Computer Science 2426F — Fall, 2020
St. George Campus — University of Toronto

Notes #10

In previous notes we have defined secure session key exchange in a PKI, and we have defined one notion of what it means to have a secure session between two parties that share a random session key. Now we really should define what it means for two parties to have a secure session in a PKI, and then prove that composing two protocols secure in the former senses yields one secure in the latter sense.

This is complicated, and we will not define here what it means to have secure sessions in a PKI. Keep in mind that we want to assume the "multi-session" setting, where each person can spawn many identical processes, and the adversary is allowed to create many identities for himself.

We will now define the notion of "public-key encryption primitive". Such a primitive has many cryptographic applications, a common one being session key exchange in a PKI. The reader should note that such a primitive is rarely used to directly encrypt the information in a session: this would be very inefficient, and hard to do securely.

A "public-key encryption primitive" works as follows: on security parameter 1^n, a procedure GEN is used to probabilistically generate a pair of keys pub and pri; an encryption algorithm ENC uses pub to probabilistically encrypt strings of fixed length – usually we will let this length be n; a decryption algorithm DEC uses pri to decrypt. The formal definition is given below.

Definition: A *public-key encryption primitive* $\mathcal{E}$ consists of the following.

- A key generating procedure GEN. GEN has as input a string 1^n together with random bits, and should be computable in time polynomial in n. The output of GEN is a pair of strings pub (a public key) and pri (a private key). We assume that the lengths of pub and pri depend only on n, and that n is determined by either of these lengths; for convenience, let us denote the length of both pub and pri by $\ell(n)$.

- A probabilistic encrypting procedure ENC that has as input a "key" pub of length $\ell(n)$, a string $m \in \{0,1\}^n$, and random bits. ENC should be computable in time polynomial in n. (We will assume that the length of the output of ENC depends only on n.) We denote by $ENC_{pub}(m,*)$ the probabilistic function of m computed by ENC on key pub (where * represents the random bits).

- A decrypting procedure DEC that has as input a "key" pri, of length $\ell(n)$ and a supposed encryption α (of the proper length), and outputs either a bit string of length n, or the symbol FAIL (indicating that α is not a legitimate encryption). DEC should be computable in time polynomial in n. We use $DEC_{pri}(\alpha)$ to denote $DEC(pri, \alpha)$.

  It should be the case that for every n, and for every pair (pub, pri) that can be output by GEN on 1^n, and for every string $m \in \{0,1\}^n$, if α is a possible output of $ENC_{pub}(m,*)$, then $DEC_{pri}(\alpha) = m$.

A few things should be noted about this definition. Note that the m that is encrypted is n bits long – it is not an arbitrary length string and it is not a single bit. Our definition of strong security will be so special that it is not obvious how to go from a secure primitive for encrypting a single bit to a secure primitive for encrypting an n-bit string. Also, in applications, all we have to do is use the primitive to encrypt a fixed length (that depends on the security parameter) string, rather than a string whose length depends on the actual amount of data to be transmitted – for example, the primitive might only be used for session key exchange. One should also note that encryption is *probabilistic*; this is important for our definition of security.

Strong Security

We now want to define security for a public-key encryption primitive. Our main definition we will call "strong security". In this definition, the adversary sees an encryption α of one of two strings, and he has to guess which one was encrypted. To help him, he sees not only the public key, but he also has access to the decryption function; that is, he can see the decryption of anything he likes, except for α. This is sometimes called "security against chosen-cipher-text attack" security, or CCA (or CCA2) security. After defining this, we will define two weaker notions of security.

Definition: *Strong Security For a Public-Key Encryption Primitive*
(nonuniform adversary setting).
Let $\mathcal{E}$ be a public-key encryption primitive. Without loss of generality, say that $l(n)$ is the length of all keys and encryptions for security parameter n. We say that $\mathcal{E}$ is *strongly secure* if the following holds for every adversary D.

Let $D = \{D_n\}$ be a polynomial size family of circuits. Consider the following experiment: GEN is evaluated on 1^n and random bits, to create *pub* and *pri*. D_n is given the string *pub* as well access to an oracle for the function DEC_{pri}. After querying the function for a while, D_n chooses two n bit strings M^0 and M^1. Then b is chosen randomly from $\{0,1\}$, and a random encryption α of M^b is created (*not* by D_n) using *pub* (and random bits). Then α is given to D_n, and D_n continues to query the function DEC_{pri}, but is not allowed to query the function on α.
Let $q(n)$ be the probability that D_n outputs b.

Then $q(n) \leq \frac{1}{2} + \frac{1}{n^c}$ for all c and sufficiently large n.

Semantic Security and Some Background Remarks

If we weaken this definition so that D_n is not allowed to query DEC_{pri} after seeing α, then this is sometimes called *CCA1 security*. If we weaken it still further so that D_n is not allowed to query DEC_{pri} at all, it is called *semantic security*.

In practice, CCA1 security is not much more useful than semantic security. Semantically secure public key encryption primitives can be obtained from a Diffie-Hellman discrete log assumption, or (in the RSA setting) from a strong assumption about the difficulty of integer factorization. For example, given the **Assumption** from Notes #9, we can prove that the "naive" example of a public key encryption primitive from Notes #8 satisfies semantic security. (In fact, the normal (rather than decisional) Diffie-Hellman assumption suffices to construct semantically secure public key encryption primitives.)

It is much harder, however, to construct strongly secure public key encryption primitives. In practice, the main reason a public key encryption primitive will not be strongly secure is that it will be *malleable*. Malleability means that an adversary, seeing an encryption α of a string m, can modify it to a different encryption α' that decrypts to a string m' related to m. This clearly makes the primitive insecure. For this reason, all the proposed constructions for strongly secure public key encryption primitives more or less (but not exactly) have the following property: if an adversary queries DEC_{pri}, then he already "knows" the answer – either he has constructed the query by legitimately encrypting some m and so he knows that m will be the answer, or he has chosen the query some other way and the answer will be FAIL. This clearly rules out malleability.

Cramer and Shoup have shown a very elegant construction of a public key encryption primitive that can be proven to be strongly secure assuming only the Decisional Diffie-Hellman Assumption. Much more theoretically and less usefully, we can show that "trapdoor permutations" can be used (in a very inefficient manner) to construct strongly secure public key encryption primitives. This result shows that, at least theoretically, a strong assumption about the difficulty of integer factorization can be used to construct strongly secure public key encryption primitives.

It is possible to use a "random oracle", together with a rather weak trapdoor complexity assumption, to construct a strongly secure public key encryption primitive. Such an oracle does not exist, but one can – quite arbitrarily and without any theoretical justification whatsoever – replace it with a simple, known function such as SHA. In practice, one RSA standard does something like this.

A New Protocol

An arbitrary semantically secure public key encryption primitive can be used (with some care) in place of DDH to create a secure session key exchange protocol similar to Protocol 2. We will now discuss a different way using a public key encryption primitive to construct a secure session key exchange protocol. In this way, the keys of the primitive will form part of the PKI, and we will require strong (CCA) security.

On input 1^n, *Gen* will randomly generate a key pair $(spub, spri)$ for a secure signature scheme, and a pair $(epub, epri)$ for a strongly secure public key encryption primitive.
The public key for the protocol will be $(spub, epub)$, and the private key will be $(spri, epri)$.
We assume signatures are of length n. We write $SIGN_B(m)$ to mean the signature of m using key $spri_B$.
We assume the encryption primitive encrypts strings of length $2n$, and that encryptions are of length $\ell'(n)$. We write $ENC_A(m)$ to mean a probabilistic encryption of m using $epub_A$.

Protocol 3: We present the role-0 algorithm and the role-1 algorithm separately.
The security parameter is 1^n.
Process $\langle A, B, 0\rangle$ works as follows:

- Choose a random n-bit string r and send it on the output channel.
- (To understand this part, see the second part of the role-1 process protocol below.)
  Receive string α of length $\ell'(n)$ followed by string σ of length n, on the read channel.
  Use $spub_B$ to check that σ is a signature by B of $[\alpha\, r\, A]$, and if not, abort with output FAIL.
  Use $epri_A$ to decrypt α; if this is FAIL then abort with output FAIL;
  if the second half of the decryption is not B then abort with output FAIL,
  else let K be the first half of the decryption.
- Use K as the session key.

Process $\langle B, A, 1\rangle$ works as follows:

- Receive an n-bit string r on the input channel.

- Choose a random n-bit string K.
  Use $epub_A$ and random bits * to compute $\alpha \leftarrow ENC_A(K\,B, *)$.
  Use $spri_B$ to compute $\sigma \leftarrow SIGN_B(\alpha\, r\, A)$.
  Send string α followed by string σ on the write channel.

- Use K as the session key.

Before arguing that Protocol 3 is secure, let us see why the protocol would be *insecure* if we attempted to simplify it in certain ways.

For example, let's say that in the second step of B's role-1 protocol, instead of $\alpha \leftarrow ENC_A(K\,B, *)$, we had $\alpha \leftarrow ENC_A(K, *)$. (Assume that we modify the checking in the second step of A's role-0 protocol appropriately.) ADV could then do the following.

- ADV creates good guys A and B and bad guy U, and generates keys for U in the correct way using GEN.

- ADV creates processes $\langle A, U, 0\rangle$ and $\langle B, A, 1\rangle$.

- ADV reads n-bit r from $\langle A, U, 0\rangle$, and sends it to $\langle B, A, 1\rangle$.

- ADV reads $\ell'(n)$-bit α followed by n-bit σ from $\langle B, A, 1\rangle$.

- ADV sends $[\alpha, SIGN_U(\alpha, r, A)]$ to $\langle A, U, 0\rangle$.

- ADV challenges $\langle B, A, 1\rangle$, receiving challenge CH.

- ADV opens the key K output by $\langle A, U, 0\rangle$.

- ADV outputs 0 if $CH = K$, and 1 otherwise.

$\langle B, A, 1\rangle$ and $\langle A, U, 0\rangle$ are non-matching processes who will output identical keys. So ADV will succeed with probability nearly 1.

In fact, this example shows why strong security is needed in Protocol 3 for our public-key encryption primitive. If our primitive only satisfied semantic security, for example, then it may be "malleable". In fact, it may be possible to convert a value for $ENC_A(K\,B, *)$ to a value for $ENC_A(K\,U, *)$, in which case we would have the same insecurity problem as above. To see how a semantically secure primitive could have this bad feature, imagine that it encrypted a string m by independently encrypting each of the bits one at a time. It would then be trivial to convert a value for $ENC_A(K\,B, *)$ to a value for $ENC_A(K\,U, *)$ without knowing K.

Theorem: Protocol 3 is secure.

We now outline a proof of this theorem.

Consider an adversary ADV that breaks the protocol.
As in Notes #9, we can assume that ADV never successfully forges a message signed by a good-guy (otherwise we could break the signature scheme).

We now describe ADV$'$ who will break the given public key encryption primitive.
So assume security parameter 1^n, and say that GEN has chosen keys pub' and pri', we are given the public key pub', as well as access to $DEC_{pri'}$. (Let's think of ourselves as ADV$'$.)

As in the proof for Protocol 2, we assume that we correctly guess the process $\langle A, C, b\rangle^i$ that ADV will challenge; A and C are good-guy names. (Of course, we will actually guess the indices of the names.)

Because the protocol is not symmetric between role 0 and role 1, we will consider two cases.

Case 0: $b = 0$. So the challenged process is $\langle A, C, 0\rangle^i$.

Proof for Case 0:
Process $\langle A, C, 0\rangle^i$ will receive a message signed by C, the last n bits of which are A. Since ADV is not forging messages, this must have been signed by a C process; looking at the protocol, we see it must have been signed by a process of type $\langle C, A, 1\rangle$. As above, say that we successfully guess the process $\langle C, A, 1\rangle^j$ that will sign this message.

So we are assuming that ADV will challenge process $\langle A, C, 0\rangle^i$, sending it the message signed by $\langle C, A, 1\rangle^j$.

ADV$'$ simulates ADV choosing the good-guy names as follows. (We have looked ahead a bit because we want to treat the name A specially when setting up its keys.)
ADV$'$ will choose the *spub* and *spri* keys for all of the good-guys, correctly according to the signature scheme.
ADV$'$ will choose the *epub* and *epri* keys for all the good-guys – *except* A – correctly according to the encryption primitive.
For A, ADV$'$ will choose $epub_A$ to be pub', the public key for the instance of the encryption primitive ADV$'$ is trying to break. ADV$'$ will think of $epri_A$ as being pri'.
ADV$'$ simulates giving all of these public keys to ADV.
ADV$'$ simulates ADV choosing the bad-guy names and their public keys in whatever way ADV wishes to.

ADV$'$ simulates ADV creating and interacting with processes, as follows.
For process $\langle A, C, 0\rangle^i$, ADV$'$ chooses a random n-bit r and (when ADV asks for it) simulates sending it to ADV.
We know that $\langle A, C, 0\rangle^i$ will receive a message signed by $\langle C, A, 1\rangle^j$, and for A to not FAIL, this message must be of the form $[\alpha\, r\, A]$ for some α. Examining the protocol, we see therefore that ADV must give $\langle C, A, 1\rangle^j$ the string r as the first thing $\langle C, A, 1\rangle^j$ reads, and $\langle C, A, 1\rangle^j$ sends α followed by the signature.

Now $epri_A = pri'$ and $epub_A = pub'$. We want to choose α in such a way that that ADV$'$ can use ADV to break the encryption primitive.

So ADV$'$ chooses two random n-bit strings k^0 and k^1 and lets $M^0 = [k^0\, C]$ and $M^1 = [k^1\, C]$. A random bit *bit* is chosen (but ADV$'$ doesn't see it), and ADV$'$ receives an encryption of M^{bit} as the challenge CH'. ADV$'$ wants to find the value of *bit*, and is allowed to query the oracle $\mathrm{DEC}_{\mathrm{pri}'}$.

So when ADV reads from $\langle C, A, 1\rangle^j$, ADV$'$ simulates ADV receiving $[CH', SIGN_C(CH'\, r\, A)]$. Later, ADV will send to $\langle A, C, 0\rangle^i$ the string $[CH', \sigma]$ where σ verifies as a signature by C of $[CH'\, r\, A]$.

$\langle C, A, 1\rangle^j$ will now generate the same session key as the matching process $\langle A, C, 0\rangle^j$. This session key must be k^{bit}.

When ADV challenges the session key of $\langle A, C, 0\rangle^i$, ADV$'$ will simulate him receiving $CH = k^0$. Observe that if $bit = 0$, then both ADV and ADV$'$ want to output 0. If bit $= 1$, then ADV$'$ wants to output 1; but ADV also wants to output 1 since k^0 is a random string independent of the challenged session key k^1.

It remains to show how we can simulate the rest of ADV perfectly. The main issue is showing how to simulate ADV getting information from the Oracle and how to simulate ADV opening the session keys of various processes.

First, consider two matching (good-guy) processes, say $\langle D, E, 0\rangle^1$ and $\langle E, D, 1\rangle^2$ that have successfully completed; under what conditions will they output the same session key? Since ADV cannot forge signatures, the $\langle D, E, 0\rangle^1$ process must have received a signed encryption of a key from some $\langle E, D, 1\rangle$ process; if it was from the $\langle E, D, 1\rangle^2$ process then the two processes will output the same session key; if it was from a different $\langle E, D, 1\rangle$ process, then (since each role 1 process chooses a random key) the session keys will almost certainly be different.

We can therefore simulate the Oracle answers that ADV receives. How can we simulate ADV receiving the session key of a process? This is trivial for every $\langle E, M, 1\rangle$ process (other than $\langle C, A, 1\rangle^j$), since we (in doing the simulation) choose the session key ourselves. It is also trivial for every $\langle D, M, 0\rangle$ process where D is not A, since we know $epri_D$.

Lastly, we have to explain how ADV$'$ can open the session key (or detect failure) of a role 0 A process, say $\langle A, M, 0\rangle^1$, that is not $\langle A, C, 0\rangle^i$. Assuming the M-signature verifies, we have to be able to decrypt the encryption α that the $\langle A, M, 0\rangle^1$ process receives, even though we do not know $epri_A = pri'$. Recall however, that we are allowed to use $DEC_{pri'}$ as long as we don't invoke it on CH'. So assume that ADV causes α to be CH'. We know that the second half of the decryption of CH' is C, so $\langle A, M, 0\rangle^1$ will FAIL unless $M = C$. So assume that process $\langle A, C, 0\rangle^1$ receives encryption CH'. Since $\langle A, C, 0\rangle^1$ initially sends out a different random string than $\langle A, C, 0\rangle^i$, the signature that $\langle A, C, 0\rangle^1$ receives must come from a different $\langle C, A, 1\rangle$ process than $\langle C, A, 1\rangle^j$, and therefore the encrypted string must (almost certainly) be different from CH'.

If ADV$'$ can correctly simulate ADV, ADV$'$ will be correct with the same probability as ADV (assuming we have guessed the two processes correctly). (We may be off by $n^d/2^n$, the probability that two of the randomly chosen n bit strings are the same.)

Case 1: $b = 1$.

Proof for Case 1: This is left as an exercise. $\square$

The reader should note that there are many variants of Protocol 3 that are also secure, and many that are not, and it is hard to have an intuition about this. Also, Protocol 3 clearly does *not* satisfy "forward security", as (vaguely) defined in Notes #9. This is because someone who recorded the key exchange and later learned $epri_A$, could then learn the session key.