KUNGL
TEKNISKA
HÖGSKOLAN

On the bit-security of RSA and related questions

Johan Håstad
Royal Institute of Stockholm
Stockholm, SWEDEN

Outline

1. The RSA function.
2. One-way functions.
3. Hard-core predicates.
4. The bits of RSA as hard-core predicates.
5. Generalizations (discrete logarithms).

The RSA function

$$
\begin{aligned}
N &= pq \\
ed &\equiv 1 \bmod \phi(N) \\
E_N(x) &\equiv x^e \bmod N
\end{aligned}
$$

Decrypted as $c^d \bmod N$

N has n bits.

Notation $[x]_N$ is $x \bmod N$.

World view

We are interested in efficient computation which is defined as probabilistic polynomial time.

We can do any probabilistic polynomial time computation and nothing more.

General background

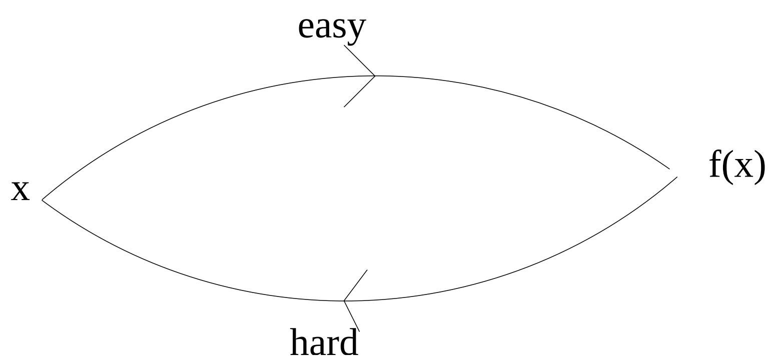
If we can factor integers then we can invert the RSA function.

The converse is not known.

We assume that we cannot invert RSA without knowledge of d or the factorization of N, i.e. given N, e and $[x^e]_N$ it is hard to find x.

True over random x for most N which is the product of two large primes and all suitable e.

One-way functions



RSA is a one-way function.

It even has the stronger property of being a trap door function. Knowledge of d enables us to invert. The latter is of little importance for us today.

General question

Given $f(x)$ for a one-way function f what can we compute about x?

All of x?

The least significant bit of x?

The exclusive or of the bits of x?

Consider functions B, $\{0,1\}^n \mapsto \{0,1\}^m$

Some B are easy

A general one-way function can output any $B(x)$.

For RSA, the Jacobi symbol, $\left(\frac{x}{N}\right)$ is easy as

$$\left(\frac{x}{N}\right) = \left(\frac{x}{N}\right)^e = \left(\frac{x^e}{N}\right) = \left(\frac{E_N(x)}{N}\right)$$

Hard-core functions and predicates

A function B is a hard-core function for f iff

- It is easy to compute $B(x)$ given x.
- It is hard to compute $B(x)$ given $f(x)$.

If $m = 1$ we have a hard-core predicate.

Hard?

For any probabilistic polynomial time predictor P.

Definition 1. Given $f(x)$ the probability of P computing $B(x)$ is $2^{-m} + O(n^{-d})$ for any d.

Definition 2. Given $f(x)$, R and $B(x)$ where R are m random bits, the probability of P guessing which is R and which is $B(x)$ is $\frac{1}{2} + O(n^{-d})$ for any d.

Equivalent definitions for $m \in O(\log n)$.

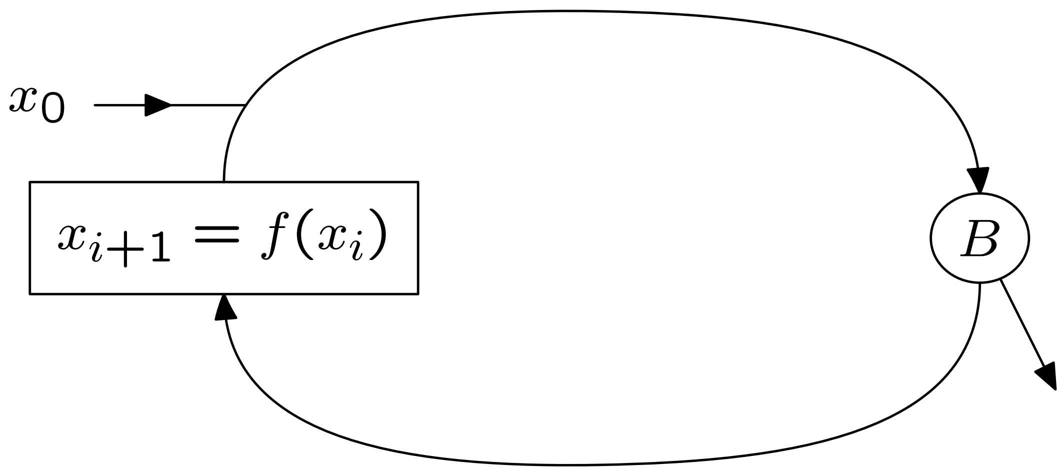
Probability is over random x and the random choices of the predictor.

Why hard-core predicates?

The property of being hard-core is simply a basic mathematical object. Studied for its own sake. What can we compute?

Hard-core functions give a cornerstone of modern cryptography.

Pseudorandom generators



If B is hard-core and f is a permutation this produces provably good pseudorandom bits. Can replace real random bits in any computation with no effect.

Commitments/Encryption from hard-core

To encrypt/commit $z \in \{0,1\}^m$ chose random x with $B(x) = z$ and send $f(x)$.

If trap door decrypt with trap door information.

If only one-way it is commitment and decommit by sending x.

Note that sending $f(z)$ is in general not a good idea.

General hard-core

Best: One hard-core B that works for all f. Cannot be deterministic.

Goldreich and Levin, 1989:

$B_r(x) \equiv \sum_{i=1}^{n} r_i x_i \bmod 2$

for random and public $r \in \{0, 1\}^n$ is a hard-core predicate for any one-way f.

High level proof of Goldreich-Levin

Assume not hard core, i.e., have $\frac{1}{2}+n^{-d}$ predictor P for $B_r(x)$ for constant d.

Mostly forget f.

For varying r for fixed x we get a nontrivial prediction of $B_r(x)$. Use this to find candidates for x.

Finally use f to check candidates.

Goldreich-Levin phrased differently

$$(B_r(x))_{r\in\{0,1\}^n}$$

is an implicitly list-decodable code.

It can tolerate a fraction $(\frac{1}{2} - n^{-d})$ errors and there is a probabilistic polynomial time algorithm that with high probability recovers all code words with at most this fraction of errors.

RSA hard-core

RSA has additional structure to allow us to prove that special B are hard-core.

$$\begin{aligned} bit_i^j(x) &= x_j x_{j-1} \ldots x_i \\ bit_i &= bit_i^i(x) = x_i \end{aligned}$$

b_0 is least significant bit and also denoted by lsb.

$Half(x) = 1$ iff $N/2 \leq x \leq N$.

$lsb(x) = Half(x/2)$.

Main theorem

Theorem 1 *For any* $j - i \in O(\log n)$*,* $bit_i^j(x)$ *is a hard-core function for the RSA encryption function.*

We concentrate on $j = i$, the i'th bit.

Needed structure from RSA

Obvious multiplicative structure.

$$E_N(cx) \equiv E_N(c)E_N(x) \bmod N$$

If we can predict the i'th bit of x then we can predict i'th bit of $[cx]_N$.

Additive structure

Need additive structure.

In general given $E_N(y), E_N(z)$ seems hard to compute $E_N(y+z)$ but if $y = ax$ and $z = bx$ for known a and b then

$$E_N(y+z) = E_N(a+b)E_N(x).$$

Almost all numbers studied are of this form which implies that RSA has additive structure.

First case, lsb

Originally from 1984 Alexi et al [ACGS] and new proof in 1997 by Fischlin and Schnorr.

We do three steps.

1. First assume (almost) perfect predictor.

2. A predictor with predicting probability $\frac{3}{4}+\epsilon$. Improve to almost perfect predictor.

3. A predictor with predicting probability $\frac{1}{2}+n^{-d}$. Again improve.

Easy case, lsb

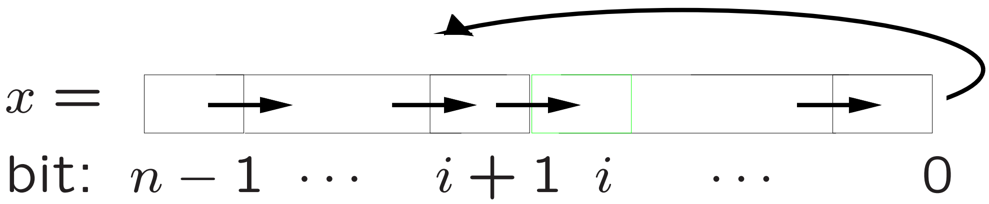
Suppose we have a perfect predictor for least significant bit.

Compute the lsb of $2^{-j}x$, $0 \leq j \leq n$ and this is enough to find x.

$lsb(x) = Half(x/2)$

$lsb(x)$, $lsb(x/2)$ gives gives size of $x/4$ upto $N/4$ etc.

In the end we know $2^{-n}x$ perfectly.



Improving $3/4+\epsilon$ predictor for small ax

Suppose $0 \leq [ax]_N \leq \epsilon N/2$, then

$$lsb(ax) = lsb(cx) \oplus lsb(cx + ax)$$

for most e.

Pick random c predict both values.

Probability of being incorrect $\leq 2(\frac{1}{4} - \epsilon) \leq \frac{1}{2} - 2\epsilon$.

Take majority vote over result for all c.

Limiting problem: Error doubling due to the two predictions.

Avoiding error-doubling

Having precomputed c_j with $lsb(c_j x)$ known and the let the answer be

$$majority_j(P(E_N((c_j + a)x)) \oplus lsb(c_j x))$$

Pairwise independent polynomially many c_j are sufficient.

Constructing the c_j

Take two random base points s and t and let

$$c_j \equiv js + t \bmod N$$

"Guess" $lsb([sx]_N), lsb([tx]_N)$ and the approximate sizes of $[sx]_N$ and $[tx]_N$.

This determines $lsb([c_j x]_N)$ (and the its approximate size) for most j.

The size guesses are needed to calculate number of reductions by N.

The punch line

A not unlucky choice of s and t together with the correct information produces a predictor for $lsb([ax]_N)$ which is correct with probability $1 - \frac{1}{n^2}$.

Use the same c_j for all iterations of the sliding technique.

Polynomially many possibilities for the information on sx and tx.

When correct x is found it can be verified.

No need to worry about what happens with incorrect information.

Eliminating "small" condition

In fact it is now sufficient to know the approximate size of $[ax]_N$. Since we know size of $[c_j x]_N$ we know $b \in \{0, 1\}$ such that $[(a + c_j)x]_N = [ax]_N + [c_j x]_N - bN$.

- "Guess" initially approximate size of x.
- Automatically get sizes of all numbers in all iterations of sliding.

Interior bits

Needed another 14 years [HN].

Same approach.

Determine i'th bit of x and then look at $x/2$ etc.

Two new problems.

1. Carries into i'th bit from lower order bits close to i.

2. Least significant bit being one can add $(N+1)/2$ to $x/2$.

Dealing with the two problems

First is not a serious problem. We just keep track of $bit_{i-c\log n}^{i-1}$ of all numbers.

Guessed initially, behaves nicely under iteration.

Second problem with lsb is a major problem. Need to determine lsb as well as bit_i in each number.

Set up

Points $c_j x$ with known information. At iteration k we are looking at $2^{-k}x$ but could be any ax.

$c_j x =$	known	known		
$ax =$	known	b'	known	b
bit:	$n-1 \cdots$	$i+1\, i$	$\cdots$	0

Looking at $ax/2$

The i'th bit of $ax/2$ depends on $i+1$'st bit of ax and possibly lsb.

Try to determine both bits looking at $P(E_N(c_j + a/2)x))$ for pregenerated points c_j.

Many cases to consider

Easy case: Good predictor and N has a nice binary expansion around bit i.

Bad case: Weak predictor and N has only zeros around bit i in the binary expansion.

Several other possibilities not discussed here.

Easy case

$bit_i((c_j+a/2)x)$ depends heavily on both $bit_{i+1}(ax)$ and $lsb(ax)$ and by appropriate sampling we can determine both.

Shift and all needed information can be updated automatically.

One hard case

$$N = 2^i N_1 + N_2$$

with $N_1 = 2^k u$ for large k and $N_2 \ll 2^i$.

lsb has originally no influence and can be ignored temporarily but not for ever and if not treated it will kill you. “A small growing cancer”.

Solution in this hard case (and some others)

$bit_i(x) \approx half(N_1 x/2)$ with very good accuracy.

Define $\tilde{P}(x) = P((N_1/2)^{-1}x)$ to get a predictor for $half$ (and hence lsb) and the [ACGS] (or [FS]) result applies.

Minor problem

$N_1/2$ might not be invertible.

In this case we have factored N and we can invert RSA in any case.

This only happens for special numbers where RSA is insecure.

Omitted details

There are other bad cases, in particular multiples of N with bad behavior around bit i.

The fact that predictor only has correctness probability $\frac{1}{2} + n^{-b}$ calls for elaborate sampling.

It all works out......

Extension

Extending from one to $O(\log n)$ is standard.

Some care is needed but no new ideas.

Hard-core in practice

RSA standard PKCS1 had a standard of RSA encryption that demanded some bits in clear text had certain values.

Incorrect crypto texts were discarded by some implementations.

This yielded a predictor of these bits.

Bleichenbacher in 1998 used this to invert RSA.

Generalizing

We proved

Predicting $bit_i([ax]_N)$ for unknown x and chosen a allows us to reconstruct x (or factor N).

This we call *Chosen multiplier hidden number problem*.

Works in this generality. Examples show that we cannot always reconstruct x when a factor of N appears.

Another example

Not so famous one-way function "exponentiation".

$$f(x) \equiv g^x \bmod p$$

for generator g.

More famous is the hard problem "discrete logarithms".

Compare to the one-way function "multiplication" and hard problem "factorization".

Bit security of discrete logarithms

Given $y \equiv g^x \bmod p$ are the bits of x hard to find?

Note that

$$lsb(x) = \left(\frac{y}{p}\right)$$

which is easy to find through computing $y^{(p-1)/2}$.

This can be extended to first t bits provided $2^t|(p-1)$.

General result applies to other bits

$$g^{ax} \equiv (g^x)^a \bmod p$$

and even

$$g^{ax+b} \equiv (g^x)^a g^b \bmod p$$

We need to think a little bit to take care of the fact that $p - 1$ is even. We work in the odd part.

The abstract chosen multiplier hidden number problem can be applied.

Discrete logarithm theorem

Theorem 2 *Let p be prime and $(p-1) = 2^t u$ with u odd. For most p all but the t least significant bits are hard core predicates for exponentiation modulo p.*

Most p only ?

If the construction returns a factor in $p-1$ this does not help us much.

If the factor is small we can extend the result.

We prove that a large factor is only returned for few p.

Open problems

- Find another nontrivial, easy predicate for RSA apart from the Jacobi symbol.

- Establish that all bits except first t are hard for all discrete logarithm problems.

- Establish whether inverting RSA is equivalent to factoring.