that any reasonable use of such a system will itself be secure.

We have not discussed the issue of *how* A and B actually wind up sharing a random key. This can be done by a low tech method such as flipping some coins, and then having a meeting or using a secure delivery service. It can also be done using a "key exchange" protocol that assumes some existing infrastructure such as "higher-level" shared keys, or such as "public keys"; this will be discussed in more detail later.

Before defining carefully what we mean by a shared-private-key cryptosystem – sometimes called a session protocol or a "unidirectional session protocol" – we will give an example. In this example, the message comes to A in n-bit "pieces". For any integer i, we define $\bar{i}$ to be the n-bit representation of i.

Cryptosystem I. Let F be a (hopefully pseudo-random) function generator.

For an n-bit key k and n-bit message pieces $m_0 m_1 m_2 \cdots$, A encrypts each m_i by $e_i = m_i \oplus F_k(\bar{i})$ and sends $e_0 e_1 \ldots$ to B.

It is easy to see how B should decrypt. For an n bit key k and received encrypted pieces $e'_0 e'_1 \ldots$ where each $|e'_i| = n$, B computes for each i:
$m'_i = e'_i \oplus F_k(\bar{i})$, and B outputs m'_i as the i-th decrypted message piece.

What security properties does Cryptosystem I possess? One property it definitely does *not* possess is "integrity", or what we shall define later as *unchangeability security*. This is because an adversary who has control over the channel when Cryptosystem I is being performed can "change" a message, that is, trick B into outputting something wrong. In fact, no matter what the adversary

chooses to send to B, B will output *something*. The situation here is especially bad, since the adversary can easily make B output something wrong but nonetheless related to the original message. For instance, he can flip some chosen bits of the original message.

However, we will see that Cryptosystem I does possess "privacy", or what we shall define later as *indistinguishability security*: an adversary cannot learn anything about the message. Even if he chooses part of the message himself, he cannot learn anything about parts of the message that he does not choose.

We will later define a session protocol to be *totally secure* if it satisfies both integrity and privacy.

The following cryptosystem does not possess any privacy at all, but we will see later that it does satisfy integrity.

Cryptosystem II. Let F be a (hopefully pseudo-random) function generator where for $k \in \{0,1\}^n$, $F_k : \{0,1\}^{2n} \to \{0,1\}^n$.

For an n-bit key k and n-bit message pieces $m_0 m_1 m_2 \cdots$, A encrypts each m_i by $e_i = [m_i, F_k(\bar{i}\, m_i)]$ and sends $e_0 e_1 \ldots$ to B.

Here is how B should decrypt. For an n bit key k, say that B receives encrypted pieces $e'_0 = [m'_0, \beta_0], e'_1 = [m'_1, \beta_1], \ldots$ where each $|m'_i|, |\beta_i| = n$. For each i, B checks that $\beta_i = F_k(\bar{i}\, m'_i)$; if so, B outputs m'_i as the i-th decrypted message piece; if not B outputs a special symbol FAIL, and then halts (or continues to output the symbol FAIL), in effect aborting the protocol.

Note that it is necessary for B to be able to *fail* in a decryption, since otherwise it would be impossible to achieve integrity. We could just have B ceasing to output rather than explicitly outputting FAIL, but we prefer for B to explicitly output something. This is because in practice, the fact of B failing or aborting in a protocol is a positive, noticeable action that the world – including the adversary – may see.

We also could have allowed B to output some pieces that decrypt properly after failing for some others, but it is simpler (for our definitions) to simply abort the whole protocol the moment B sees that something is wrong. This raises the question of how A and B are supposed to "recover" once B discovers an error and aborts. The error may be due to an adversary – who may or may not still be present – or to noise on the channel. We will not discuss here how to recover from such an error. In practice this is a very important issue, and a poorly designed recovery method may introduce insecurities. For instance, if we ever have A and B start over again with the same key, then this in general will cause a system to become insecure.

Definitions

We will now define the notion of a shared-private-key cryptosystem, or session protocol; this will be followed by definitions of *unchangeability security* (or integrity) and then *indistinguishability security* (or privacy). Such a cryptosystem consists of an Encryption algorithm and a Decryption algorithm. Both algorithms have access to a security parameter n as well as to a shared private key of length related to n. For convenience, we will assume that the key is of length exactly n, although in practice we will allow it to be of length $2n$ or n^2, for example, as convenient. (Recall that if we require a random key to be of length exactly n, we can use a pseudo-random number generator to expand it to a longer length.) A cryptosystem also specifies, for security parameter n, three lengths: $z(n)$, $z'(n)$, and $z''(n)$.

$z(n)$ will be the length of each "piece" that is individually encrypted or decrypted. For example, in each of Cryptosystems I and II, we have $z(n) = n$. A large value of $z(n)$ places a constraint on A and B, since B cannot start to decrypt a piece until the entire piece has been read and encrypted

by A. A large value of $z(n)$ also places a constraint on the adversary. Recall that the adversary will be choosing the message pieces himself, after seeing encryptions of previous message pieces. He must choose a piece all at once; he will not be allowed to choose just part of a piece and see its encryption (and this may not even make sense) before choosing the rest of the piece. In effect $z(n)$ specifies the "granularity" of the cryptosystem.

$z'(n)$ will be the size of the encryption of each piece. For example, in Cryptosystem I, $z'(n) = n$, and in Cryptosystem II, $z'(n) = 2n$.

$z''(n)$ will be an upper bound on the size of a "history" or "memory" that A and B must remember in order to perform the encryptions and decryptions. For some systems (we shall see later) this will be 0, that is, no memory is necessary. In Cryptosystems I and II above, both A and B must remember the number of pieces (mod 2^n) that have already been encrypted/decrypted; it suffices, therefore, that $z''(n) = n$.

We will allow encryption to be probabilistic, but not decryption. Note that in Cryptosystems I and II above, encryption is deterministic. We will see examples later of probabilistic encryption. In fact, it is easy to see that if we want a cryptosystems where no history has to be remembered (that is, $z''(n) = 0$) and that satisfies indistinguishability security, then we need encryption to be probabilistic. This is because if encryption is deterministic and history-free, then identical message pieces will be encrypted identically, making the system not indistinguishability secure. Note, however, that even probabilistic encryption does not make it possible to be history-free and still satisfy unchangeability security. This is because if pieces are being encrypted in a history-free way, it is trivial for an adversary to reorder, repeat, or omit pieces without being detected.

Definition: A *shared-private-key encryption scheme*, or *session protocol* consists of algorithms *ENC* and *DEC*, and length functions $z(n)$, $z'(n)$ and $z''(n)$; it should be the case that $z(n)$, $z'(n)$ and $z''(n)$ are all computable (in unary) in time polynomial in n. *ENC* and *DEC* should satisfy the following:

- *ENC* has as input the following bit strings: the key k of (say) length n, a "piece" m of length $z(n)$, a "history" h of length $z''(n)$, and possibly a string of random bits *rand*. *ENC*$(k, m, h,$ *rand*$)$ should be computable in time polynomial in n, and should consist of a bit string of length $z'(n)$ representing the encryption of m, together with a bit string of length $z''(n)$ representing the new "history" that has to be remembered in order to encrypt the pieces that will follow. (Our convention is that the original history is the all 0 string.)

- *DEC* has as input the following strings: the key k of length n, a (supposedly) encrypted piece e of length $z'(n)$, and a "history" h of length $z''(n)$. *DEC*(k, e, h) should be computable in time polynomial in n; it should consist of either the special symbol FAIL, or of a bit string of length $z(n)$ (that is the supposedly decrypted piece) together with a bit string of length $z''(n)$ representing the new history that has to be remembered in order to decrypt the pieces that will follow. In case the output is FAIL, the decryption process halts. (Our convention is that the original history is the all 0 string.)

- We now state the condition that, in the absence of an adversary, the cryptosystem must work as expected, that is, pieces must be decrypted properly.
  Let k be a bit string of length n.
  Then for every bit string m of length $z(n)$ and every bit string h of length at most $z''(n)$ and every bit string *rand* of the appropriate length, if *ENC*$(k, m, h,$ *rand*$) = (e, h')$, then *DEC*$(k, e, h) = (m, h')$.

(Note that this is actually stronger than what we need, since it stipulates that the decryptor remembers exactly the same histories as the encryptor. More generally, we can insist on the following:
Let $m_0, m_1, \ldots$ be a sequence of pieces each of length $z(n)$. Define the sequences $e_0, e_1, \ldots$ and $h_0, h_1, \ldots$ and $m'_0, m'_1, \ldots$ and $h'_0, h'_1, \ldots$ as follows, where k is an n-bit string and $rand_0, rand_1, \ldots$ are (appropriate size) bit strings:
$h_0 = h'_0 =$ the all 0 string;
$(e_i, h_{i+1}) = ENC(k, m_i, h_i, rand_i)$;
$(m'_i, h'_{i+1}) = DEC(k, e_i, h'_i)$ (and in particular, we insist $DEC(k, e_i, h'_i) \neq$ FAIL).
Then it must be the case that for all i, $m'_i = m_i$.)

It is easy to see how both Cryptosystems I and II can be expressed so as to be in the above specified form.

We will now define *unchangeability security* – or *integrity* – for a session protocol (ENC, DEC). We wish to express the idea that an adversary cannot cause B to output something wrong. Recall that an adversary has complete control over the channel, and so he can in effect cut it so that B stops decrypting, or he can send garbage over it so that B outputs FAIL and aborts. But he shouldn't be able to make B output for his i-th piece a string m'_i different from the string m_i that A encrypted. We allow the adversary himself to choose the input pieces for A in an interactive manner: he chooses m_0, sees the encryption e_0, chooses m_1, sees the encryption e_1, etc. After doing this for a polynomial amount of time, he computes a string α that he sends to B. Keep in mind that α is the only string he sends to B, and it may or may not bear any relationship to e_0, e_1, etc.

More formally, we shall deal with the "nonuniform adversary" setting.
The adversary will be a polynomial-size family $\{C_n\}$.
C_n will have as input the bits 0 and 1;
C_n will also have as input (a polynomial number of) $z'(n)$-bit strings $e_0, e_1, \ldots, e_\ell$, but it can only access them in the following way.
First C_n, using only inputs 0 and 1, computes a $z(n)$-bit string m_0;
then C_n can use e_0 to compute a $z(n)$-bit string m_1;
then C_n can use e_1 (as well as e_0) to compute a $z(n)$-bit string m_2;
$\vdots$
then C_n can use $e_{\ell-1}$ (as well as $e_0, e_1, \ldots, e_{\ell-2}$) to compute a $z(n)$-bit string m_ℓ;

Eventually, C_n uses $e_0, e_1, \ldots, e_\ell$ to outputs a string α. The intuition is that α consists of parts of length $z'(n)$ that will be decrypted by B.

We define the probability $p_C(n)$ in the obvious way. Consider the following experiment.
A random n-bit string k is chosen, as well as random bit strings of the appropriate length $rand_0, rand_1, \ldots$ and we let h_0 be the all 0 string;
C_n computes m_0;
let $(e_0, h_1) = ENC(k, m_0, h_0, rand_0)$ and give e_0 to C_n; C_n computes m_1;
let $(e_1, h_2) = ENC(k, m_1, h_1, rand_1)$ and give e_1 to C_n; C_n computes m_2; etc;
eventually, C_n outputs a string α.
Run DEC on key k and encrypted-piece-stream α to obtain a string of supposedly decrypted pieces that are each of length $z(n)$: $m'_0, m'_1, \ldots, m'_{\ell'}$ (we omit FAIL from this list, if it occurs).
Define $p_C(n)$ to be the probability that there exists an i, $0 \leq i \leq \ell'$ such that either $i > \ell$, or $i \leq \ell$

and $m_i \neq m'_i$. Note that the randomness in this experiment includes the choice of k, as well as all the randomness, if any, used by *ENC*.

Definition: We say a session protocol is *unchangeability secure* – or satisfies *integrity* – if for every adversary C as described above, $p(n) \leq \frac{1}{n^d}$ for each d and sufficiently large n.

This definition is more subtle and more robust than it might at first appear. For example, if we had insisted that ℓ' be no bigger than ℓ, then the definition would not have been weakened (exercise!).

Also, it would not strengthen the definition if we allowed the adversary to give parts of α to B, and then see the reaction of B, before creating further parts of α. The reason is that the adversary should assume that the i-th part of α of length $z'(n)$ is decrypted as m_i, since if it is decrypted as something else then the adversary has already won, and if it causes FAIL then it is too late for the adversary to win if it has not done so already.

There is something else that this definition gives us for free. In practice, because of the presence of noise, data sent over a communication channel is usually sent using – at a low level of the network protocol – "error detecting/correcting codes". We are viewing this error correction as being part of the channel, rather than part of the encryption/decryption protocol. But what if noise is so bad or so special that errors creep in that are *not detected*? Because of unchangeability security, this (almost certainly) will not violate integrity: noise on the channel can just be viewed as a dumb kind of adversary, and we are automatically protected against it causing errors. Of course, we do not – and in our model we cannot – protect against noise or an adversary causing FAIL, that is, abortion of the protocol.

The fact that our adversary is assumed to have complete control of the channel makes our definition of security very strong, but it also makes it impossible for us to meaningfully discuss the possibility of protecting – even somewhat – against "denial of service attacks". In order to discuss this, we would have to deal with a large network setting where it is assumed that an adversary is restricted in how much control he can achieve, and that is outside the scope of these discussions. Keep in mind that our "channel" may in practice be implemented as the entire world-wide internet; it is only the fact that we allow the adversary complete control over it that allows us to think of the channel as a single wire.

Theorem: Cryptosystem II satisfies integrity (i.e., unchangeability security).

Proof: Say that adversary C breaks the unchangeability security of Cryptosystem II. We will show how to break the pseudo-randomness of the underlying function generator F. Fix n such that $p_C(n) > 1/n^d$.

Say we are given a function $f : \{0,1\}^{2n} \to \{0,1\}^n$.
Using C_n we create $m_0 \in \{0,1\}^n$, see $\beta_0 = f(\overline{0}\, m_0)$, we create $m_1 \in \{0,1\}^n$, see $\beta_1 = f(\overline{1}\, m_1)$, ..., we create $m_\ell \in \{0,1\}^n$, see $\beta_\ell = f(\overline{\ell}\, m_\ell)$. Let us denote $e_i = [m_i, \beta_i]$.
We now use C_n to compute a sequence of strings for a decryptor: $e'_0 = [m'_0, \beta'_0], \ldots, e_{\ell'} = [m'_{\ell'}, \beta'_{\ell'}]$. Our new adversary will accept if there is an i such that $i > \ell$ or $e'_i \neq e_i$, and if for the first such i, $m'_i \neq m_i$ (if $i \leq \ell$) and $\beta'_i = f(\overline{i}\, m'_i)$.

A pseudo-randomly generated f will be accepted with probability $> 1/n^d$.

Now say that f is randomly generated. To accept f there must be some i such that,
$e'_0 = e_0, e'_1 = e_1, \ldots, e'_{i-1} = e_{i-1}$ and such that $e'_i = [m'_i, f(\overline{i}\, m'_i)]$ where (if $i \leq \ell$) $m'_i \neq m_i$. But C_n does not see any value of $f(\overline{i}\, u)$ except for $u = m_i \neq m'_i$ (if $i \leq \ell$), and so the probability that C_n will guess the value of $f(\overline{i}\, m'_i)$ is $1/2^n$. $\square$

It is interesting to consider variants of Cryptosystem II. For example, what if instead of letting $e_i = [m_i, F_k(\bar{i}\, m_i)]$ we had defined $e_i = [m_i, F_k(\bar{i}) \oplus m_i]$? This would *not* satisfy unchangeability security. An adversary could let $m_0 = \overline{0}$, and then see $e_0 = [\overline{0}, F_k(\overline{0})]$; he could then let m_0' be anything he likes and send $e_0' = [m_0', F_k(\overline{0}) \oplus m_0']$ to B; this would cause B to output the decrypted message piece m_0'.

It would also not satisfy unchangeability security if we let $e_i = [m_i, F_k(\bar{i} \oplus m_i)]$. An adversary could let $m_0 = m_1 = \overline{0}$, and then see $e_0 = [\overline{0}, F_k(\overline{0})]$ and $e_1 = [\overline{0}, F_k(\overline{1})]$; he could then send $e_0' = [\overline{1}, F_k(\overline{1})]$ to B; this would cause B to output the decrypted message piece $m_0' = \overline{1}$.

Next lecture we will define *privacy*, or *indistinguishability security*. This essentially says that an adversary who is able to choose all message pieces except one is not able to learn anything about that one.

When defining security, it is important that we allow our adversary to have access to all information and abilities that a real adversary might have. There is at least one important sense, however, in which our definitions in this course are inadequate. The adversaries in our definitions have access to the results of a party's computations, but not to the actual time that is spent performing those computations. In the real world, in spite of the latencies of the internet, an adversary might be able to get information about how long it takes a particular computation to be performed, and this in turn could potentially give helpful information away about a key, or about something that is being encrypted or decrypted.

Although these considerations will not play a role in out definitions, the reader should keep in mind that it is crucial that when a computation is performed, it is done in such a way that the time taken is completely independent of the values of the input strings. It will be assumed that everybody knows the *lengths* of these strings, but nothing else about them should be given away by computation times. This may involve some subtle programming. The use of optimizing compilers and caches can make this task especially difficult.

There is one setting in which it is especially easy to get timing – and other – information: the "smart card". We consider a "smart card" to be a computing device (usually quite small) that has embedded in it a key or some other information that is supposed to be secret from the owner/user. The user should be able to use the card to do certain computations, but *nothing else*. Any attempt to "open" the card should make it self-destruct. However, there are all sorts of measurements one can perform on the card. Certainly timing measurements are easy to do. One can also measure power used, as well as the noise made by any hard drive or other component. One attack that is especially interesting is to do measurements on the *analogue* signals that are produced by the card, rather than merely looking at their digital translations. We will ignore all of these issues in these notes.

There is another kind of information that an adversary can get that our models will always ignore: an adversary, by listening in on various channels, may be able to learn who is talking to whom, when, and for how long. This is called "traffic analysis". We do not pretend to protect against this at all in these notes. Of course, you can protect against this by always sending traffic all of the time to everyone you know, but this is inefficient. More practical approaches are possible, but these can only be discussed in a more sophisticated network setting where, unlike in these notes, the adversary is restricted in how much of the network he can control.