mimoo / RSA-and-LLL-attacks

implementations of attacks on RSA through LLL reductions

171 commits | 1 branch | 0 releases | 1 contributor

Branch: master ▾ RSA-and-LLL-attacks / +

fixed url to survey

mimoo authored on 15 Aug — latest commit 51ca069591

.gitignore	nom nom nom	8 months ago
README.md	fixed url to survey	a month ago
boneh_durfee.sage	merged with the experimental branch	5 months ago
coppersmith.sage	cleaning	7 months ago
rapport.tex	fixes	6 months ago
research.sage	comment	5 months ago
slides.pdf	slides done	6 months ago
slides.pptx	slides done	6 months ago
survey_final.pdf	fix	6 months ago

README.md

Lattice based attacks on RSA

This repo will host implementations and explanations of different RSA attacks using **lattice reduction** techniques (in particular **LLL**).

First, we'll see how **Coppersmith** found out that you could use lattice reduction techniques to attack a relaxed model of RSA (we know parts of the message, or we know parts of one of the prime, ...). And how **Howgrave-Graham** reformulated his attack.

Second we'll see how **Boneh and Durfee** used a coppersmith-like attack to factor the RSA modulus when the private key is too small (`d < N^0.929`). Followed by a simplification from **Herrman and May**.

If you want to use the implementations, see below for explanations on Coppersmith and Boneh-Durfee. If you want to dig deeper you can also read my survey or watch my video.

I've also done some personal researches on the Boneh-Durfee algorithm and I do get better results. Check `research.sage` (note: most was merged in `boneh_durfee.sage` , just use `helpful_only = True` to make use of it)

Coppersmith

I've implemented the work of **Coppersmith** (to be correct the reformulation of his attack by **Howgrave-Graham**) in coppersmith.sage.

I've used it in two examples in the code:

Stereotyped messages

For example if **you know the most significant bits of the message**. You can **find the rest of the message** with this method.

The usual RSA model is this one: you have a ciphertext `c` a modulus `N` and a public exponent `e`. Find `m` such that `m^e = c mod N`.

Now, this is the **relaxed model** we can solve: you have `c = (m + x)^e`, you know a part of the message, `m`, but you don't know `x`. For example the message is always something like "*the password today is: [password]*". Coppersmith says that if you are looking for `N^1/e` of the message it is then a `small root` and you should be able to find it pretty quickly.

let our polynomial be `f(x) = (m + x)^e - c` which has a root we want to find `modulo N`. Here's how to do it with my implementation:

```
dd = f.degree()
beta = 1
epsilon = beta / 7
mm = ceil(beta**2 / (dd * epsilon))
tt = floor(dd * mm * ((1/beta) - 1))
XX = ceil(N**((beta**2/dd) - epsilon))
roots = coppersmith_howgrave_univariate(f, N, beta, mm, tt, XX)
```

You can play with the values until it finds the root. The default values should be a good start. If you want to tweak:

- beta is always 1 in this case.
- `XX` is your upper bound on the root. **The bigger is the unknown, the bigger XX should be**. And the bigger it is... the more time it takes.

Factoring with high bits known

Another case is factoring `N` knowing high bits of `q`.

The Factorization problem normally is: give `N = pq`, find `q`. In our **relaxed** model we know an approximation `q'` of `q`.

Here's how to do it with my implementation:

let `f(x) = x - q'` which has a root modulo q

```
beta = 0.5
dd = f.degree()
epsilon = beta / 7
mm = ceil(beta**2 / (dd * epsilon))
tt = floor(dd * mm * ((1/beta) - 1))
XX = ceil(N**((beta**2/dd) - epsilon)) + 100000000000000000000000000000000
roots = coppersmith_howgrave_univariate(f, N, beta, mm, tt, XX)
```

What is important here if you want to find a solution:

- we should have `q >= N^beta`
- as usual `XX` is the upper bound of the root, so the difference should be: |diff| < X

note: `diff = |q-q'|`

Boneh Durfee

The implementation of **Boneh and Durfee** attack (simplified by **Herrmann and May**) can be found in boneh_durfee.sage.

The attack allows us to break RSA and the private exponent `d`. Here's why RSA works (where `e` is the public exponent, `phi` is euler's totient function, `N` is the public modulus):

```
    ed = 1 mod phi(N)
=> ed = k phi(N) + 1 over Z
=> k phi(N) + 1 = 0 mod e
=> k (N + 1 - p - q) + 1 = 0 mod e
=> 2k [(N + 1)/2 + (-p -q)/2] + 1 = 0 mod e
```

The last equation gives us a bivariate polynomial `f(x,y) = 1 + x * (A + y)`. Finding the roots of this polynomial will allow us to easily compute the private exponent `d`.

The attack works if the private exponent `d` is too small compared to the modulus: `d < N^0.292`.

To use it:

1. look at the **how to use** section at the end of the file in boneh_durfee.sage and replace the values according to your problem: the variable `delta` is your hypothesis on the private exponent `d`. If you don't have `d < N^delta` you will not find solutions. Start small (delta = 0.26) and increase slowly (maximum is `0.292`)
2. Run the program. If you get an error: "Try with highers m and t" you should increase `m`. The more you increase it, the longer the program will need to run. Increase it until you get rid of the error.
3. If you do not want to increase `m` (because it takes too long for example) you can try to decrease `X` because it happens that it is too high compared to the root of the `x` you are trying to find. This is a last recourse tweak though.
4. If you still don't find anything for high values of `delta`, `m` and `t` then you can try to do an exhaustive search on `d > N^0.292`. Good luck!
5. Once you found solutions for `x` and `y`, you can easily insert them into the equation:

```
e d = x [(N + 1)/2 + y] + 1
```

The example in the code should be clear enough, there is also a write-up of a CTF challenge using this code.

PS: You can also try to use `research.sage`. It tries to remove *unhelpful vectors* when it doesn't break the triangular form of the lattice's basis. It might help you to use a lower `m` than necessary!