Survey: Lattice Reduction Attacks on RSA

David Wong
supervised by Guilhem Castagnos

University of Bordeaux, March 2015

Abstract

RSA, carrying the names of **Ron Rivest**, **Adi Shamir** and **Leonard Adleman**, is one of the first practicable **public-key** cryptosystem. The algorithm was publicly described for the first time in **1977** and has since been the most used cryptosystem when it comes to asymmetric problems. For now more than **30 years**, Cryptanalysts and Researchers have looked for ways to **attack RSA**.
One branch of cryptanalysis on RSA is to take an interest in a **relaxed model** of RSA. A model where we know part of the message, or we know an approximation of the primes, or the private exponent is too small... In these sorts of problems, **lattice reduction techniques** have proved to be very relevant. **Coppersmith** opened the way with his constructive theorem on how to find small roots of **univariate** polynomials using reductions of lattices. **Boneh** and **Durfee** recently followed with a method on how to find small roots of **bivariate** polynomials using Coppersmith's heuristics on multivariate polynomials. In this survey we will see how each algorithm work and how they were respectively made simpler by **Howgrave-Graham** and the duo **Herrmann** and **May**.

Keywords : RSA, lattice, LLL, Coppersmith, Howgrave-Graham, Boneh-Durfee, Herrmann-May.

1 Introduction

In 1995, **Coppersmith** released a paper on how to attack RSA using **Lattices** and **Lattice reduction techniques** such as **LLL**. A few years later, **Howgrave-Graham** revisited Coppersmith's algorithm and made it easier to understand and apply. His work was implemented for various problems from revealing part of a message if most of the message was known, to breaking RSA if a good enough approximation of one of the prime was known. Attacks based on Lattice reduction techniques caught up and several researches were done on the subject. In 1990, **Wiener** had found that you

could successfully break RSA if the private exponent was too small ($d < N^{1/4}$). In 2000, **Boneh** and **Durfee** improved that bound ($d < N^{0.292}$) using lattices and LLL in a Coppersmith-like attack. Their work was later simplified by **Herrmann** and **May**.
In the 2nd and 3rd sections of this survey I will briefly explain how RSA and Lattice work. In section 4 we will see in what **model** the attacks are taking place and see **Håstad's Broadcast Attack** as an introduction to Coppersmith. Section 5 will be an overview of the Coppersmith algorithm revisited by Howgrave-Graham. Section 6 will be an overview of the Boneh and Durfee algorithm revisited by Herrmann and May. Finally the **implementations** of both attack will be added as an appendix.

2 RSA

Let's quickly recall **what is** and **how** RSA works :
RSA is an **asymmetric cryptosystem**. A generator algorithm derives two kinds of keys : a **public key** and a **private key**, both can be used either to encrypt or decrypt thanks to the asymmetric property of RSA to allow us to use the system as an **encryption** system or as a **signature** system.

2.1 Generation

To use RSA for **encryption** we need a public key to encrypt and a private key to decrypt. We first generate the **public key** as follow :

We generate **two primes** p and q. For security issues they should be around the **same size**. Those primes are the **core elements** of RSA. Knowing one of those allows us to compute the private key, thus allowing us to break the system. They can also be used to speed up calculations using the Chinese Remainder Theorem.
Knowing p and q we can then compute the **modulus** $N = p \times q$ which will be part of the public key as well as the private key. And you will see why.

Now comes the interesting part, we need to find a **public exponent** which will be used for **encryption**. For computation purpose a **Fermat prime** $(2^m + 1)$ is often used as public exponent (it makes things faster in **binary exponentiation**). In the case of a **signature scheme**, we would want the speed up to occur for the **private exponent** so we would use such a number as a private exponent and we would reverse the following steps.
Anyway, any kind of exponent can theoretically be chosen, as long as it is coprime with $\varphi(N)$ (The **Euler's Totient function**).

$$e \leftarrow \mathbb{Z}^*_{\varphi(N)}$$

if e is **coprime** with $\varphi(N)$ then it is part of the multiplicative group $(\mathbb{Z}/\varphi(N)\mathbb{Z})^*$ and thus **invertible** in $\mathbb{Z}/\varphi(N)\mathbb{Z}$.

Now it is pretty easy to find the private exponent d by inverting our public exponent e.

All of this is possible because we can easily compute $\varphi(N)$:

$$\varphi(N) = (p-1) \times (q-1)$$

And here resides the **security** of RSA. Imagine for a moment that we could easily factor N into p and q, then we would be able to invert the public exponent e. That's why we say that **the security of RSA is reduced to the Factorization Problem**.

Now let the **private key** be **(N, d)** with the addition of (p, q) if we need to speed up calculations. And let the **public key** be **(N, e)**.

2.2 Encryption/Decryption

To **encrypt** a message m, with $m < N$ we just do :

$$c = m^e \pmod{N}$$

And to **decrypt** :

$$m = c^d \pmod{N}$$

This works because the decryption step gives us :

$$c^d = (m^e)^d \pmod{N}$$

And e being d's inverse tells us that :

$$\begin{aligned} e &= d^{-1} \pmod{\varphi(N)} \\ \implies ed &= 1 \pmod{\varphi(N)} \\ \implies ed &= \varphi(N) + 1 \end{aligned}$$

Coupled with **Euler's Theorem** stating that if a and n are coprime then :

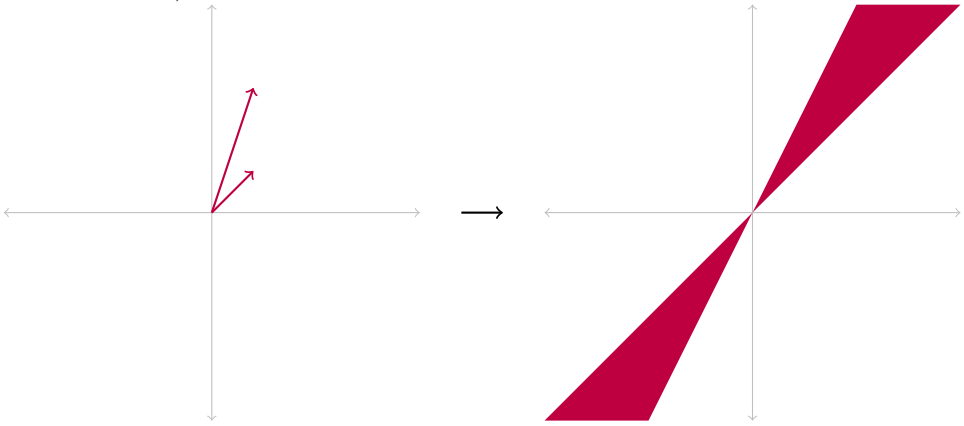
$$a^{\varphi(n)} = 1 \pmod{n}$$

Tells us that $m^{ed} = m \pmod{N}$

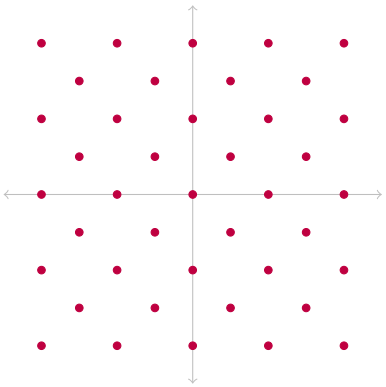
3 Lattice

3.1 Introduction

The attacks we will describe later both make use of the **Lenstra–Lenstra–Lovász lattice basis reduction algorithm**. Hence it is necessary for us to understand what is a lattice and why is this **LLL** algorithm so useful.
Think about Lattices like **Vector Spaces**. Imagine a simple vector space of two vectors. You can add them together, multiply them by scalars (let's say numbers of $\mathbb{R}$) and it spans a vector space.



Now imagine that our vector space's **scalars are the integers**, taken in $\mathbb{Z}$. The space spanned by the vectors is now made out of points. It's **discrete**. Meaning that for any point of this lattice there is a ball centered around that point of radius different from zero that contains only that point. Nothing else.



Lattice are interesting in cryptography because we seldom deal with real numbers and they bring us a lot of tools to deal with integers.
Just as vector spaces, lattices can also be described by different basis represented as **matrices**. Contrary to vector spaces, we generally represent the

vectors of the basis as **rows** in their corresponding matrices.

Last but not least, if $\{\tilde{b}_1 \ldots, \tilde{b}_w\}$ are the vectors of the Gram-Schmidt basis of a lattice L then we define the **determinant** of the lattice as such :

$$det(L) := \prod_{i=1}^{w} \|\tilde{b}_i\|$$

You will see that in the technique we present, to easily compute the determinant of a lattice we will make the lattice **full rank** (dimension = rank) and **triangular**. So that the determinant is computable by doing the products of the **diagonal terms** of the lattice basis.

3.2 LLL

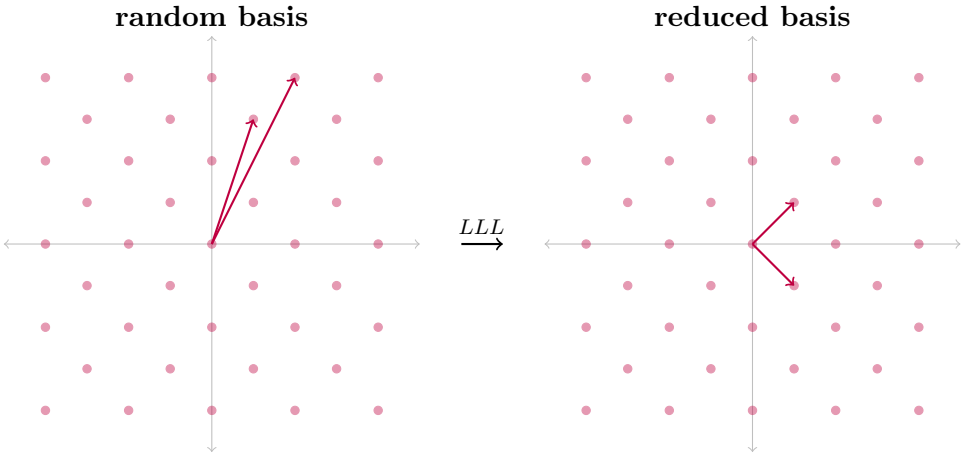
The **Lenstra–Lenstra–Lovász** *lattice basis reduction algorithm* is a step by step calculus that reduces a lattice basis in polynomial time. The lattice is left unchanged but the row vectors of its new basis are "**smaller**" according to some definitions :

Definition 1. *Let L be a lattice with a basis B. The δ-LLL algorithm applied on L's basis B produces a new basis of L : $B' = \{b_1, \ldots, b_n\}$ satisfying :*

$$\forall\, 1 \leq j < i \leq n \text{ we have } |\mu_{i,j}| \leq \frac{1}{2} \tag{1}$$

$$\forall\, 1 \leq i < n \text{ we have } \delta \cdot \|\tilde{b_i}\|^2 \leq \|\mu_{i+1,i} \cdot \tilde{b}_i + \tilde{b}_{i+1}\|^2 \tag{2}$$

$$\text{with } \mu_{i,j} = \frac{b_i \cdot \tilde{b}_j}{\tilde{b}_j \cdot \tilde{b}_j} \text{ and } \tilde{b}_1 = b_1 \text{ (Gram-Schmidt)}$$



We will not dig into the interals of LLL here, see Chris Peikert's course[1] for detailed explanations of the algorithm.

3.3 Wanted properties of LLL

LLL yields an approximation of the **Shortest Vector Problem**. This is useful for us because if we consider the row vectors of a lattice's basis as **coefficient vectors of polynomials**. We can find a **linear combination** of those polynomials that has **"particularly small" coefficients**. But let's not unveil too much too soon. Here is the relevant property of a LLL reduced basis that we will need later :

Property 1. *Let L be a lattice of dimension n. In polynomial time, the LLL algorithm outputs reduced basis vectors v_i, for $1 \leq i \leq n$, satisfying :*

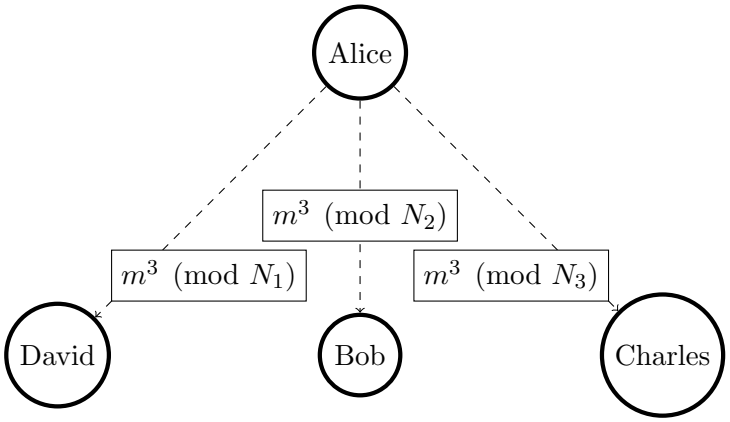
$$\|v_1\| \leq \|v_2\| \leq \ldots \leq \|v_i\| \leq 2^{\frac{n(n-1)}{4(n+1-i)}} \cdot det(L)^{\frac{1}{n+1-i}}$$

We can see that we can modify the bound on our vectors by modifying the dimension and the determinant of the lattice basis.

4 Relaxed models and small roots problem

Attacks on RSA falls into two categories : the attacks on the **implementation** or the **mathematical attacks**. Over the years the mathematical cryptanalysis on RSA have proven to be hard and thus the cryptosystem is still considered as secure nowadays (march 2015). But what a researcher could find interesting is to attack a **relaxed** model of the RSA problem. What if we knew "a part" of the message, or what if we knew "an approximation" of one of the prime, or what if the private exponent was "too small"...

Let's imagine for an instant that Alice used RSA to encrypt the same message to 3 different people, all using the **same "small" public exponent** $e = 3$ as it's common to do. There is an attack, called **Håstad's Broadcast Attack**, that breaks this model.



$$\begin{aligned} c_1 &= m^3 \pmod{N_1} \\ c_2 &= m^3 \pmod{N_2} \\ c_3 &= m^3 \pmod{N_3} \end{aligned}$$

Here the trick is to use the **Chinese Remainder Theorem** to create an equation modulo $N_1 \times N_2 \times N_3$:

$$\begin{aligned} m^3 =& c_1 \cdot N_2N_3 \cdot [(N_2N_3)^{-1} \pmod{N_1}] \\ &+ c_2 \cdot N_1N_3 \cdot [(N_1N_3)^{-1} \pmod{N_2}] \\ &+ c_3 \cdot N_1N_2 \cdot [(N_1N_2)^{-1} \pmod{N_3}] \pmod{N_1N_2N_3} \end{aligned}$$

The method is similar to **Lagrange Interpolation**. For example let me quickly explain the first term, this m^e has to be equal to c_1 only when modulo N_1, so we can multiply the term c_1 by N_2 and N_3 so that it cancels out when modulo N_2 or N_3. But when it is modulo N_1 we don't want those terms, so we multiply our term by their inverse modulo N_1 as well. Easy no ? All the variables are known so calculating $m^e \pmod{N_1N_2N_3}$ is straight forward.
Let's notice that since $m < N_1, m < N_2, m < N_3$, we must have :

$$m \times m \times m = m^3 < N_1 \times N_2 \times N_3$$

So our $m^3 \pmod{N_1N_2N_3}$ is actually just m^3 over $\mathbb{Z}$.
To recover the message we just have to calculate the cubic root of that big value we just calculated.

Generalizing it is pretty easy and let's formulate **Håstad**'s findings :

Theorem 1. *If $c = m^e \pmod{N}$, then we can find m in time polynomial if $|m| < N^{1/e}$.*

That's the introduction to our "small root" problem. Now what about if $|m| > N^{1/e}$ but we know a part of the message m_0 :

$$c = (m_0 + x)^e \pmod{N}$$

Can we efficiently recover x ? That's the question Coppersmith is answering.

5 Coppersmith

This survey is no replacement for the original papers of Coppersmith[2] and Howgrave-Graham[3]. If you want to get a real understanding of those techniques I also advise you to read the survey from May[4].

5.1 Known modulus

That being said, let's dig into Coppersmith's use of LLL to crack RSA. We'll first see one of the problem it solves and build it from there.
Imagine that you know a part of the message : this is called the **Stereotyped Messages Attack**. For example you know that the Alice always sends her messages this way : "the password is : cupcake".
Let's say we know m_0 of the message $m = m_0 + x_0$. And of course we don't know x_0. We have **our problem** translated to the following polynomial :

$$f(x) = (m_0 + x)^e - c \text{ with } f(x_0) = 0 \pmod{N}$$

Well. **Coppersmith** says we can solve this in polynomial time if x_0 and e are small enough :

Theorem 2. *Let N be an integer of unknown factorization, which has a divisor $b \geq N^\beta$, $0 < \beta \leq 1$. Let $f(x)$ be a univariate monic polynomial of degree δ and let $c \geq 1$.*
Then we can find in time $\mathcal{O}(c\delta^5 log^9(N))$ all solutions x_0 of the equation

$$f(x) = 0 \pmod{b} \quad with \quad |x_0| \leq c \cdot N^{\frac{\beta^2}{\delta}}$$

In our case that would mean that for $c = 1$ and $beta = 1$ we could find a solution of our previous equation if $|x_0| \leq N^{\frac{1}{e}}$. And here we find something very similar to Håstad's Broadcast Attack.
To find the roots of a polynomial **over a ring of integers modulo** N is a very **difficult** task, whereas we possess efficient tools to find roots of polynomials **over the integers** (Berlekamp–Zassenhaus, van Hoeij, Hensel lifting...). Hence Coppersmith's intuition to look for such a polynomial :

$f(x_0) = 0 \pmod{N}$ with $|x_0| < X$

$\downarrow$

$g(x_0) = 0$ over $\mathbb{Z}$

But how can we go from f to g here ? The theorem of **Howgrave-Graham** gives us a clue :

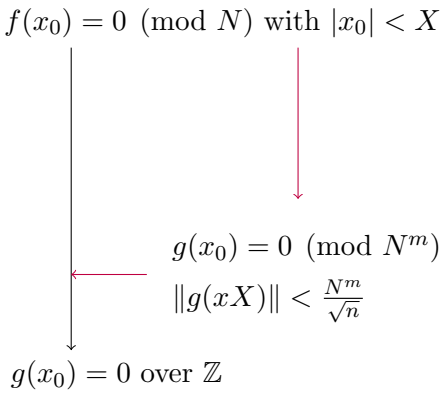
Theorem 3. *Let $g(x)$ be an univariate polynomial with n monomials. Further, let m be a positive integer. Suppose that*

$$g(x_0) = 0 \pmod{N^m} \quad where \quad |x_0| \leq X \tag{1}$$

$$\|g(xX)\| < \frac{N^m}{\sqrt{n}} \tag{2}$$

Then $g(x_0) = 0$ holds over the integers.

What Howgrave-Graham is saying is that we need to **find a polynomial** that shares the same root as our function f but modulo N^m and it has to have **"small" coefficients** so that its norm would be "small" as well.



Howgrave-Graham's idea is that we need to find this polynomial g by **combining** polynomials who also have x_0 as roots. The more polynomials we can play with, the better. We will see later that it is very easy for us to create polynomials f_i such that $f_i(x_0) = 0 \pmod{N^m}$. And that is the reason why we choose to find a polynomial over N^m and not over N.

The **LLL reduction** has two properties that are useful to us :
– It only does **integer linear operations** on the basis vectors
– The **shortest vector of the output basis is bound** (as seen in **Property 1**)

The first point allows us to combine them to build a function that still has x_0 as root modulo N^m :

$$g(x_0) = \sum_{i=1}^{n} a_i \cdot f_i(x_0) = 0 \pmod{N^m} \quad a_i \in \mathbb{Z}$$

The second point allows us to get Howgrave-Graham's second point ($\|g(xX)\| < \frac{b^m}{\sqrt{n}}$).

But first let's see how to **build the polynomials** f_i (we will call them $g_{i,j}$ and h_i) we will build our $g(x_0) = 0$ with. Note that δ is the degree of f :

$$g_{i,j}(x) = x^j \cdot N^i \cdot f^{m-i}(x) \text{ for } i = 0, \ldots, m-1, \;\; j = 0, \ldots, \delta - 1$$
$$h_i(x) = x^i \cdot f^m(x) \text{ for } i = 0, \ldots, t-1$$

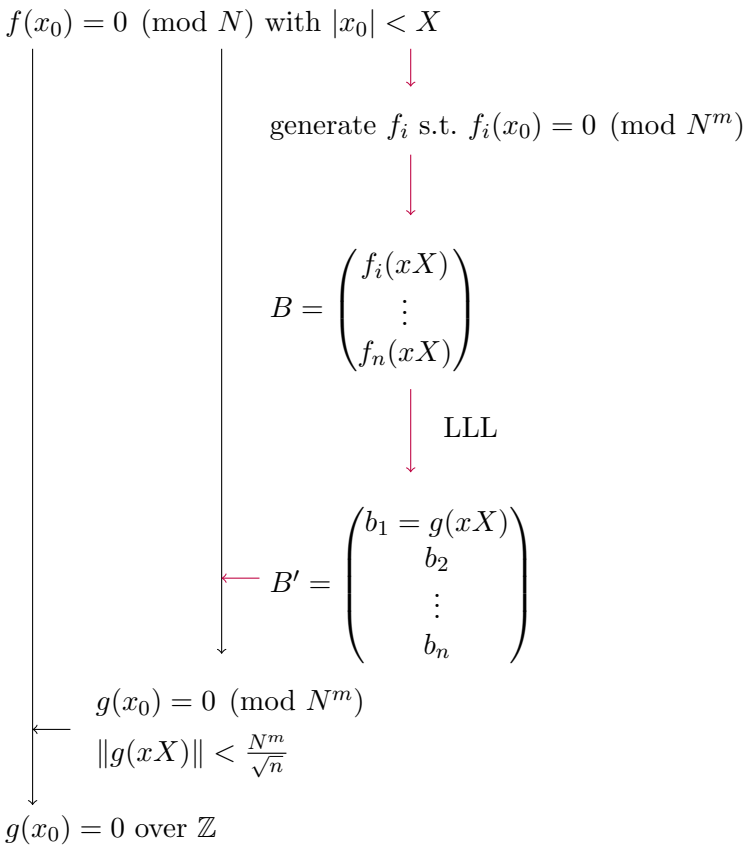
Those polynomials achieve two things :
– they have the **same root** x_0 but modulo N^m

– each iteration introduce a new polynomial. That allows us to build a **triangular** lattice (so that the determinant is easier to calculate)

If you don't understand how they have the same root x_0 remember that since $f(x_0) = 0 \pmod N$ we know that $f(x_0) = k \cdot N$

Now we just have to create a lattice basis with $f_i(xX)$ as row vectors (because we want them to build a polynomial $g(xX)$ to test Howgrave-Graham second point).

Let's take a look at the **overview** again :



Now the shortest vector of B' (the LLL-reduced basis) should be the coefficient vector of $g(xX)$.

As I said earlier, the LLL reduction allows us to achieve an **upper bound** on this shortest vector :

$$\|b_1\| \leq 2^{\frac{n-1}{4}} \cdot det(L)^{\frac{1}{n}}$$

And recall **Howgrave-Graham Theorem**'s second point :

$$\|b1\| = \|g(xX)\| < \frac{N^m}{\sqrt{n}}$$

Now, to obtain Howgrave-Graham's second point on our g we have to manipulate $g_{i,j}(xX)$ and $h_i(xX)$ to obtain a small enough determinant. From the previous equations we **bound the determinant** :

$$det(L) < 2^{-\frac{n(n-1)}{4}} \cdot n^{-\frac{n}{2}} \cdot N^{n \cdot m}$$

The small terms can be considered as "error terms" to **simplify our bound** :

$$det(L) < N^{m \cdot n}$$

It is from these equations that Coppersmith bounded the value of x in his theorem. Now if we want to use this algorithm we will have to **tweak** m and t until we obtain the correct bounds. Note that the bound on the shortest vector of the reduced lattice basis is generous. That means that even if we don't correctly bound our determinant, we might find an answer.

5.2 Any modulus

Coppersmith method is actually more general : it also works for unknown modulus.
We will see how the **Factoring with High Bits Known** attack works to understand this part. Imagine the relaxed problem of RSA where we know an approximation $\tilde{p}$ of one of the prime p. The approximation is bounded as followed :

$$|\tilde{p} - p| < N^{\frac{1}{4}}$$

Now we have an equation with one unknown, modulo another unknown :

$$\tilde{p} = x_0 \pmod{p}$$

This gives us an equation $f(x) = \tilde{p} - x$ such that $f(x_0) = 0 \pmod{p}$. We can use that in the Coppersmith algorithm we have seen earlier. This is because Howgrave-Graham's theorem works for unknown modulus. Let's see this theorem again :

Theorem 4. *Let $g(x)$ be an univariate polynomial with n monomials. Further, let m be a positive integer. Suppose that*

$$g(x_0) = 0 \pmod{b^m} \quad \textit{where} \quad |x_0| \leq X \tag{1}$$

$$\|g(xX)\| < \frac{b^m}{\sqrt{n}} \tag{2}$$

Then $g(x_0) = 0$ holds over the integers.

We know we can build the f_i polynomials as we did before. And here instead of bounding $\|g(xX)\|$ with p^m we can bound it with $N^{\beta m}$ (since we have $p > N^\beta$ in Coppersmith Theorem). This allows us to formulate problems with unknown modulus.
And now obtain a bound on the determinant :

$$det(L) < N^{m \cdot n \cdot \beta}$$

5.3 How were the bounds calculated ?

Let's go back from the start. Here's our new **general equation** with this time an unknown modulus b :

$$f(x) = (m_0 + x)^e - c = 0 \pmod b$$

we know m_0, e, c, N and β. We don't know b, and x. Here are the relations we know :

$$\begin{cases} b \geq N^\beta, \, 0 < \beta \leq 1 \\ |x_0| < X \end{cases}$$

What we want is to find the **biggest possible** X for which it is possible to find the root x_0 of this polynomial f. So we have to find that upperbound X such that there exists m and t to construct a lattice basis of dimensions $n = \delta m + t$ that will yield satisfactory results after a LLL reduction. This happens if we respect **Howgrave-Graham's second point**, that we couple with our **LLL's property** to **bound the determinant** :

$$\left.\begin{aligned} \|g(xX)\| &< \frac{b^m}{\sqrt{n}} \\ \|v_1\| &\leq 2^{\frac{n-1}{4}} det(L)^{\frac{1}{n}} \end{aligned}\right\} \implies 2^{\frac{n-1}{4}} det(L)^{\frac{1}{n}} < \frac{b^m}{\sqrt{n}}$$

Since we don't know b, as I explained earlier we use N^β instead :

$$\implies 2^{\frac{n-1}{4}} det(L)^{\frac{1}{n}} < \frac{N^{\beta m}}{\sqrt{n}}$$

And here's the **determinant** :

$$det(L) = N^{\frac{1}{2}\delta m(m+1)} X^{\frac{1}{2}n(n-1)}$$

We can use both equations to bound X as such :

$$\implies X \leq \frac{1}{2} N^{\frac{2\beta m}{n-1} - \frac{\delta m(m+1)}{n(n-1)}} \tag{1}$$

Earlier I said that we wanted the row vectors in our basis to be **helpful**. Meaning their highest monomial, the one appearing in the diagonal thus in the determinant, had to be less than $N^{\beta m}$. Bounding the last and highest monomial of the diagonal of the basis gives us :

$$X^{n-1} < N^{\beta m} \tag{2}$$

Coppersmith's **constructive proof** showed that using $X = \frac{1}{2}N^{\frac{\beta^2}{\delta}-\epsilon}$ we could satisfy these two previous inequalities for those values :

$$\begin{cases} 0 < \epsilon \leq \frac{1}{7}\beta \\ m = \left\lfloor \frac{\beta^2}{\delta\epsilon} \right\rfloor \\ t = \left\lfloor \delta m(\frac{1}{\beta} - 1) \right\rfloor \end{cases}$$

5.4 Experiments

I used Sage 6.4 in a Virtualbox with 512Mo of RAM and 1 core from an Intel i7 @ 2.30GHz.
Here are the experiments for the **Stereotyped Message Attack** :

size of x_0	size of N	e	m	t	running time
100	512	3	3	0	0.02s
200	1024	3	3	0	0.05s

Here are the experiments for the **Factoring with High Bits Known Attack** :

size of $\lvert p - \tilde{p} \rvert$	size of N	m	t	running time
110	512	4	4	0.01s
200	1024	4	4	0.03s

6 Boneh-Durfee

6.1 Overview of the method

This survey is no replacement for the original papers of Boneh and Durfee[5] and Herrmann and May[6].
We've seen how Coppersmith found a way of using lattices and the LLL algorithm to find small roots to particular univariate polynomials. What about problems that have two unknowns ? Coppersmith gave an heuristic for finding roots of **bivariate polynomials** but left it at that. More recently, Boneh and Durfee have released papers on some RSA attacks that make

use of the initial ideas of Coppersmith for finding small roots of bivariate polynomials.
Let's introduce the problem :
Boneh and Durfee are telling us we can, **most of the time** (they released a heuristic and not a theorem), successfully **factor N** if the **private exponent d is too small**. Precisely if $d < N^{0.292}$.

Recall how RSA works :

$$\begin{aligned} & e \cdot d = 1 \pmod{\varphi(N)} \\ \implies & e \cdot d = k \cdot \varphi(N) + 1 \\ \implies & k \cdot \varphi(N) + 1 = 0 \pmod{e} \\ \implies & k \cdot (N + 1 - p - q) + 1 = 0 \pmod{e} \end{aligned}$$

Here the unknowns are k and $(-p - q)$. We can write that problem as a polynomial with root x_0 and y_0 :

$$f(x, y) = x \cdot (A + y) \text{ such that } f(x_0, y_0) = 0 \pmod{e}$$

with $A = N + 1$ and $y = -p - q$.

Now we use **Coppersmith's heuristic for multivariate polynomials**. Coupled with **Howgrave-Graham's Theorem** for bivariate polynomials :

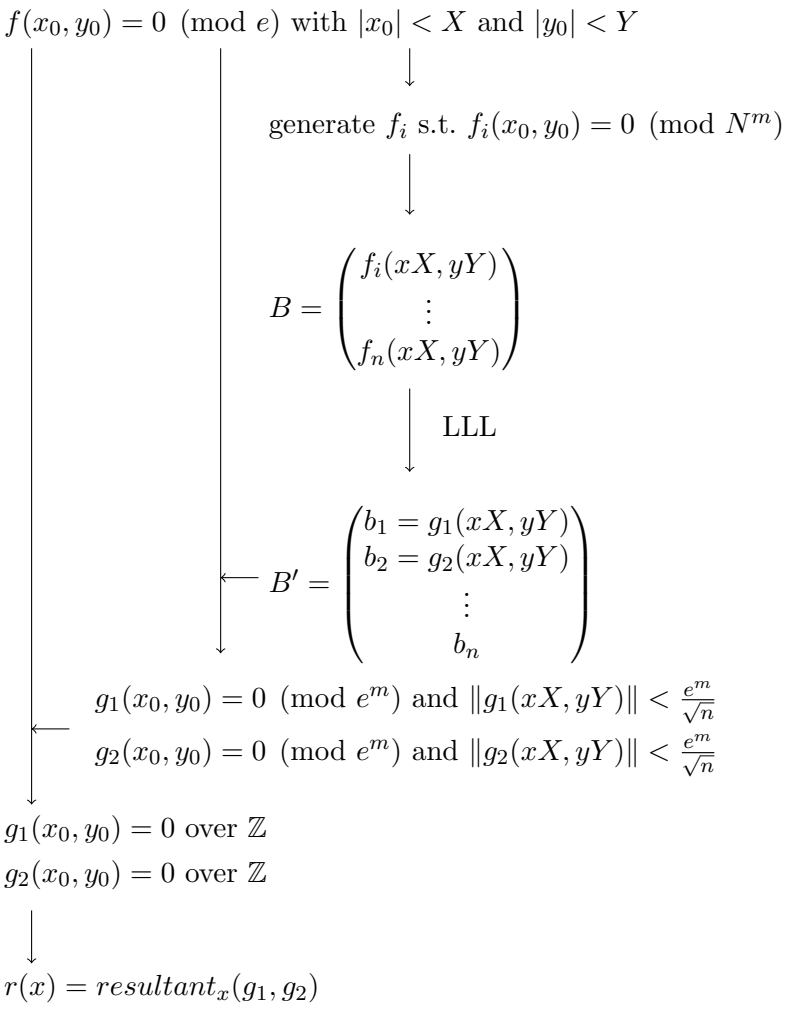
Theorem 5. *Let $g(x)$ be an bivariate polynomial with at most n monomials. Further, let m be a positive integer. Suppose that*

$$g(x_0, y_0) = 0 \pmod{e^m} \quad \textit{where} \quad |x_0| \leq X \textit{ and } |y_0| \leq Y \tag{1}$$

$$\|g(xX, yY)\| < \frac{e^m}{\sqrt{n}} \tag{2}$$

Then $g(x_0, y_0) = 0$ holds over the integers.

But the problem here is that **one polynomial is not enough** to get the roots of a bivariate equation. What we need are **two polynomials**, then we could use the **resultant** or a Gröbner basis to find the roots.
Coppersmith proposed to take **the two shortest vectors**, of the LLL-reduced basis, as polynomials. Let's take a look at what it should look from a distance :



And once we find the root x_0 of r we can re-inject it in g_1 to find y_0.
This doesn't always yield a solution. For example, if the two polynomials g_1 and g_2 are not independent, the resultant will be zero.

Boneh and Durfee proposed a construction of the f_i polynomials as follow :

$$\text{for } k = 0, \ldots, m :$$
$$g_{i,k}(x) = x^i \cdot f^k(x, y) \cdot e^{m-k} \text{ for } i = 0, \ldots, m - k$$
$$h_{j,k}(x) = y^j \cdot f^k(x, y) \cdot e^{m-k} \text{ for } j = 0, \ldots, t$$

They called the $g_{i,k}$ **x-shifts** and the $h_{j,k}$ **y-shifts**.

By using these polynomials to build the lattice, carefully balancing the variables so that the determinant of the triangular basis doesn't exceed e^{mn},

Boneh and Durfee showed that LLL successfuly yielded useful results if $d < N^{0.284}$.

To achieve their **improved results** of $d < N^{0.292}$, they showed that using a sublattice by **ignoring some of the y-shifts**, the bounds on the shortest vectors found by LLL were improved.
This is because of the "**helpful vectors**" notion of Howgrave-Graham. A vector is helpful if his contribution to the determinant (its monomial that appears in the diagonal of the lattice basis) is less than e^m. Boneh and Durfee's method is to discard all y-shifts when their highest monomial exceeds e^m.

	1	x	xy	x^2	x^2y	x^2y^2	y	xy^2	x^2y^3
e^2	e^2								
xe^2		e^2X							
fe	e	eAX	eXY						
x^2e^2				e^2X^2					
xfe		eX		eAX^2	eX^2Y				
f^2	1	$2AX$	$2XY$	A^2X^2	$2AX^2Y$	X^2Y^2			
ye^2						e^2Y			
yfe			$eAXY$				eY	eXY^2	
yf^2			$2AXY$		A^2X^2Y	$2AX^2Y^2$	Y	$2XY^2$	X^2Y^3

Boneh-Durfee basis matrix for $m = 2$, $t = 1$

	1	x	xy	x^2	x^2y	x^2y^2	y	xy^2	x^2y^3
e^2	e^2								
xe^2		e^2X							
fe	e	eAX	eXY						
x^2e^2				e^2X^2					
xfe		eX		eAX^2	eX^2Y				
f^2	1	$2AX$	$2XY$	A^2X^2	$2AX^2Y$	X^2Y^2			
yf^2			$2AXY$		A^2X^2Y	$2AX^2Y^2$	Y	$2XY^2$	X^2Y^3

After removing the damaging y-shifts' coefficient vectors

Unfortunately by doing this we **lose the triangular structure** of the basis and evaluating the determinant of the new rectangular basis is tricky.
Boneh and Durfee developed the notion of **Geometrically progressive matrices** to handle these non-triangular lattice basis. Later **Blömer and May** took a different approach by noticing that some of the columns could be removed without affecting the determinant too much, that allowed the lattice basis to return to a triangular structure. Both those methods are quite difficult to handle and it's more recently that **Herrmann and May** found a clever and better way :

The **Unravelled Linearization** technique consists in **modifying** our initial polynomial f. Done cleverly this will modify the f_i so that after removing the y-shifts, our sublattice basis will **directly be triangular**.
Herrmann and May propose to do the following **substitution** on f :

$$f(x,y) = \underbrace{1+xy}_{u} + Ax \pmod{e}$$

This leaves us with the **linear polynomial** $\bar{f}(u,x) = u + Ax$ (note the lexicographic order, u before x) and a relation $xy = u - 1$.
The x-shifts are still constructed as usual :

$$\bar{g}_{i,k}(u,x) = x^i \cdot \bar{f}^k \cdot e^{m-k} \text{ for } k = 0,\dots,m \text{ and } i = 0,\dots,m-k$$

The y-shifts are constructed the same way, but we need to apply our relation again afterward to completely perform our unravelled linearization :

$$\bar{h}_{j,k}(u,x,y) = y^j \cdot \bar{f}^k \cdot e^{m-k} \text{ for } j = 1,\dots,t \text{ and } k = \left\lfloor \frac{m}{t} \right\rfloor \cdot j,\dots,m$$

They are **selected** with the notion of "**increasing pattern**" in mind, so that using the previous relation $xy = u - 1$ we end up with a **triangular** lattice basis :

$$\begin{array}{c} \\ e^2 \\ xe^2 \\ \bar{f}e \\ x^2e^2 \\ x\bar{f}e \\ \bar{f}^2 \\ y\bar{f}^2 \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & x & u & x^2 & ux & u^2 & u^2y \end{array} \\ \begin{pmatrix} e^2 & & & & & & \\ & e^2X & & & & & \\ & eAX & eU & & & & \\ & & & e^2X^2 & & & \\ & & & eAX^2 & eUX & & \\ & & & A^2X^2 & 2AUX & U^2 & \\ & -A^2X & -2AU & & A^2UX & 2AU^2 & U^2Y \end{pmatrix} \end{array}$$

The same matrix as above after unravelled linearization

Now that we have built a lattice basis, we have to know how we need to **bound our two shortest vectors** so that Howgrave-Graham second point is respected. From LLL's property :

$$\|v_1\| \leq \|v_2\| \leq 2^{\frac{n}{4}} \cdot det(L)^{\frac{1}{n-1}}$$

We need to bound v_1 and v_2 so that they respect Howgrave-Graham's theorem second point. This is the bound we end up with on the determinant :

$$det(L) < \frac{e^{m(n-1)}}{(n2^n)^{\frac{n-1}{2}}}$$

That can be reduced by removing the "error terms" :

$$det(L) < e^{mn}$$

6.2 How was the bound on d calculated ?

Let's go back to our first equation :

$$e \cdot d = k \cdot \varphi N + 1 = k \cdot (N + 1 - p - q)$$

Let's recap what we know :
– p and q should be half the size of N :

$$\frac{log(p)}{log(N)} \approx \frac{1}{2}$$
$$\implies p, q \approx N^{\frac{1}{2}}$$

– $e \approx N$
– $d < N^{\delta}$

The first two are what we usually use in RSA. Our last one is the bound we are trying to define. Of course we want a δ as small as possible. Now that we have defined all these, let's go back to our previous equation and let's bound our unknowns :

$$k \cdot (\underbrace{N+1}_{A} + \underbrace{-p-q}_{s}) \pmod{e} \quad \text{with} \begin{cases} |s| \approx 2N^{\frac{1}{2}} \\ |k| < \frac{N^{\delta}-1}{N-1+2N^{\frac{1}{2}}} \end{cases}$$

s should be pretty close to our prediction. But k ? It could be way lower, this incertitude tells us we have the possibility to play with its bound.
Let's note that $s >> k$ and that reducing the size of our unknown s is a good idea. Notice that both s and A are even, that allows us to reformulate our problem with a smaller s :

$$f(x, y) = x \cdot (A + y) \pmod{e} \quad \text{with} \begin{cases} x = 2k \\ y = (-p-q)/2 \\ A = (N+1)/2 \end{cases}$$

We can now reformulate our bounds and remove the negligible terms :

$$\begin{cases} |y| \approx N^{\frac{1}{2}} \\ |x| < 2N^{\delta} \end{cases}$$

We do not need to bound them accurately as the LLL property on the shortest vector bound is a dramatic one.

Now for our algorithm we need to fix a m and a t such that $dim(L) = n$. For Herrmann and May's selection of the y-shifts we need $m >= t$ so we will write $t = \tau m$ with $\tau < 1$.
Remember earlier we said that we need to bound our determinant to obtain **Howgrave-Graham's Theorem second point**, and **LLL's property** gave us this **bound** :

$$det(L) < e^{mn}$$

$det(L)$ being a function of $e \approx N$, δ, m and t. The bound being a function of e, m and t.
from Herrmann and May's proof we end up with this formula as **determinant** (after removing negligible terms that were calculated for $m \to \infty$) :

$$det(L) = X^{\frac{1}{6}m^3} Y^{\frac{\tau^2}{6}m^3} U^{(\frac{1}{6}+\frac{\tau}{3})m^3} e^{(\frac{1}{3}+\frac{\tau}{2})m^2}$$

with this dimension of the lattice :

$$dim(L) = n = \left(\frac{1}{2} + \frac{\tau}{2}\right) m^2$$

We can now develop the previous bound the determinant, and we fix $t = \tau m$ since $t < m$:

$$X^{\frac{1+\tau}{3}m^3} Y^{\frac{(\tau+1)^2}{6}m^3} e^{(\frac{1}{3}+\frac{\tau}{6})m^3} < e^{(\frac{1}{2}+\frac{\tau}{2})m^2} \text{ for } m \to \infty$$

We then inject $X = e^{\delta}$ and $Y = e^{\frac{1}{2}}$ (since we said $e \approx N$) in our equation :

$$(e^{\delta})^{\frac{1+\tau}{3}m^3} (e^{\frac{1}{2}})^{\frac{(\tau+1)^2}{6}m^3} e^{(\frac{\tau+2}{6})m^3} < e^{\frac{1+\tau}{2}m^2}$$

Boneh and Durfee derived an optimized value of $\tau = (1 - 2\delta)$ that simplifies the inequality to :

$$-\frac{1}{3}\delta^2 + \frac{2}{3}\delta - \frac{1}{6} < 0$$

And thus can be derived the **Boneh-Durfee bound** :

$$\delta < \frac{1}{2}(2 - \sqrt{2}) \approx 0.292$$

6.3 Experiments

My experiments were far from Boneh and Durfee's. When they took hours to solve for small m and t, I took seconds. This is mostly due by the gap of computation power and LLL's implementation between 1999 and now (2015) : I used Sage 6.4 in a Virtualbox with 512Mo of RAM and 1 core from an Intel i7 @ 2.30GHz. I found out that in practice, the bound of X was often way higher than the root x_0, decreasing X until the algorithm worked was a good way to find the roots.

δ	size of N (bits)	size of d (bits)	m	t	dim(L)	running time
0.25	2048	512	3	1	11	0.5s
0.26	2048	532	3	1	11	1.9s
0.27	2048	553	6	2	33	2m 27s

7 Summary

Boneh and Durfee were the last ones to improve the bound on the low private exponent problem. At the end of their paper they claimed that "a bound of $d < N^{1-\frac{1}{\sqrt{2}}}$ cannot be the final answer. It is too unnatural", emitting the idea of a more "natural" bound of $d < N^{\frac{1}{2}}$. It was more than 15 years ago. Coppersmith's ideas of attacking cryptosystems using the lattice reduction algorithm LLL (invented in 1982) seem to still have a lot of room for growing. But at the moment, they are all working in a relaxed version of the problems, which most of the time are far from the reality.
Without even talking about Coppersmith, LLL seems to have a lot of applications in Cryptography and we should see it used more and more during the next years of research.

Références

[1] Chris Peikert *Lattices in Cryptography, Georgia Tech, Fall 2013 : Lecture 2, 3*

[2] Don Coppersmith *Finding Small Solutions to Small Degree Polynomials*

[3] Nicholas Howgrave-Graham *Finding Small Roots of Univariate Modular Equations Revisited*

[4] Alexander May *Using LLL-Reduction for Solving RSA and Factorization Problems*

[5] Boneh and Durfee *Cryptanalysis of RSA with Private Key d Less Than* $N^{0.292}$

[6] Herrmann and May *Maximizing Small Root Bounds by Linearization and Applications to Small Secret Exponent RSA*

```
import time

debug = True

# display matrix picture with 0 and X
def matrix_overview(BB, bound):
    for ii in range(BB.dimensions()[0]):
        a = ('%02d ' % ii)
        for jj in range(BB.dimensions()[1]):
            a += '0' if BB[ii,jj] == 0 else 'X'
            a += ' '
        if BB[ii, ii] >= bound:
            a += '~'
        print a

def coppersmith_howgrave_univariate(pol, modulus, beta, mm, tt, XX):
    """
    Coppersmith revisited by Howgrave-Graham

    finds a solution if:
    * b|modulus, b >= modulus^beta , 0 < beta <= 1
    * |x| < XX
    """
    #
    # init
    #
    dd = pol.degree()
    nn = dd * mm + tt

    #
    # checks
    #
    if not 0 < beta <= 1:
        raise ValueError("beta should belongs in (0, 1]")

    if not pol.is_monic():
        raise ArithmeticError("Polynomial must be monic.")

    #
    # calculate bounds and display them
    #
    """
    * we want to find g(x) such that ||g(xX)|| <= b^m / sqrt(n)

    * we know LLL will give us a short vector v such that:
    ||v|| <= 2^((n - 1)/4) * det(L)^(1/n)

    * we will use that vector as a coefficient vector for our g(x)

    * so we want to satisfy:
    2^((n - 1)/4) * det(L)^(1/n) < N^(beta*m) / sqrt(n)

    so we can obtain ||v|| < N^(beta*m) / sqrt(n) <= b^m / sqrt(n)
    (it's important to use N because we might not know b)
    """
    if debug:
        # t optimized?
        print "\n# Optimized t?\n"
        print "we want X^(n-1) < N^(beta*m) so that each vector is helpful"
        cond1 = RR(XX^(nn-1))
        print "* X^(n-1) = ", cond1
        cond2 = pow(modulus, beta*mm)
        print "* N^(beta*m) = ", cond2
        print "* X^(n-1) < N^(beta*m) \n-> GOOD" if cond1 < cond2 else "* X^(n-1) >= N^(beta*m) \n-> NOT GOOD"

        # bound for X
        print "\n# X bound respected?\n"
        print "we want X <= N^(((2*beta*m)/(n-1)) - ((delta*m*(m+1))/(n*(n-1)))) / 2 = M"
        print "* X =", XX
        cond2 = RR(modulus^(((2*beta*mm)/(nn-1)) - ((dd*mm*(mm+1))/(nn*(nn-1)))) / 2)
```

```
        print "* M =", cond2
        print "* X <= M \n-> GOOD" if XX <= cond2 else "* X > M \n-> NOT GOOD"

        # solution possible?
        print "\n# Solutions possible?\n"
        detL = RR(modulus^(dd * mm * (mm + 1) / 2) * XX^(nn * (nn - 1) / 2))
        print "we can find a solution if 2^((n - 1)/4) * det(L)^(1/n) < N^(beta*m) /
 sqrt(n)"
        cond1 = RR(2^((nn - 1)/4) * detL^(1/nn))
        print "* 2^((n - 1)/4) * det(L)^(1/n) = ", cond1
        cond2 = RR(modulus^(beta*mm) / sqrt(nn))
        print "* N^(beta*m) / sqrt(n) = ", cond2
        print "* 2^((n - 1)/4) * det(L)^(1/n) < N^(beta*m) / sqrt(n) \n-> SOLUTION W
ILL BE FOUND" if cond1 < cond2 else "* 2^((n - 1)/4) * det(L)^(1/n) >= N^(beta*m) /
sqroot(n) \n-> NO SOLUTIONS MIGHT BE FOUND (but we never know)"

        # warning about X
        print "\n# Note that no solutions will be found _for sure_ if you don't resp
ect:\n* |root| < X \n* b >= modulus^beta\n"

    #
    # Coppersmith revisited algo for univariate
    #

    # change ring of pol and x
    polZ = pol.change_ring(ZZ)
    x = polZ.parent().gen()

    # compute polynomials
    gg = []
    for ii in range(mm):
        for jj in range(dd):
            gg.append((x * XX)**jj * modulus**(mm - ii) * polZ(x * XX)**ii)
    for ii in range(tt):
        gg.append((x * XX)**ii * polZ(x * XX)**mm)

    # construct lattice B
    BB = Matrix(ZZ, nn)

    for ii in range(nn):
        for jj in range(ii+1):
            BB[ii, jj] = gg[ii][jj]

    # display basis matrix
    if debug:
        matrix_overview(BB, modulus^mm)

    # LLL
    BB = BB.LLL()

    # transform shortest vector in polynomial
    new_pol = 0
    for ii in range(nn):
        new_pol += x**ii * BB[0, ii] / XX**ii

    # factor polynomial
    potential_roots = new_pol.roots()
    print "potential roots:", potential_roots

    # test roots
    roots = []
    for root in potential_roots:
        if root[0].is_integer():
            result = polZ(ZZ(root[0]))
            if gcd(modulus, result) >= modulus^beta:
                roots.append(ZZ(root[0]))

    #
    return roots

########################################
# Test on Stereotyped Messages
########################################
```

```
print "/////////////////////////////////"
print "// TEST 1"
print "/////////////////////////////////"

# RSA gen options (for the demo)
length_N = 1024  # size of the modulus
Kbits = 200      # size of the root
e = 3

# RSA gen (for the demo)
p = next_prime(2^int(round(length_N/2)))
q = next_prime(p)
N = p*q
ZmodN = Zmod(N);

# Create problem (for the demo)
K = ZZ.random_element(0, 2^Kbits)
Kdigits = K.digits(2)
M = [0]*Kbits + [1]*(length_N-Kbits);
for i in range(len(Kdigits)):
    M[i] = Kdigits[i]
M = ZZ(M, 2)
C = ZmodN(M)^e

# Problem to equation (default)
P.<x> = PolynomialRing(ZmodN) #, implementation='NTL')
pol = (2^length_N - 2^Kbits + x)^e - C
dd = pol.degree()

# Tweak those
beta = 1                                # b = N
epsilon = beta / 7                      # <= beta / 7
mm = ceil(beta**2 / (dd * epsilon))     # optimized value
tt = floor(dd * mm * ((1/beta) - 1))    # optimized value
XX = ceil(N**((beta**2/dd) - epsilon))  # optimized value

# Coppersmith
start_time = time.time()
roots = coppersmith_howgrave_univariate(pol, N, beta, mm, tt, XX)

# output
print "\n# Solutions"
print "we want to find:",str(K)
print "we found:", str(roots)
print("in: %s seconds " % (time.time() - start_time))
print "\n"

##########################################
# Test on Factoring with High Bits Known
##########################################
print "/////////////////////////////////"
print "// TEST 2"
print "/////////////////////////////////"

# RSA gen
length_N = 1024;
p = next_prime(2^int(round(length_N/2)));
q = next_prime( round(pi.n()*p) );
N = p*q;

# qbar is q + [hidden_size_random]
hidden = 200;
diff = ZZ.random_element(0, 2^hidden-1)
qbar = q + diff;

F.<x> = PolynomialRing(Zmod(N), implementation='NTL');
pol = x - qbar
dd = pol.degree()

# PLAY WITH THOSE:
beta = 0.5                              # we should have q >= N^beta
epsilon = beta / 7                      # <= beta/7
```

```
mm = ceil(beta**2 / (dd * epsilon))    # optimized
tt = floor(dd * mm * ((1/beta) - 1))   # optimized
XX = ceil(N**((beta**2/dd) - epsilon)) # we should have |diff| < X

# Coppersmith
start_time = time.time()
roots = coppersmith_howgrave_univariate(pol, N, beta, mm, tt, XX)

# output
print "\n# Solutions"
print "we want to find:", qbar - q
print "we found:", roots
print("in: %s seconds " % (time.time() - start_time))
```

```
import time

debug = True

# display stats on helpful vectors
def helpful_vectors(BB, modulus):
    nothelpful = 0
    for ii in range(BB.dimensions()[0]):
        if BB[ii,ii] >= modulus:
            nothelpful += 1

    print nothelpful, "/", BB.dimensions()[0], " vectors are not helpful"

# display matrix picture with 0 and X
def matrix_overview(BB, bound):
    for ii in range(BB.dimensions()[0]):
        a = ('%02d ' % ii)
        for jj in range(BB.dimensions()[1]):
            a += '0' if BB[ii,jj] == 0 else 'X'
            a += ' '
        if BB[ii, ii] >= bound:
            a += '~'
        print a

def boneh_durfee(pol, modulus, mm, tt, XX, YY):
    """
    Boneh and Durfee revisited by Herrmann and May

    finds a solution if:
    * d < N^delta
    * |x| < e^delta
    * |y| < e^0.5
    whenever delta < 1 - sqrt(2)/2 ~ 0.292
    """

    # substitution (Herrman and May)
    PR.<u, x, y> = PolynomialRing(ZZ)
    Q = PR.quotient(x*y + 1 - u) # u = x*y + 1
    polZ = Q(pol).lift()

    UU = XX*YY + 1

    # x-shifts
    gg = []

    for kk in range(mm + 1):
        for ii in range(mm - kk + 1):
            xshift = x^ii * modulus^(mm - kk) * polZ(u, x, y)^kk
            gg.append(xshift)

    gg.sort()

    # x-shifts monomials
    monomials = []

    for polynomial in gg:
        for monomial in polynomial.monomials():
            if monomial not in monomials:
                monomials.append(monomial)

    monomials.sort()

    # y-shifts (selected by Herrman and May)
    for jj in range(1, tt + 1):
        for kk in range(floor(mm/tt) * jj, mm + 1):
            yshift = y^jj * polZ(u, x, y)^kk * modulus^(mm - kk)
            yshift = Q(yshift).lift()
            gg.append(yshift) # substitution

    # y-shifts monomials
    for jj in range(1, tt + 1):
        for kk in range(floor(mm/tt) * jj, mm + 1):
            monomials.append(u^kk * y^jj)
```

```
    # construct lattice B
    nn = len(monomials)

    BB = Matrix(ZZ, nn)

    for ii in range(nn):

        BB[ii, 0] = gg[ii](0, 0, 0)

        for jj in range(1, ii + 1):
            if monomials[jj] in gg[ii].monomials():
                BB[ii, jj] = gg[ii].monomial_coefficient(monomials[jj]) * monomials[
jj](UU,XX,YY)

    # check if vectors are helpful
    if debug:
        helpful_vectors(BB, modulus^mm)

    # check if determinant is correctly bounded
    if debug:
        det = BB.det()
        bound = modulus^(mm*nn)
        if det >= bound:
            print "We do not have det < bound. Solutions might not be found."
            diff = (log(det) - log(bound)) / log(2)
            print "size det(L) - size e^(m*n) = ", floor(diff)
        else:
            print "det(L) < e^(m*n)"

    # debug: display matrix
    if debug:
        matrix_overview(BB, modulus^mm)

    # LLL
    BB = BB.LLL()

    # vectors -> polynomials
    PR.<x,y> = PolynomialRing(ZZ)

    pols = []
    for ii in range(nn):
        pols.append(0)
        for jj in range(nn):
            pols[-1] += monomials[jj](x*y+1,x,y) * BB[ii, jj] / monomials[jj](UU,XX,
YY)
        if pols[-1](xx,yy) != 0:
            pols.pop()
            break

    # find two vectors we can work with
    pol1 = pol2 = 0
    found = False

    for ii, pol in enumerate(pols):
        if found:
            break
        for jj in range(ii + 1, len(pols)):
            if gcd(pol, pols[jj]) == 1:
                print "using vectors", ii, "and", jj
                pol1 = pol
                pol2 = pols[jj]
                # break from that double loop
                found = True
                break

    # failure
    if pol1 == pol2 == 0:
        print "failure"
        return 0, 0

    # resultant
    PR.<x> = PolynomialRing(ZZ)
```

```
    rr = pol1.resultant(pol2)
    rr = rr(x, x)

    # solutions
    soly = rr.roots()[0][0]
    print "found for y_0:", soly

    ss = pol1(x, soly)
    solx = ss.roots()[0][0]
    print "found for x_0:", solx

    #
    return solx, soly


########################################
# Test
########################################

# RSA gen options (tweakable)
length_N = 2048
length_d = 0.27

# RSA gen (for the demo)
p = next_prime(2^int(round(length_N/2)))
q = next_prime(round(pi.n()*p))
N = p*q
phi = (p-1)*(q-1)
d = int(N^length_d)
if d % 2 == 0: d += 1
while gcd(d, phi) != 1:
    d += 2
e = d.inverse_mod((p-1)*(q-1))

# Problem put in equation (default)
P.<x,y> = PolynomialRing(Zmod(e))
A = int((N+1)/2)
pol = 1 + x * (A + y)

# and the solutions to be found (for the demo)
yy = (-p -q)/2
xx = (e * d - 1) / (A + yy)

#
# Default values
# you should tweak delta and m. X should be OK as well
#
delta = 0.27              # < 0.292 (Boneh & Durfee's bound)
X = 2*floor(N^delta)      # this _might_ be too much
Y = floor(N^(1/2))        # correct if p, q are ~ same size
m = 7                     # bigger is better (but takes longer)
t = int((1-2*delta) * m)  # optimization from Herrmann and May
# Checking bounds (for the demo)
print "=== checking values ==="
print "* |y| < Y:", abs(yy) < Y
print "* |x| < X:", abs(xx) < X
print "* d < N^0.292", d < N^(0.292)
print "* size of d:", int(log(d)/log(2))

# boneh_durfee
print "=== running algorithm ==="
start_time = time.time()
solx, soly = boneh_durfee(pol, e, m, t, X, Y)

# Checking solutions (for the demo)
if xx == solx and yy == soly:
    print "\n=== the solutions are correct ==="
else:
    print "=== FAIL ==="

# Stats
print("=== %s seconds ===" % (time.time() - start_time))
```