# Lecture 11: Concrete candidates for public key crypto

Boaz Barak

In the previous lecture we talked about *public key cryptography* and saw the Diffie Hellman system and the DSA signature scheme. In this lecture, we will see the RSA trapdoor function and how to use it for both encryptions and signatures.

## Some number theory.

(See Shoup's excellent and freely available book for extensive coverage of these and many other topics.)

For every number $m$, we define $\mathbb{Z}_m$ to be the set $\{0,\ldots,m-1\}$ with the addition and multiplication operations modulo $m$. When two elements are in $\mathbb{Z}_n$ then we will always assume that all operations are done modulo $m$ unless stated otherwise. We let $\mathbb{Z}^*_n = \{a \in \mathbb{Z}_n : gcd(a,m) = 1\}$. Note that $n$ is prime if and only if $|\mathbb{Z}^*_m| = m-1$. For every $a \in \mathbb{Z}^*_m$ we can find using the extended gcd algorithm an element $b$ (typically denoted as $a^{-1}$) such that $ab = 1$ (can you see why?). The set $\mathbb{Z}^*_m$ is an abelian group with the multiplication operation, and hence by the observations of the previous lecture, $a^{|\mathbb{Z}^*_m|} = 1$ for every $a \in \mathbb{Z}^*_m$. In the case that $m$ is prime, this result is known as "Fermat's Little Theorem" and is typically stated as $a^{p-1} = 1 \pmod p$ for every $a \neq 0$.

> **Note on $n$ bits vs a number $n$:** One aspect that is often confusing in number-theoretic based cryptography, is that one needs to always keep track whether we are talking about "big" numbers or "small" numbers. In many cases in crypto, we use $n$ to talk about our key size or security parameter, in which case we think of $n$ as a "small" number of size $100-1000$ or so. However, when we work with $\mathbb{Z}^*_m$ we often think of $m$ as a "big" number having about $100-1000$ *digits*; that is $m$ would be roughly $2^{100}---2^{1000}$ or so. I will try to reserve the notation $n$ for "small" numbers but may sometimes forget to do so, and other descriptions of RSA etc.. often use $n$ for "big" numbers. It is important that whenever you see a number $x$, you make sure you have a sense whether it is a "small" number (in which case $poly(x)$ time is considered efficient) or whether it is a "large"

number (in which case only $poly(log(x))$ time would be considered efficient).

One procedure we often need is to find a prime of $n$ bits. The typical way people do it is by choosing a random $n$-bit number $m$, and testing whether it is prime. This relies on the following two facts:

**Theorem 1:** The probability that a random $n$ bit number is prime is at least $\Omega(1/n)$. In fact, it is $(1 \pm o(1))\frac{1}{n \ln 2}$ by the famous *Prime Number Theorem* obtained in the 1890's by Hadamard and de la Vall'{e}e Poussin).

**Theorem 2:** There is an $poly(n)$-time algorithm to test whether a given $n$-bit number is prime or composite. This was first shown in 1970's by Solovay, Strassen, Miller and Rabin via a *probabilistic* algorithm (that can make a mistake with probability exponentially small in the number of coins it uses), and in a 2002 breakthrough Agrawal, Kayal, and Saxena gave a *deterministic* polynomial time algorithm for the same problem.

We now show "poor man's versions" of both theorems:

**Lemma 1:** The probability that a random $n$ bit number is prime is at least $\Omega(1/n)$.

**Proof:** Let $N = 2^n$. We need to show that the number of primes between 1 and $N$ is at least $\Omega(N/\log N)$. Consider the number $\binom{2N}{N} = \frac{2N!}{N!N!}$. By Stirling's formula we know that $\log\binom{2N}{N} = (1 - o(1))2N$ and in particular $N \le \log\binom{2N}{N} \le 2N$. Also, by the formula using factorials, all the prime factors of $\binom{2N}{N}$ are between 0 and $2N$, and each factor $P$ cannot appear more than $k = \lfloor\frac{\log 2N}{\log P}\rfloor$ times. Indeed, for every $N$, the number of times $P$ appears in the factorization of $N!$ is $\sum_i \lfloor\frac{N}{P^i}\rfloor$, since we get $\lfloor\frac{N}{P}\rfloor$ times a factor $P$ in the factorizations of $\{1, \ldots, N\}$, $\lfloor\frac{N}{P^2}\rfloor$ times a factor of the form $P^2$, etc... Thus the number of times $P$ appears in the factorization of $\binom{2N}{N} = \frac{(2N)!}{N!N!}$ is equal to $\sum_i \lfloor\frac{2N}{P^i}\rfloor - 2\lfloor\frac{N}{P^i}\rfloor$: a sum of at most $k$ elements (since $P^{k+1} > 2N$) each of which is either 0 or 1.

Thus, $\binom{2N}{N} \le \prod_{\substack{1 \le P \le 2N \\ P \text{ prime}}} P^{\lfloor\frac{\log 2N}{\log P}\rfloor}$. Taking logs we get that $N \le \log\binom{2N}{N} \le$
$\sum_{P \text{prime} \in [2n]} \lfloor\frac{\log 2N}{\log P}\rfloor \log P \le$

$\sum_{P \text{prime} \in [2n]} \log 2N$

establishing that the number of primes in $[1, N]$ is $\Omega(\frac{N}{\log N})$. QED

**Lemma 2:** There is a probabilistic polynomial time algorithm $A$ that on input a number $m$, if $m$ is prime $A$ outputs `YES` with probability 1 and if $A$ is not even a "pseudoprime" it outputs `NO` with probability at least 1/2. (The definition of "pseudo-prime" will be clarified in the proof below.)

**Proof:** The algorithm is very simple and is based on Fermat's Little Theorem:

on input $m$, pick a random $a \in \{2, \ldots, m-1\}$, and if $gcd(a,m) \neq 1$ or $a^{m-1} \neq 1 \pmod{m}$ return `NO` and otherwise return `YES`.

By Fermat's little theorem, the algorithm will always return `YES` on a prime $m$. We define a "pseudoprime" to be a non-prime number $m$ such that $a^{m-1} = 1 \pmod{m}$ for all $a$ such that $gcd(a,m) = 1$.
If $n$ is *not* a pseudoprime then the set $S = \{a \in \mathbb{Z}_m^* : a^{m-1} = 1\}$ is a strict subset of $\mathbb{Z}_m^*$. But it is easy to see that $S$ is a *group* and hence $|S|$ must divide $|Z_n^*|$ and hence in particular it must be the case that $|S| < |\mathbb{Z}_n^*|/2$ and so with probability at least $1/2$ the algorithm will output `NO`. QED

Lemma 2 on its own might not seem very meaningful since it's not clear how many pseudoprimes are there. However, it turns out these pseudoprimes, also known as "Carmichael numbers", are much less prevalent than the primes (there are about $N/2^{-\Theta(\log N/\log\log N)} \ll N/\log N$ of those between $q$ and $N$). Moreover, as mentioned above, there are better algorithms that succeed for *all* numbers.

In contrast to *testing* if a number is prime or composite, there is no known efficient algorithm to actually *find* the factorization of a composite number. As we mentioned, the best known algorithms run in time roughly $2^{\tilde{O}(n^{1/3})}$.

## Fields

If $p$ is a prime then $\mathbb{Z}_p$ is a *field* which means it is closed under addition and multiplication and has 0 and 1 elements. One property of a field is the following:

**Theorem:** If $f$ is a nonzero polynomial of degree $d$ over $\mathbb{Z}_p$ then there are at most $d$ distinct inputs $x$ such that $f(x) = 0$.

(If you're curious why, you can see that the task of, given $x_1, \ldots, x_{d+1}$ finding the coefficients for a polynomial vanishing on the $x_i$'s amounts to solving a linear system in $d+1$ variables with $d+1$ equations that are independent due to the non-singularity of the Vandermonde matrix.)

In particular every $x \in \mathbb{Z}_p$ has at most two *square roots*. In fact, just like over the reals, every $x \in \mathbb{Z}_p$ either has no square roots or exactly two square roots of the form $\pm s$.

We can efficiently find square roots modulo a prime. In fact, the following result is known:

**Theorem:** There is a probabilistic $poly(\log p, d)$ time algorithm to find the roots of a degree $d$ polynomial over $\mathbb{Z}_p$.

This is a special case of the problem of factoring polynomials over finite fields, shown in 1967 by Berlekamp and on which much other work has been done; see Chapter 20 in Shoup).

**Chinese remainder theorem**

Suppose that $m = pq$ is a product of two primes. In this case $Z^*_m$ does not contain_all_ the numbers from 1 to $m-1$. Indeed, all the numbers of the form $p, 2p, 3p, \ldots, (q-1)p$ and $q, 2q, \ldots, (p-1)q$ will have non-trivial g.c.d. with $m$. There are exactly $q-1+p-1$ such numbers (because $p$ and $q$ are prime all the numbers of the forms above are distinct). Hence $|Z^*_m| = m-1-(p-1)-(q-1) = pq-p-q+1 = (p-1)(q-1)$.

Note that $|Z^*_m| = |\mathbb{Z}^*_p| \cdot |\mathbb{Z}^*_q$. It turns out this is no accident:

**Theorem (Chinese Remainder Theorem – CRT):** If $m = pq$ then there is an isomorphism $\varphi : \mathbb{Z}^*_m \to \mathbb{Z}^*_p \times \mathbb{Z}^*_q$. That is, $\varphi$ is one to one and onto and maps $x \in \mathbb{Z}^*_m$ into a pair $(\varphi_1(x), \varphi_2(x)) \in \mathbb{Z}^*_p \times \mathbb{Z}^*_q$ such that for every $x, y \in \mathbb{Z}^*_m$: * $\varphi_1(x+y) = \varphi_1(x) + \varphi_1(y) \pmod p$ * $\varphi_2(x+y) = \varphi_2(x) + \varphi_2(y) \pmod q$ * $\varphi_1(x \cdot y) = \varphi_1(x) \cdot \varphi_1(y) \pmod p$ * $\varphi_2(x \cdot y) = \varphi_2(x) \cdot \varphi_2(y) \pmod q$

**Proof:** $\varphi$ simply maps $x \in \mathbb{Z}^*_m$ to the pair $(x \mod p, x \mod q)$. Verifying that it satisfies all desired properties is a good exercise. QED

In particular, for every polynomial $f()$ and $x \in \mathbb{Z}^*_m$, $f(x) = 0 \pmod m$ iff $f(x) = 0 \pmod p$ and $f(x) = 0 \pmod q$. Therefore finding the roots of a polynomial $f()$ modulo a composite $m$ is easy *if you know m's factorization*. However, if you don't know the factorization then this is hard. In particular, extracting square roots is as hard as finding out the factors:

**Theorem (Square root extraction implies factoring):** Suppose that $m = pq$ there is an efficient algorithm $A$ such that for every $a \in \mathbb{Z}^*_m$, $A(a^2 \pmod m) = b$ such that $a^2 = b^2 \pmod m$. Then, there is an efficient algorithm to recover $p, q$ from $m$.

**Proof:** Suppose that there is such an algorithm $A$. Using the CRT we can define $f : \mathbb{Z}^*_p \times \mathbb{Z}^*_q \to \mathbb{Z}^*_p \times Z^*_q$ as $f(x,y) = \varphi(A(\varphi^{-1}(x^2, y^2)))$ for all $x \in \mathbb{Z}^*_p$ and $y \in \mathbb{Z}^*_q$. Now, for any $x, y$ let $(x', y') = f(x,y)$. Since $x^2 = x'^2 \pmod p$ and $y^2 = y'^2 \pmod q$ we know that $x' \in \{\pm x\}$ and $y' \in \{\pm y\}$. Since flipping signs doesn't change the value of $(x', y') = f(x,y)$, by flipping one or both of the signs of $x$ or $y$ we can ensure that $x' = x$ and $y' = -y$. Hence $(x, x') - (y, y') = (0, 2y)$. In other words, if $c = \varphi^{-1}(x - x', y - y')$ then $c = 0 \pmod p$ but $c \neq 0 \pmod q$ which in particular means that the greatest common divisor of $c$ and $m$ is $q$. So, by taking $gcd(A(\varphi^{-1}(x,y)), m)$ we will find $q$, from which we can find $p = m/q$.

This almost works, but there is a question of how can we find $\varphi^{-1}(x,y)$, given that we don't know $p$ and $q$? The crucial observation is that we don't need to. We can simply pick a value $a$ at random in $\{1, \ldots, m\}$. With very high probability (namely $(p-1+q-1)/pq$) $a$ will be in $\mathbb{Z}^*_m$, and so we can imagine this process as equivalent to the process of taking a random $x \in \mathbb{Z}^*_p$, a random $y \in \mathbb{Z}^*_q$ and then flipping the signs of $x$ and $y$ randomly and taking $a = \varphi(x,y)$. By the arguments above with probability at least 1/4, it will hold that $gcd(a - A(a^2), m)$ will equal $q$. QED

Note that this argument generalizes to work even if the algorithm $A$ is an *average case* algorithm that only succeeds in finding a square root for a significant fraction of the inputs. This observation is crucial for cryptographic applications.

### The RSA and Rabin functions

We are now ready to describe the RSA and Rabin trapdoor functions:

**Def:** Given a number $m = pq$ and $e$ such that $gcd((p-1)(q-1), e) = 1$, the *RSA function* w.r.t $m$ and $e$ is the map $f_{m,e} : \mathbb{Z}_m^* \to \mathbb{Z}_m^*$ such that $RSA_{m,e}(x) = x^e \pmod{m}$.

**Def:** Given a number $m = pq$, the *Rabin function* w.r.t. $m$, is the map $Rabin_m : \mathbb{Z}_m^* \to Z_m^*$ such that $Rabin_m(x) = x^2 \pmod{m}$.

Note that both maps can be computed in polynomial time. Using the theorem above, we know that both functions can be *inverted* efficiently if we know the factorization (at least for not too large $e$; indeed $e$ is often ). However, it turns out that this is a much too big of a Hammer to solve this, and there are direct and simple inversion algorithms (see homework exercises). By the discussion above, inverting the Rabin function amounts to factoring $m$. No such result is known for the RSA function, but there is no better algorithm known to attack it than proceeding via factorization of $m$. The RSA function has the advantage that it is a *permutation* over $\mathbb{Z}_m^*$:

**Lemma:** $RSA_{m,e}$ is one to one over $\mathbb{Z}_m^*$.

**Proof:** Suppose that $RSA_{m,e}(a) = RSA_{m,e}(a')$. By the CRT, it means that there is $(x, y) \neq (x', y') \in \mathbb{Z}_p^* \times \mathbb{Z}_q^*$ such that $x^e = x'^e \pmod{p}$ and $y^e = y'^e \pmod{q}$. But if that's the case we get that $(xx'^{-1})^e = 1 \pmod{p}$ and $(yy'^{-1})^e = 1 \pmod{q}$. But this means that $e$ has to be a multiple of the *order* of $xx'^{-1}$ and $yy'^{-1}$ (at least one of which is *not* 1 and hence has order $> 1$). But since the order always divides the group size, this implies that $e$ has to have non-trivial gcd with either $|Z_p^*|$ or $|\mathbb{Z}_q^*|$ and hence with $(p-1)(q-1)$. QED

**Note:** The RSA trapdoor function is known also as "plain RSA encryption". This is because initially Diffie and Hellman (and following them, RSA) thought of an encryption scheme as a deterministic procedure. Today however we know that it is insecure to use a trapdoor function directly as an encryption scheme without adding some randomization.

### Abstraction: trapdoor permutations

**Def:** We can abstract away the particular construction of the RSA and Rabin functions to talk about a general *trapdoor permutation family (TDP)*. This is a family of functions $\{p_k\}$ such that for every $k \in \{0,1\}^n$, the function $p_k$ is a permutation on $\{0,1\}^n$ and the map $k, x \mapsto p_k(x)$ is efficiently computable, but:

* For every efficient adversary $A$, $\Pr_{y\in\{0,1\}^n}[A(y) = p_k^{-1}(y)] = negl(n)$. * There is a *key generation algorithm* $G$ such that on input $1^n$ it outputs a pair $(k, \tau)$ such that the map $\tau, y \mapsto p_k^{-1}(y)$.

**Note:** The reader might note that the RSA function is not a permutation over the set of strings but rather over $\mathbb{Z}_m^*$ for some $m = pq$. However, if we find primes $p, q$ in the interval $[2^{n/2}(1 - negl(n)), 2^{n/2}]$, then $m$ will be in the interval $[2^n(1-negl(n)), 2^n]$ and hence $\mathbb{Z}_m^*$ (which has size $pq-p-q+1 = 2^n(1-negl(n))$) can be thought of as essentially identical to $\{0,1\}^n$, since we will always pick elements from $\{0,1\}^n$ at random and hence they will be in $\mathbb{Z}_m^*$ with probability $1 - negl(n)$. It is widely believed that for every sufficiently large $n$ there is a prime in the interval $[2^n - poly(n), 2^n]$ (this follows from the *Extended Reimann Hypothesis*) and Baker, Harman and Pintz *proved* that there is a prime in the interval $[2^n - 2^{0.6n}, 2^n]$.[1]

## Public key encryption from trapdoor permutations

Here is how we can get a public key encryption from a trapdoor permutation scheme $\{p_k\}$.

- *Key generation:* Run the key generation algorithm of the TDP to get $(k, \tau)$. $k$ is the *public encryption key* and $\tau$ is the *secret decryption key.*
- *Encryption:* To encrypt a message $m$ with key $k \in \{0,1\}^n$, choose $x \in \{0,1\}^n$ and output $(p_k(x), H(x) \oplus m)$ where $H : \{0,1\}^n \to \{0,1\}^\ell$ is a hash function we model as a random oracle.
- *Decryption:* To decrypt the ciphertext $(y, z)$ with key $\tau$, output $m = H(p_k^{-1}(y)) \oplus z$.

**Theorem:** If $\{p_k\}$ is a secure TDP and $H$ is a random oracle then the above scheme is CPA secure public key encryption scheme.

**Proof:** (To be completed)

**Note:** We do *not* need to use a random oracle to get security in this scheme, especially if $\ell$ is sufficiently short. We can replace $H()$ with a hash function of specific properties known as a *hard core* construction; this was first shown by Goldreich and Levin.

## Digital signatures from trapdoor permutations

Here is how we can get digital signatures from trapdoor permutations $\{p_k\}$. This is known as the "full domain hash" signatures.

[1] Another, more minor issue is that the description of the key might not have the same length as $\log m$; I defined them to be the same for simplicity of notation, and this can be ensured via some padding and concatenation tricks.

- *Key generation:* Run the key generation algorithm of the TDP to get $(k, \tau)$. $k$ is the *public verification key* and $\tau$ is the *secret signing key*.
- *Signature:* To sign a message $m$ with key $\tau$, we output $p_k^{-1}(H(m))$ where $H : \{0,1\}^* \to \{0,1\}^n$ is a hash function modeled as a random oracle.
- *Verification:* To verify a message-signature pair $(m, x)$ we check that $p_k(x) = H(m)$.

**Theorem:** If $\{p_k\}$ is a secure TDP and $H$ is a random oracle then the above scheme is chosen message attack secure digital signature scheme.

**Proof:** (To be completed).

## Key exchange, authenticated and password-authenticated key exchange

(To be completed)