11
Concrete candidates for public key crypto

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.

11.1 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 $\mathbb{Z}^*_m = \{a \in \mathbb{Z}_m : gcd(a, m) = 1\}$. Note that m 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 $a \neq 0$.

R

Remark 11.1 — 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} to 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).

R

Remark 11.2 — The number m vs the message m. In much of this course we use m to denote a string which is our plaintext message to be encrypted or authenticated. In the context of integer factoring, it is convenient to use $m = pq$ as the composite number that is to be factored. To keep things interesting (or more honestly, because I keep running out of letters) in this lecture we will have both usages of m (though hopefully not in the same theorem or definition!). When we talk about factoring, RSA, and Rabin, then we will use m as the composite number, while in the context of the abstract trapdoor-permutation based encryption and signatures we will use m for the message. When you see an instance of m, make sure you understand what is its usage.

11.1.1 Primaliy testing

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 p, and testing whether it is prime. We showed in the previous lecture that a random n bit number is prime with probability at least $\Omega(1/n^2)$ (in fact the probability is $\frac{1\pm o(1)}{\ln n}$ by the Prime Number Theorem). We now discuss how we can test for primality.

Theorem 11.3 — Primality Testing. There is an $poly(n)$-time algorithm to test whether a given n-bit number is prime or composite.

Theorem 11.3 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.

Lemma 11.4 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 $a \in \{2, \dots, m-1\}$, 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`.

■

Lemma 11.4 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, specifically, there are about $N/2^{-\Theta(\log N/\log\log N)}$ pseudoprimes between 1 and N. If we choose a random number $m \in [2^n]$ and output it if and only if the algorithm of Lemma 11.4 algorithm outputs `YES` (otherwise resampling), then the probability we make a mistake and output a pseudoprime is equal to the ratio of the set of pseudoprimes in $[2^n]$ to the set of primes in $[2^n]$. Since there are $\Omega(2^n/n)$ primes in $[2^n]$, this ratio is $\frac{n}{2^{-\Omega(n/\log n)}}$ which is a negligible quantity. 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. The best known algorithms run in time roughly $2^{\tilde{O}(n^{1/3})}$ where n is the number of bits.

11.1.2 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 11.5 — Fundamental Theorem of Algebra, mod p version.** 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 $x_1, \dots, x_{d+1}$ 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* (numbers s such that $s^2 = x \mod p$). 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 11.6 — Finding roots.** 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).

11.1.3 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 11.7 — 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, φ 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. φ 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 11.8 — Square root extraction implies factoring. Suppose and there is an efficient algorithm A such that for every $m \in \mathbb{N}$ and $a \in \mathbb{Z}^*_m$, $A(m, a^2 \pmod m) = b$ 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 $f : \mathbb{Z}^*_p \times \mathbb{Z}^*_q \to \mathbb{Z}^*_p \times \mathbb{Z}^*_q$ 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 $(x, y) - (x', y') = (0, 2y)$. 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.

■

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.

11.1.4 The RSA and Rabin functions

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

Definition 11.9 — RSA function. 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 $RSA_{m,e}(x) = x^e \pmod m$.

Definition 11.10 — Rabin function. Given a number $m = pq$, the *Rabin function* w.r.t. m, is the map $Rabin_m : \mathbb{Z}^*_m \to \mathbb{Z}^*_m$ such that $Rabin_m(x) = x^2 \pmod m$.

Note that both maps can be computed in polynomial time. Using the Chinese Remainder Theorem and Theorem 11.6, we know that both functions can be *inverted* efficiently if we know the factorization.[1]

However Theorem 11.6 is a much too big of a Hammer to invert the RSA and Rabin functions, and there are direct and simple inversion algorithms (see homework exercises). By Theorem 11.8, 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 $\mathbb{Z}_m^*$:

[1] Using Theorem 11.6 to invert the function requires e to be not too large. However, as we will see below it turns out that using the factorization we can invert the RSA function for every e. Also, in practice people often use a small value for e (sometimes as small as $e = 3$) for reasons of efficiency.

Lemma 11.11 $RSA_{m,e}$ 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 $(p-1)(q-1)$.

■

Remark 11.12 — Plain/Textbook RSA. The RSA trapdoor function is known also as "plain" or "textbook" RSA encryption. This is because initially Diffie and Hellman (and following them, RSA) thought of an encryption scheme as a deterministic procedure and so considered simply encrypting a message x by applying $ESA_{m,e}(x)$. Today however we know that it is insecure to use a trapdoor function directly as an encryption scheme without adding some randomization.

11.1.5 Abstraction: trapdoor permutations

We can abstract away the particular construction of the RSA and Rabin functions to talk about a general *trapdoor permutation family*. We make the following definition

Definition 11.13 — Trapdoor permutation. A *trapdoor permutation family* (TDP) 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:

* There is a *key generation algorithm* G such that on input 1^n it outputs a pair (k, τ) such that the maps $k, x \mapsto p_k(x)$ and $\tau, y \mapsto p_k^{-1}(y)$ are efficiently computable.

- For every efficient adversary A, $\Pr_{(k,\tau)\leftarrow_R G(1^n), y\in\{0,1\}^n}[A(k,y) = p_k^{-1}(y)] < negl(n)$.

R

Remark 11.14 — Domain of permutations. 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 $[2^n - 2^{0.6n}, 2^n]$. [2]

[2] 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.

11.1.6 Public key encryption from trapdoor permutations

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

TDP-based public key encryption (TDPENC):

- *Key generation:* Run the key generation algorithm of the TDP to get (k,τ). k is the *public encryption key* and τ 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 τ, output $m = H(p_k^{-1}(y)) \oplus z$.

P

Please verify that you understand why TDPENC is a *valid* encryption scheme, in the sense that decryption of an encryption of m yields m.

Theorem 11.15 — Public key encryption from trapdoor permutations. If $\{p_k\}$ is a secure TDP and H is a random oracle then TDPENC is a CPA secure public key encryption scheme.

Proof. Suppose, towards the sake of contradiction, that there is a polynomial-size adversary A that succeeds in the CPA game of TD-PENC (with access to a random oracle H) with non-negligible advantage ϵ over half. We will use A to design an algorithm I that inverts the trapdoor permutation.

Recall that the CPA game works as follows:

- The adversary A gets as input a key $k \in \{0,1\}^n$.
- The algorithm A makes some polynomial amount of computation and $T_1 = poly(n)$ queries to the random oracle H and produces a pair of messages $m_0, m_1 \in \{0,1\}^\ell$.
- The "challenger" chooses $b^* \leftarrow_R \{0,1\}$, chooses $x^* \leftarrow_R \{0,1\}^n$ and computes the ciphertext $(y^* = p_k(x^*), z^* = H(x^*) \oplus m_{b^*})$ which is an encryption of m_{b^*}.
- The adversary A gets (y^*, z^*) as input, makes some additional polynomial amount of computation and $T_2 = poly(n)$ queries to H, and then outputs b.
- The adversary *wins* if $b = b^*$.

We make the following claim:

CLAIM: With probability at least ϵ, the adversary A will make the query x^* to the random oracle.

PROOF: Suppose otherwise. We will prove the claim using the "forgetful gnome" technique as used in the Boneh Shoup book. By the "lazy evaluation" paradigm, we can imagine that queries to H are answered by a "faithful gnome" that whenever presented with a new query x, chooses a uniform and independent value $w \leftarrow_R \{0,1\}^\ell$ as a response, and then records that $H(x) = w$ to use that as answers for future queries.

Now consider the experiment where in the challenge part we use a "forgetful gnome" that answers $H(x^*)$ by a uniform and independent string $w^* \leftarrow_R \{0,1\}^\ell$ and *does not* record the answer for future queries. In the "forgetful experiment", the second component of the ciphertext $z^* = w^* \oplus m_{b^*}$ is distributed uniformly in $\{0,1\}^\ell$ and independently from all other random choices, regardless of whether $b^* = 0$ or $b^* = 1$. Hence in this "forgetful experiment" the adversary gets no information about b^* and its probability of winning is at most $1/2$. But the forgetful experiment is identical to the actual experiment if the value x^* is only queried to H once. Apart from the query of x^* by the challenger, all other queries to H are made by the adversary. Under our assumption, the adversary makes the query x^* with probability at most ϵ, and conditioned on this not happening the two experiments are identical. Since the probability of winning in the forgetful experiment is at most $1/2$, the probability of winning in the overall experiment is less than $1/2+\epsilon$, thus yielding a contradiction and establishing the claim. (These kind of analyses on sample spaces can be confusing; See Fig. 11.1 for a graphical illustration of this argument.)

Given the claim, we can now construct our inverter algorithm I as follows:

- The input to I is the key k to the trapdoor permutation and $y^* = p_k(x^*)$. The goal of I is to output x^*.
- The inverter simulates the adversary in a CPA attack, answering all its queries to the oracle H by random values if they are new or the previously supplied answers if they were asked before. Whenever the adversary makes a query x to H, I checks if $p_h(x) = y^*$ and if so halts and outputs x.
- When the time comes to produce the challenge, the inverter I chooses z^* at random and provides the adversary with (y^*, z^*) where $z^* = w^* \oplus m_{b^*}$.[3]
- The inverter continues the simulation again halting an outputting x if the adversary makes the query x such that $p_k(x) = y^*$ to H.

[3] It would have been equivalent to answer the adversary with a uniformly chosen z^* in $\{0,1\}^\ell$, can you see why?

We claim that up to the point we halt, the experiment is identical to the actual attack. Indeed, since p_k is a permutation, we know that if the time came to produce the challenge and we have not halted, then the query x^* has not been made yet to H. Therefore we are free to choose an independent random value w^* as the value $H(x^*)$. (Our inverter does not know what the value x^* is, but this does not matter for this argument: can you see why?) Therefore, since by the claim the adversary will make the query x^* to H with probability at least ϵ, our inverter will succeed with the same probability. ■

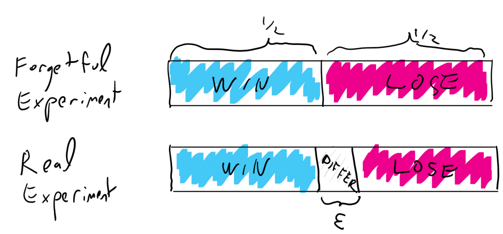

Figure 11.1: In the proof of security of TDPENC, we show that if the assumption of the claim is violated, the "forgetful experiment" is identical to the real experiment with probability larger $1-\epsilon$. In such a case, even if all that probability mass was on the points in the sample space where the adversary in the forgetful experiment will lose and the adversary of the real experiment will win, the probability of winning in the latter experiment would still be less than $1/2+\epsilon$.

P

This proof of Theorem 11.15 is not very long but it is somewhat subtle. Please re-read it and make sure you understand it. I also recommend you look at the version of the same proof in Boneh Shoup: Theorem 11.2 in Section 11.4 ("Encryption based on a trapdoor function scheme").

R

Remark 11.16 — Security without random oracles. We do *not* need to use a random oracle to get security in this scheme, especially if ℓ 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.

11.1.7 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.

Full domain hash signatures (FDHSIG):

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

We now prove the security of full domain hash:

Theorem 11.17 — Full domain hash security. If $\{p_k\}$ is a secure TDP and H is a random oracle then FDHSIG is chosen message attack secure digital signature scheme.

Proof. Suppose towards the sake of contradiction that there is a polynomial-sized adversary A that succeeds in a chosen message attack with non-negligible probability $\epsilon > 0$. We will construct an inverter I for the trapdoor permutation collection that succeeds with non-negligible probability as well.

Recall that in a chosen message attack the adversary makes T queries $m_1, \ldots, m_T$ to its signing box which are interspersed with T' queries $m'_1, \ldots, m'_{T'}$ to the random oracle H. We can assume without loss of generality (by modifying the adversary and at most doubling the number of queries) that the adversary always queries the message m_i to the random oracle *before* it queries it to the signing box, though it can also make additional queries to the random oracle (and hence in particular $T' \geq T$). At the end of the attack the adversary outputs with probability ϵ a pair (x^*, m^*) such that m^* was not queried to the signing box and $p_k(x^*) = H(m^*)$.

Our inverter I works as follows:

- **Input:** k and $y^* = p_k(y^*)$. Goal is to output x^*.

- I will guess at random t^* which is the step in which the adversary will query to H the message m^* that it is eventually going to forge in. With probability $1/T'$ the guess will be correct.

- I simulates the execution of A. Except for step t^*, whenever A makes a new query m to the random oracle, I will choose a random $x \leftarrow \{0,1\}^n$, compute $y = p_k(x)$ and designate $H(m) = y$. In step t^*, when the adversary makes the query m^*, the inverter I will return $H(m^*) = y^*$. I will record the values (x, y) and so in particular will always know $p_k^{-1}(H(m))$ for every $H(m) \neq y^*$ that it returned as answer from its oracle on query m.

- When A makes the query m to the signature box, then since m was queried before to H, if $m \neq m^*$ then I returns $x = p_k^{-1}(H(m))$ using its records. If $m = m^*$ then I halts and outputs "failure".

- At the end of the game, the adversary outputs (m^*, x^*). If $p_k(x^*) = y^*$ then I outputs x^*.

We claim that, conditioned on the probability $\geq \epsilon/T'$ event that the adversary is successful and the final message m^* is the one queried in step t^*, we provide a perfect simulation of the actual game. Indeed, while in an actual game, the value $y = H(m)$ will be chosen independently at random in $\{0,1\}^n$, this is equivalent to choosing $x \leftarrow_R \{0,1\}^n$ and letting $y = p_k(x)$. After all, a permutation applied to the uniform distribution is uniform.

Therefore with probability at least ϵ/T' the inverter I will output x^* such that $p_k(x^*) = y^*$ hence succeeding in the inverter.

■

P

Once again, this proof is somewhat subtle. I recommend you also read the version of this proof in Section 13.4 of Boneh-Shoup.

R

Remark 11.18 — Hash and sign. There is another reason to use hash functions with signatures. By combining a collision-resistant hash function $h : \{0,1\}^* \rightarrow \{0,1\}^\ell$ with a signature scheme (S, V) for ℓ-length messages, we can obtain a signature for arbitrary length messages by defining $S'_s(m) = S_s(h(m))$ and $V'_v(m, \sigma) = V_v(h(m), \sigma)$.

11.2 HARDCORE BITS AND SECURITY WITHOUT RANDOM ORACLES

To be completed.