Last Revised: 2/19/96.

ECS 227: Modern Cryptography
Lecture 10

Asymmetric Encryption

Phillip Rogaway
13 February 1996

In the last lecture we introduced, informally, two different notions: a *trapdoor permutation* (our example was the RSA function), and a *public key encryption scheme* (our example was based on the RSA function). Today we develop those notions more precisely.

8.3 Trapdoor permutations

Informally, a trapdoor permutation makes it easily compute $f(x)$ from x, but makes it hard to compute x from $f(x)$, *unless* you know some secret, f^{-1}. If you do know this secret, then it is easy to compute x from $f(x)$.

As usual, it is convenient to separate the syntax and the security goal in defining this idea.

Definition 8.3.1 *A* trapdoor permutation generator, $\mathcal{F}$, *is a PPT function which maps a string* $1^k \in 1^*$ *to a pair of strings* (f, f^{-1}), *where* f *and* f^{-1} *specify functions* $f, f^{-1} : \{0,1\}^k \to \{0,1\}^k$ *which are inverses of one another. Each must be computable in time* $\mathrm{poly}(k)$, *where* $\mathrm{poly}(\cdot)$ *is some fixed polynomial associated to* $\mathcal{F}$.

Putting RSA into the above framework, the trapdoor permutation generator $\mathcal{RSA}$ behaves as follows: on input 1^k,

> Choose random distinct $k/2$-bit primes, p and q.
> Let $N = pq$.
> Chooses a random $e \in \mathsf{Z}^*_{\varphi(N)}$.
> Compute $d \in \mathsf{Z}^*_{\varphi(N)}$ such that $ed \equiv 1 \bmod \varphi(N)$.
> Return (N, e), (N, d), the first of these encoding a function $f(x) = x^e \bmod N$, and the second encoding a function $f^{-1}(x) = x^d \bmod n$.

Our definition of security speaks to the difficulty of computing $f^{-1}(x)$ from x:

Definition 8.3.2 *An adversary I is said to (t, ϵ)-*invert *the trapdoor permutation generator $\mathcal{F}(1^k)$ if*

$$\begin{aligned} \mathrm{Adv}_I &\stackrel{\text{def}}{=} \Pr\left[(f, f^{-1}) \leftarrow \mathcal{F}(1^k);\ x \leftarrow \{0,1\}^k;\ y \leftarrow f(x) :\ I(1^k, f, y) = x\right] \\ &\geq \epsilon \end{aligned}$$

and $I(1^k, \cdots)$ runs in at most t time.

There is a corresponding asymptotic notion, in which we say that $\mathcal{F}$ is a secure trapdoor permutation if, for ever PPT I, $\mathrm{Adv}_I(k)$ is negligible. Usually the word "secure" is omitted.

Now our astute class has immediately noticed my lie in calling $\mathcal{RSA}$ a trapdoor permutation generator: with a trapdoor permutation f and f^{-1} were supposed to have domain of $\{0,1\}^k$, but with $\mathcal{RSA}$ their domain was the proper subset of this Z_N^*. We prefer to ignore this technicality, for whenever you want to use a trapdoor permutation you will find it possible to work around the inconvenience of an "incomplete' domain. Let us briefly elaborate.

There are two different "defects" in the domain of each function f returned by $\mathcal{RSA}$:

- First, f is missing a whole bunch of consecutive points— all the $x \in \{0,1\}^k - \{N, N+1, N+2, \cdots, 2^k - 1\}$ are outside the domain. This set of missing points could amount to almost half of $\{0,1\}^k$.

- Second, f is missing a spattering of points in $\{0, 1, \ldots, N-1\}$. In particular, all of those points which happen to be multiples of p or q are outside the domain of f.

In typical usages of RSA, the first issue is usually dealt with by forcing our algorithms to use make use of only a certain prefix of $\{0,1\}^k$, such as $\{0,1\}^{k-1}$ or $\{0,1\}^{k-8}$. The second issue is most often ignored: after all, there are only a fraction $(N - \varphi(N))/N \approx 2^{-k/2}$ of the points in $[0..N-1]$ are "missing," and so, with typical values of k, random points in $[0..N-1]$ are, for all practical purposes, always going to wind up in Z_N^*.

8.4 A definition for secure encryption

Definitions for asymmetric encryption are similar to their symmetric counterparts. We can adapt either our notion of security in the sense of indistinguishability, or our notion of semantic security. Let us do the former, beginning, as we often do, by defining the syntax.

Definition 8.4.1 *A (probabilistic, asymmetric)* encryption scheme, *$\mathcal{G}$, is a PPT algorithm which maps $1^k \in 1^*$ to a pair of strings $(\mathcal{E}, \mathcal{D})$, where*

$$\begin{aligned} \mathcal{E} &: \mathcal{M} \times \{0,1\}^\infty \to \{0,1\}^*, \quad \textit{and} \\ \mathcal{D} &: \{0,1\}^* \to \mathcal{M} \cup \{*\}\,. \end{aligned}$$

We require that $\mathcal{D}(\mathcal{E}(x,r)) = x$ for all $x \in \mathcal{M}$ and $r \in \{0,1\}^\infty$. $\mathcal{E}$ and $\mathcal{D}$ should be poly*-time computable, for some particular polynomial* $\mathrm{poly}(\cdot)$ *associated to the encryption scheme. We assume that, if $x \in \mathcal{M}$ then $x' \in \mathcal{M}$ for all x' such that $|x'| = |x|$.*

Notes — revise to make symmetric/asymmetric encryption and digital signatures/MACs all syntactically analogous– e.g., the above becomes a triple $(\mathcal{E}, \mathcal{D}, \mathcal{K})$, where $\mathcal{K}$ outputs *pk* and *sk*.

Now recall our notion of a symmetric encryption scheme which is secure in the sense of indistinguishability. To define security we gave an adversary A access to an oracle, which was either a $\mathcal{E}_a(\cdot)$ oracle or a $\mathcal{E}_a(\$^{|\cdot|})$ oracle. The oracle was fulfilling two different purposes: first, it was modeling the possibility that A might launch a chosen-plaintext attack; second, it provided the means to test how well the adversary was doing.

For asymmetric encryption, we need to give the adversary the public key encryption algorithm, $\mathcal{E}$. This is given to her as a string, not an oracle. Having this, the adversary has no need for an *oracle* to allow the adversary to encrypt strings of her choice: the adversary can compute desired ⟨plaintext, ciphertext⟩ pairs on her own. In other words, a known-plaintext attack is intrinsically within the adversary's capabilities in asymmetric encryption. We can retain an oracle in our formulation of security as a convenient way to test how well our adversary is doing.

Definition 8.4.2 *Let $\mathcal{G}$ be an (probabilistic, asymmetric) encryption scheme and let A be an adversary. Then A is said to (t, q, μ, ϵ)-break $\mathcal{G}(1^k)$ if*

$$\begin{aligned} Adv_A(k) &\stackrel{\text{def}}{=} \Pr\left[(\mathcal{E}, \mathcal{D}) \leftarrow \mathcal{G}(1^k);\ A^{\mathcal{E}(\cdot)}(1^k, \mathcal{E}) = 1\right] - \\ &\qquad \Pr\left[(\mathcal{E}, \mathcal{D}) \leftarrow \mathcal{G}(1^k);\ A^{\mathcal{E}(\$^{|\cdot|})}(1^k, \mathcal{E}) = 1\right] \\ &\geq \epsilon \end{aligned}$$

and $A(1^k, \cdots)$ runs in at most t steps, asks its oracle at most q queries, and these queries total at most μ bits.

Definition 8.4.3 *(Probabilistic, asymmetric) encryption scheme $\mathcal{G}$ is (t, q, μ, ϵ)-secure if there does not exist an adversary A who (t, q, μ, ϵ)-breaks it.*

There is, of course, the corresponding asymptotic notion, which simply demands that $\mathrm{Adv}_A(k)$ be negligible when $t(k)$ is polynomial.

8.5 Equivalent definitions

One can adapt our notion of semantic security in the symmetric setting to come up with a corresponding notion of semantic security for asymmetric encryption. We will omit doing this, as well as proving that the adapted notion turns out to be equivalent (apart from a polynomial loss in security) to what we've defined above.

For these notes, I will repeat a sidebar we went through in class to improve our skills in defining things.

We imagine an adversary running in two stages. In the first stage she tries to find two equal-length messages which she can do a good job distinguishing encryptions of. In the second phase she gets the encryption of a random one of these messages. She must guess which of the two messages it corresponds to. Successful adversaries are those which can guess well. Here is the definition.

Definition 8.5.1 *Let $\mathcal{G}$ be an (probabilistic, asymmetric) encryption scheme and let A be an adversary. Then A is said to (t, ϵ)-break $\mathcal{G}(1^k)$, in the message-finder sense, if*

$$\begin{aligned} Adv_A(k) &\stackrel{\text{def}}{=} 2 \cdot \Pr[(\mathcal{E}, \mathcal{D}) \leftarrow \mathcal{G}(1^k);\ (x_0, x_1, c) \leftarrow A(1^k, \mathsf{find}, \mathcal{E}); \\ &\qquad b \leftarrow \{0,1\};\ y \leftarrow \mathcal{E}(x_b) :\ A(1^k, \mathsf{guess}, y) = b] - 1 \\ &\geq \epsilon \end{aligned}$$

and the total running time of $A(1^k, \mathsf{find}, \cdots)$ and $A(1^k, \mathsf{guess}, \cdots)$ is at most t. We demand that, for any (x_0, x_1, c) which $\mathcal{E}(1^k, \mathsf{find}, \cdots)$ might output, $|x_0| = |x_1|$.

What is the purpose of c in the formulation above? It is whatever "saved state" —coin flips, x_0, x_1, etc.— which the adversary running in her "message finder stage" might like to pass on to herself, in her "guess b stage."

In class and in the notes we did not argue the equivalence (apart from polynomial losses in security) of the above definition to our previous one. Try to figure this out on your own.

8.6 Achieving asymmetric encryption

How are we going to achieve our goal of asymmetric encryption? Now that we have a definition in place, it is obvious that no deterministic scheme will do: if $\mathcal{E}$ is deterministic, an adversary will have no trouble at all distinguishing an oracle for $\mathcal{E}(\cdot)$ from an oracle for $\mathcal{E}(\$^{|\cdot|})$. What can we do?

To simplify matters, let us try to transmit a single bit. This isn't such an over-simplification, in the sense that if we can encrypt a single bit securely, then we ought to be able to encrypt arbitrary strings securely: just encrypt the long string bit-by-bit. Of course this isn't likely to be a route leading to much efficiency— but we shall address efficiency later on.

First, let us think specifically for RSA. What might we do to securely transmit a bit with RSA?

Here's a reasonable-looking strategy. We start off by using the $\mathcal{RSA}$ generator to come up with a public key (N, e) and a secret key $(N.d)$. These will stay with us for a long time— we don't have to choose a new public key with each bit. Now to encrypt a bit b, do the following:

> Choose a random $x \in \mathsf{Z}_N^*$.
> Let $\mathrm{lsb}(x)$ denote the least significant bit of x.
> Encrypt b as $\langle x^e \bmod N,\ b \oplus \mathrm{lsb}(x)\rangle$.

Does the above strategy work? Its working means that, if $\mathcal{RSA}$ is a secure trapdoor permutation, then it should be possible to show that the above algorithm is a secure encryption scheme.

I claim that it is not at all clear if the above method works. For if we assume that $\mathcal{RSA}$ is a trapdoor permutation then what we know is that it is computationally infeasible to find (all of) x from $x^e \bmod N$, when x is chosen at random from Z_N^* and (N, e) are chosen according to $\mathcal{RSA}$. But this assumption says nothing about how easy or hard it might be to compute a single bit of x, $\mathrm{lsb}(x)$, from $x^e \bmod N$. Finding $\mathrm{lsb}(x)$ might be trivial, even though it is hard to compute all of x.

In fact, this is not the case. A nice result of Chor and Goldreich says that, if an adversary could do a good job of predicting $\mathrm{lsb}(x)$ from $x^e \bmod N$ then, in fact, another adversary could do a good job computing all of x from $x^e \bmod N$.

It is clear that there is something special about the lsb with respect to RSA— it has "concentrated" the hardness of inverting RSA into a single bit. If we had chosen the most significant bit, say, then our scheme might not have worked, because the lsb "works" for RSA, but the msb might not. We say that the function $B(x) = \mathrm{lsb}(x)$ is a *hard-core bit* of $\mathcal{RSA}$.

We want to show that if a trapdoor permutation exists then it is possible to achieve our notion of secure asymmetric encryption. Our strategy is to show that, if a trapdoor permutation exists, then there is a trapdoor permutation which has some particular function, B, as its hard-core bit. Then the strategy to encrypt a bit b becomes the following. Use the trapdoor permutation generator to generate an $(f, f^{-1}) \in \mathcal{F}(1^k)$. Let f be the public key; let f^{-1} be the secret key. To encrypt the bit b:

> Choose a random $x \in \{0,1\}^k$.
> Encrypt b as $\langle x^e \bmod N,\ b \oplus B(x)$.

To encrypt a sequence of bits $b_1 b_2 \cdots$, separately encrypt each bit b_i.

Our next lecture will define a hard-core bit of a one-way function f, and then we'll prove that a certain function B is hard-core bit for a one-way function f_2, where f_2 is obtained by a certain generic modification to an arbitrary one-way function f_1. While our intended application entails f_2 being a trapdoor permutation, this extra structure is irrelevant to making our hard-core bit. That is why we simplify and explain the hard-core bit construction simply assuming that f_1 is a one-way function.