Public key encryption candidates

Boaz Barak

**Disclaimer:** The MS-Word version of this text has significant formatting issues compared to the PDF and HTML versions.

# 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.

## Some number theory.

(See [Shoup’s excellent and freely available book](http://www.shoup.net/ntb/) for extensive coverage of these and many other topics.)

For every number , we define to be the set with the addition and multiplication operations modulo . When two elements are in then we will always assume that all operations are done modulo unless stated otherwise. We let . Note that is prime if and only if . For every we can find using the extended gcd algorithm an element (typically denoted as ) such that (can you see why?). The set is an abelian group with the multiplication operation, and hence by the observations of the previous lecture, for every . In the case that is prime, this result is known as “Fermat’s Little Theorem” and is typically stated as for every .

# 

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 to talk about our key size or security parameter, in which case we think of as a “small” number of size or so. However, when we work with we often think of as a “big” number having about *digits*; that is would be roughly to or so. I will try to reserve the notation for “small” numbers but may sometimes forget to do so, and other descriptions of RSA etc.. often use for “big” numbers. It is important that whenever you see a number , you make sure you have a sense whether it is a “small” number (in which case time is considered efficient) or whether it is a “large” number (in which case only time would be considered efficient).

# 

In much of this course we use 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 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 (though hopefully not in the same theorem or definition!). When we talk about factoring, RSA, and Rabin, then we will use as the composite number, while in the context of the abstract trapdoor-permutation based encryption and signatures we will use for the message. When you see an instance of , make sure you understand what is its usage.

### Primaliy testing

One procedure we often need is to find a prime of bits. The typical way people do it is by choosing a random -bit number , and testing whether it is prime. We showed in the previous lecture that a random bit number is prime with probability at least (in fact the probability is by the [Prime Number Theorem](https://goo.gl/ChrXJY)). We now discuss how we can test for primality.

# 

There is an -time algorithm to test whether a given -bit number is prime or composite.

primalitytesting 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.

# 

There is a probabilistic polynomial time algorithm that on input a number , if is prime outputs YES with probability and if is not even a “pseudoprime” it outputs NO with probability at least . (The definition of “pseudo-prime” will be clarified in the proof below.)

# 

The algorithm is very simple and is based on Fermat’s Little Theorem: on input , pick a random , and if or return NO and otherwise return YES.

By Fermat’s little theorem, the algorithm will always return YES on a prime . We define a “pseudoprime” to be a non-prime number such that for all such that .  
If is *not* a pseudoprime then the set is a strict subset of . But it is easy to see that is a *group* and hence must divide and hence in particular it must be the case that and so with probability at least the algorithm will output NO.

pseudoprimelem 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 pseudoprimes between and . If we choose a random number and output it if and only if the algorithm of pseudoprimelem 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 to the set of primes in . Since there are primes in , this ratio is 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 where is the number of bits.

### Fields

If is a prime then is a *field* which means it is closed under addition and multiplication and has and elements. One property of a field is the following:

# 

If is a nonzero polynomial of degree over then there are at most distinct inputs such that .

(If you’re curious why, you can see that the task of, given finding the coefficients for a polynomial vanishing on the ’s amounts to solving a linear system in variables with equations that are independent due to the non-singularity of the Vandermonde matrix.)

In particular every has at most two *square roots* (numbers such that ). In fact, just like over the reals, every either has no square roots or exactly two square roots of the form .

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

# 

There is a probabilistic time algorithm to find the roots of a degree polynomial over .

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](http://www.shoup.net/ntb/)).

### Chinese remainder theorem

Suppose that is a product of two primes. In this case does not contain *all* the numbers from to . Indeed, all the numbers of the form and will have non-trivial g.c.d. with . There are exactly such numbers (because and are prime all the numbers of the forms above are distinct). Hence .

Note that . It turns out this is no accident:

# 

If then there is an isomorphism . That is, is one to one and onto and maps into a pair such that for every :  
\*   
\*   
\*   
\*

# 

simply maps to the pair . Verifying that it satisfies all desired properties is a good exercise. QED

In particular, for every polynomial and , iff and . Therefore finding the roots of a polynomial modulo a composite is easy *if you know ’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:

# 

Suppose and there is an efficient algorithm such that for every and , such that . Then, there is an efficient algorithm to recover from .

# 

Suppose that there is such an algorithm . Using the CRT we can define as for all and . Now, for any let . Since and we know that and . Since flipping signs doesn’t change the value of , by flipping one or both of the signs of or we can ensure that and . Hence . In other words, if then but which in particular means that the greatest common divisor of and is . So, by taking we will find , from which we can find .

This almost works, but there is a question of how can we find , given that we don’t know and ? The crucial observation is that we don’t need to. We can simply pick a value at random in . With very high probability (namely ) will be in , and so we can imagine this process as equivalent to the process of taking a random , a random and then flipping the signs of and randomly and taking . By the arguments above with probability at least , it will hold that will equal .

Note that this argument generalizes to work even if the algorithm 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:

# 

Given a number and such that , the *RSA function* w.r.t and is the map such that .

# 

Given a number , the *Rabin function* w.r.t. , is the map such that .

Note that both maps can be computed in polynomial time. Using the Chinese Remainder Theorem and rootfindingthm, we know that both functions can be *inverted* efficiently if we know the factorization.[[1]](#footnote-42)  
However rootfindingthm 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 squarerootfactthm, inverting the Rabin function amounts to factoring . No such result is known for the RSA function, but there is no better algorithm known to attack it than proceeding via factorization of . The RSA function has the advantage that it is a *permutation* over :

# 

is one to one over .

# 

Suppose that . By the CRT, it means that there is such that and . But if that’s the case we get that and . But this means that has to be a multiple of the *order* of and (at least one of which is *not* and hence has order ). But since the order always divides the group size, this implies that has to have non-trivial gcd with either or and hence with .

# 

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 by applying . 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

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

# 

A *trapdoor permutation family (TDP)* is a family of functions such that for every , the function is a permutation on and:  
\* There is a *key generation algorithm* such that on input it outputs a pair such that the maps and are efficiently computable.

* For every efficient adversary , .

# 

The RSA function is not a permutation over the set of strings but rather over for some . However, if we find primes in the interval , then will be in the interval and hence (which has size ) can be thought of as essentially identical to , since we will always pick elements from at random and hence they will be in with probability . It is widely believed that for every sufficiently large there is a prime in the interval (this follows from the *Extended Reimann Hypothesis*) and Baker, Harman and Pintz *proved* that there is a prime in the interval .[[2]](#footnote-49)

### Public key encryption from trapdoor permutations

Here is how we can get a public key encryption from a trapdoor permutation scheme .

**TDP-based public key encryption (TDPENC):**

* *Key generation:* Run the key generation algorithm of the TDP to get . is the *public encryption key* and is the *secret decryption key*.
* *Encryption:* To encrypt a message with key , choose and output where is a hash function we model as a random oracle.
* *Decryption:* To decrypt the ciphertext with key , output .

# 

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

# 

If is a secure TDP and is a random oracle then TDPENC is a CPA secure public key encryption scheme.

# 

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

Recall that the CPA game works as follows:

* The adversary gets as input a key .
* The algorithm makes some polynomial amount of computation and queries to the random oracle and produces a pair of messages .
* The “challenger” chooses , chooses and computes the ciphertext which is an encryption of .
* The adversary gets as input, makes some additional polynomial amount of computation and queries to , and then outputs .
* The adversary *wins* if .

We make the following claim:

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

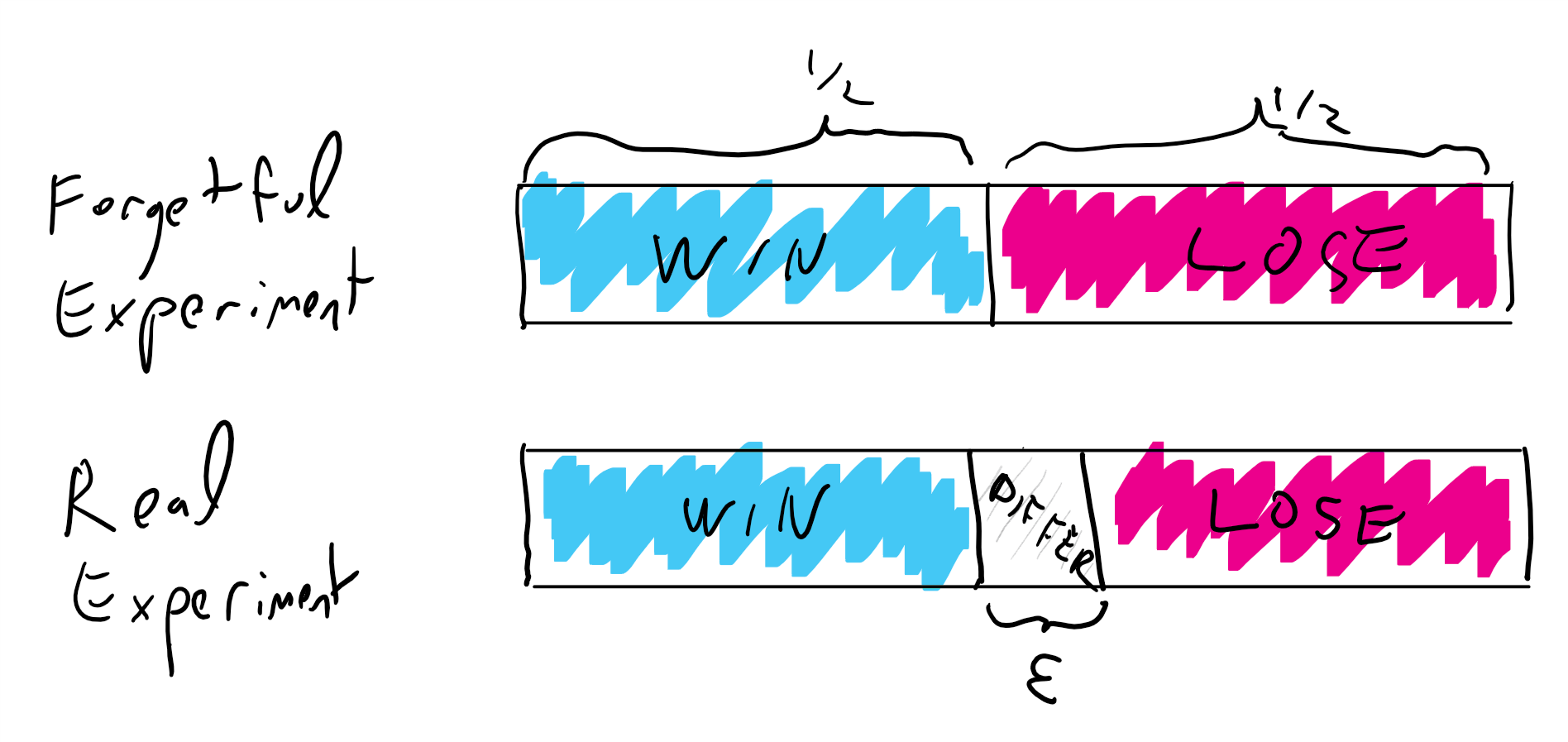
**PROOF:** Suppose otherwise. We will prove the claim using the “forgetful gnome” technique as used in the Boneh Shoup book. By the “lazy evaluation” paradigm, we can imagine that queries to are answered by a “faithful gnome” that whenever presented with a new query , chooses a uniform and independent value as a response, and then records that to use that as answers for future queries.

Now consider the experiment where in the challenge part we use a “forgetful gnome” that answers by a uniform and independent string and *does not* record the answer for future queries. In the “forgetful experiment”, the second component of the ciphertext is distributed uniformly in and independently from all other random choices, regardless of whether or . Hence in this “forgetful experiment” the adversary gets no information about and its probability of winning is at most . But the forgetful experiment is identical to the actual experiment if the value is only queried to once. Apart from the query of by the challenger, all other queries to are made by the adversary. Under our assumption, the adversary makes the query with probability at most , and conditioned on this not happening the two experiments are identical. Since the probability of winning in the forgetful experiment is at most , the probability of winning in the overall experiment is less than , thus yielding a contradiction and establishing the claim. (These kind of analyses on sample spaces can be confusing; See TDPENCgnomefig for a graphical illustration of this argument.)

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

* The input to is the key to the trapdoor permutation and . The goal of is to output .
* The inverter simulates the adversary in a CPA attack, answering all its queries to the oracle by random values if they are new or the previously supplied answers if they were asked before. Whenever the adversary makes a query to , checks if and if so halts and outputs .
* When the time comes to produce the challenge, the inverter chooses at random and provides the adversary with where .[[3]](#footnote-54)
* The inverter continues the simulation again halting an outputting if the adversary makes the query such that to .

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



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 . 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 .

# 

This proof of TDPpkcthm 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”).

# 

We do *not* need to use a random oracle to get security in this scheme, especially if is sufficiently short. We can replace 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 . 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 . is the *public verification key* and is the *secret signing key*.
* *Signing:* To sign a message with key , we output where is a hash function modeled as a random oracle.
* *Verification:* To verify a message-signature pair we check that .

We now prove the security of full domain hash:

# 

If is a secure TDP and is a random oracle then FDHSIG is chosen message attack secure digital signature scheme.

# 

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

Recall that in a chosen message attack the adversary makes queries to its signing box which are interspersed with queries to the random oracle . 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 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 ). At the end of the attack the adversary outputs with probability a pair such that was not queried to the signing box and .

Our inverter works as follows:

* **Input:** and . Goal is to output .
* will guess at random which is the step in which the adversary will query to the message that it is eventually going to forge in. With probability the guess will be correct.
* simulates the execution of . Except for step , whenever makes a new query to the random oracle, will choose a random , compute and designate . In step , when the adversary makes the query , the inverter will return . will record the values and so in particular will always know for every that it returned as answer from its oracle on query .
* When makes the query to the signature box, then since was queried before to , if then returns using its records. If then halts and outputs “failure”.
* At the end of the game, the adversary outputs . If then outputs .

We claim that, conditioned on the probability event that the adversary is successful and the final message is the one queried in step , we provide a perfect simulation of the actual game. Indeed, while in an actual game, the value will be chosen independently at random in , this is equivalent to choosing and letting . After all, a permutation applied to the uniform distribution is uniform.

Therefore with probability at least the inverter will output such that hence succeeding in the inverter.

# 

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

# 

There is another reason to use hash functions with signatures. By combining a collision-resistant hash function with a signature scheme for -length messages, we can obtain a signature for arbitrary length messages by defining and .

## Hardcore bits and security without random oracles

To be completed.

1. Using rootfindingthm to invert the function requires 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 . Also, in practice people often use a small value for (sometimes as small as ) for reasons of efficiency. [↑](#footnote-ref-42)
2. Another, more minor issue is that the description of the key might not have the same length as ; I defined them to be the same for simplicity of notation, and this can be ensured via some padding and concatenation tricks. [↑](#footnote-ref-49)
3. It would have been equivalent to answer the adversary with a uniformly chosen in , can you see why? [↑](#footnote-ref-54)