Lecture 34

1 Public-Key Encryption From General Assumptions

Thus far, the security of all the public-key encryption schemes we have seen (i.e., the Goldwasser-Micali encryption scheme based on quadratic residuosity and the El Gamal encryption scheme based on the DDH assumption) are based on *specific* number-theoretic assumptions. In this lecture, we define the more general notion of a trapdoor permutation, give some examples of (assumed) trapdoor permutations, and show how to construct a secure public-key encryption scheme from any trapdoor permutation.

Why do we care about a result of this sort? From a theoretical point of view, it is interesting to characterize public-key encryption based on what primitives are necessary in order to construct provably-secure schemes. This can also lead to greater understanding of public-key encryption, which may yield other benefits. From a more practical point of view, a generic result of the form "trapdoor permutations imply the existence of public-key encryption schemes" allows us to instantiate the trapdoor permutation in a number of different ways. In other words, when someone develops a new (candidate) trapdoor permutation, we do not have to come up with (and prove secure) a new construction of a public-key encryption scheme; instead, we simply use the generic result to immediately give a scheme that we then know is secure. Note that basing results on as general an assumption as possible also protects us in case, for example, factoring turns out to be easy!

1.1 Trapdoor Permutations

Informally, recall that f is a one-way permutation if the following hold:

- f is a permutation.
- f is efficiently computable.
- f is hard to invert (on a randomly-chosen input).

A *trapdoor* permutation g will be defined similarly with one exception: although g will be hard to invert *in general*, there is some *trapdoor information* which enables efficient inversion of g. Because of the availability of trapdoor information, our treatment of trapdoor permutations will be slightly different from our treatment of one-way permutations, as we will see in the following definition.

Definition 1 *A* (t, ϵ)-trapdoor permutation *is a triple of poly-time algorithms* $(\mathcal{K}, f, \mathsf{Inv})$ *such that:*

- *$\mathcal{K}$ is a randomized key generation algorithm which outputs a pair $(\mathsf{k}, \mathsf{td})$ where td is called a* trapdoor.

- *f is a deterministic algorithm which takes as input a key k and an input $x \in \mathcal{D}_{\mathsf{k}}$ (we refer to $\mathcal{D}_{\mathsf{k}}$ as the* domain of key k*); it returns an output $y = f(\mathsf{k}, x) \in \mathcal{D}_{\mathsf{k}}$. (We denote $f(k, \cdot)$ by $f_{\mathsf{k}}(\cdot)$.)*

- Inv *takes as input the trapdoor* td *and an input y and returns a value x.*

Furthermore, $(\mathcal{K}, f, \mathsf{Inv})$ satisfies the following:

- *For all $(\mathsf{k}, \mathsf{td})$ output by $\mathcal{K}$ and all $x \in \mathcal{D}_{\mathsf{k}}$ we have: $\mathsf{Inv}(\mathsf{td}, f_{\mathsf{k}}(x)) = x$. (For this reason, we simply denote $\mathsf{Inv}(\mathsf{td}, \cdot)$ by $f_{\mathsf{k}}^{-1}(\cdot)$ but it is important to remember that efficiently computing f_{k}^{-1} is only possible with td.) In particular, this implies that for all k output by $\mathcal{K}$, f_{k} is a permutation on $\mathcal{D}_{\mathsf{k}}$.*

- *f is efficiently computable. More formally, given any k output by $\mathcal{K}$ and any $x \in \mathcal{D}_{\mathsf{k}}$, it is possible to efficiently compute $f_{\mathsf{k}}(x)$.*

- *f is hard to invert* without the trapdoor *(even when given k). Namely, for all algorithms A running in time t we have:*

$$\Pr[(\mathsf{k}, \mathsf{td}) \leftarrow \mathcal{K}; x \leftarrow \mathcal{D}_{\mathsf{k}}; y = f_{\mathsf{k}}(x) : A(\mathsf{k}, y) = x] \leq \epsilon.$$

  (Again, with the trapdoor *f_{k} is easy to invert [in other words, f_{k}^{-1} is easy to compute] using algorithm* Inv*.)*

It is worth noting that every trapdoor permutation is also a one-way permutation. We give a few examples of trapdoor permutations that we have already seen.

EXAMPLE 1: RSA We have discussed the RSA permutation before, but we now explicitly present it in a form compatible with the above definition. Let $\mathcal{K}$ be an algorithm that chooses a random modulus N (which is the product of two primes) and also chooses e, d such that $ed = 1 \bmod \varphi(N)$. The key k consists of (N, e) while the trapdoor td is simply (N, d). We have $\mathcal{D}_{\mathsf{k}} = \mathcal{D}_{N,e} = \mathbb{Z}_N^*$ and

- $f_{\mathsf{k}}(x) = f_{N,e}(x) = x^e \bmod N$.
- $\mathsf{Inv}(\mathsf{td}, y) = \mathsf{Inv}((N, d), y) = y^d \bmod N = f_{N,e}^{-1}(y)$.

EXAMPLE 2: SQUARING We also show how the Rabin squaring permutation satisfies Definition 1. Let $N = pq$ where p, q are prime and $p = q = 3 \bmod 4$. We noted in a previous lecture that the function $f_N(x) = x^2 \bmod N$ is a permutation over $\mathcal{QR}_N$. So, our key generation algorithm $\mathcal{K}$ will be an algorithm that chooses a modulus N which is a product of primes p, q with $p = q = 3 \bmod 4$. The key k is simply N, the trapdoor is the factorization (p, q) of N, and $\mathcal{D}_{\mathsf{k}} = \mathcal{D}_N = \mathcal{QR}_N$. We have already defined our function $f_N(\cdot)$; we also note that inversion can be done efficiently (i.e., square roots can be efficiently computed modulo N) when the factorization of N is known.

1.2 Public-Key Encryption from Trapdoor Permutations

It may seem "obvious" that trapdoor permutations and public-key encryption schemes are essentially equivalent. *This intuition is* **wrong**! For one thing, *not every public-key encryption scheme is based on a trapdoor permutation.* In fact, neither of the schemes we have seen so far are based on trapdoor permutations: the hardness of deciding quadratic residuosity clearly does not give a trapdoor permutation directly[1] and the DDH assumption is also unrelated to trapdoor permutations. In fact, it is believed that public-key encryption is *weaker* than trapdoor permutations, in the sense that the former may exist even though the latter do not.

Secondly, it is not trivial to construct a *provably-secure* public-key encryption scheme from any trapdoor permutation. The "obvious" way of doing so, in which message m is encrypted by computing $f_{\mathsf{k}}(m)$ (and decryption of C is done by having the receiver — who has the trapdoor — compute $f^{-1}(C)$) is completely insecure: this is a deterministic scheme, so cannot possibly be secure as a public-key encryption scheme.

Before we give a secure construction of a public-key encryption scheme from any trapdoor permutation, we recall the notion of a *hard-core bit* that we saw previously in the context of one-way permutations. Let f be a one-way permutation over some domain $\mathcal{D}$. We say that $h : \mathcal{D} \to \{0,1\}$ is a hard-core bit for f if (informally) it is "hard" to predict the bit $h(x)$ (with probability much better than 1/2) given only $f(x)$; more formally:

Definition 2 *Let f be a permutation over some domain $\mathcal{D}$. We say that $h : \mathcal{D} \to \{0,1\}$ is a (t, ϵ)-*hard-core bit for* f *if, for all algorithms A running in time t we have:*

$$|\Pr[x \leftarrow \mathcal{D}; y = f(x) : A(y) = h(x)] - 1/2| \leq \epsilon.$$

We may give an entirely analogous definition for a hard-core bit of a *trapdoor* permutation. We do not give the formal definition here, but informally we say that keyed function h (with $h_{\mathsf{k}} : \mathcal{D}_{\mathsf{k}} \to \{0,1\}$) is a hard-core bit for trapdoor permutation f if it is hard to compute $h_{\mathsf{k}}(x)$ (with probability much better than 1/2) given $f_{\mathsf{k}}(x)$ and k, but without the associated trapdoor td for k.

We saw in a previous lecture that for every one-way permutation f, there is a hard-core bit h for f. The same result holds for trapdoor permutations. We now show how to construct a public-key encryption scheme from any trapdoor permutation. Let $(\mathcal{K}, f, \mathsf{Inv})$ be a trapdoor permutation, and let h be a keyed function which is a hard-core bit for f. The encryption scheme is defined as follows:

- The key generation algorithm runs $\mathcal{K}$ to obtain a pair $(\mathsf{k}, \mathsf{td})$. The public key consists of k and (a description of) h; the secret key is the trapdoor td.
- To encrypt a single bit b using public key $pk = (\mathsf{k}, h)$, choose a random $r \in \mathcal{D}_{\mathsf{k}}$ and send $\langle f_{\mathsf{k}}(r), h_{\mathsf{k}}(r) \oplus b \rangle$.
- To decrypt ciphertext $\langle C_1, C_2 \rangle$ using the secret key, compute $b = C_2 \oplus h_{\mathsf{k}}(f_{\mathsf{k}}^{-1}(C_1))$. (Note that this can be done efficiently since the secret key contains td.)

[1] However, the quadratic residuosity assumption does imply that factoring is hard, which gives rise to the Rabin trapdoor permutation described above. This is more of an "accident" than anything else.

It is not hard to see that the encryption scheme always gives correct decryption.

Theorem 1 *If h is a hard-core bit for trapdoor permutation $(\mathcal{K}, f, \mathsf{Inv})$ then the above is a secure public-key encryption scheme.*

Although we do not give a full proof of this theorem here, we hope the reader will be convinced by the following informal argument. Recall in the case of a one-way permutation f with hard-core bit h that the function $G(r) \stackrel{\text{def}}{=} f(r) \circ h(r)$ is a pseudorandom generator. This means that for a random r, the string $f(r) \circ h(r)$ "looks random" (in a way made formal by the definition of pseudorandomness). A completely analogous result holds when f is a trapdoor permutation. So, a "secure" way to encrypt a message m would be to compute $m \oplus (f_{\mathsf{k}}(r) \circ h(r))$. While this works for private-key encryption when r is pre-shared (indeed, this was one of the first secure schemes we showed), this does not work for the case of public-key encryption, where the sender and receiver do not pre-share r. How to we fix this?

In effect, we allow the receiver to compute r by sending $f(r)$ alone (i.e., without xor'ing $f(r)$ with any part of the message). The sender can now xor the 1-bit message with the final bit $h(r)$ of the "pseudorandom string" $f(r), h(r)$. In the case of pseudorandomness, we saw that having $f(r)$ did not allow an adversary to predict $h(r)$ (with probability better than 1/2). So too, here, an adversary who sees $f_{\mathsf{k}}(r)$ cannot predict $h_{\mathsf{k}}(r)$ (with probability better than 1/2). So $h_{\mathsf{k}}(r)$ can be used as a "one-time pad" to encrypt the 1-bit message.

1.3 Encrypting Longer Messages

We have already seen in previous lectures that a secure public-key encryption scheme for 1-bit messages allows encryption of arbitrary-length messages as well (by concatenating encryptions of each bit of the message). But this is going to be inefficient, both in communication (there will be $k+1$ bits of ciphertext per bit of the message, where k is the input/output length of f_{k}) and computation (one evaluation of f_{k} is required for each bit of the message). The hybrid approach we saw in a previous lecture will help, but only if we first use the public key scheme to encrypt sufficiently-many bits (i.e., enough to use as a key for a private-key scheme).

Hopefully, the discussion of the previous section (comparing the encryption scheme to the construction of a PRG) motivates the following, more efficient construction. In the case of PRGs, we saw that the construction $G(r) = f^{\ell}(r) \circ h(f^{\ell-1}(r)) \circ \cdots \circ h(r)$ is a PRG stretching its input by ℓ bits. We can use the same technique for public-key encryption! Namely, to encrypt an ℓ-bit message $b_1 b_2 \cdots b_\ell$, we pick a random r, compute $C_1 = f_{\mathsf{k}}(r)$, and

$$C_2 = (h_{\mathsf{k}}(f\mathsf{k}^{\ell-1}(r)) \circ h_{\mathsf{k}}(f_{\mathsf{k}}^{\ell-2}(r)) \circ \cdots \circ h_{\mathsf{k}}(r)) \oplus (b_1 \cdots b_\ell).$$

The transmitted ciphertext is C_1, C_2. (Key generation is done as before, and decryption should be obvious.)

How efficient is this scheme? Unfortunately, the computation required is still one evaluation of f_{k} per bit of the message, but the ciphertext length is now $k + \ell$ for an ℓ-bit message (in the original scheme, the ciphertext would have been $(k+1)\ell$ bits long for an ℓ-bit message).