**Lecture 4**
**Coppersmith, Cryptanalysis**

Instructor: Chris Peikert
Scribe: Jacob Alperin-Sheriff

## 1 Coppersmith's Theorem

Today we prove the "full version" of Coppersmith's Theorem, stated here.

**Theorem 1.1.** *Let $N$ be a positive integer and $f(x) \in \mathbb{Z}[x]$ be a monic, degree-$d$ polynomial. There is an algorithm that, given $N$ and $f$, efficiently (i.e., in time polynomial in the bit length of the inputs) finds all integers $x_0$ such that $f(x_0) = 0 \bmod N$ and $|x_0| \leq B \approx N^{1/d}$.*

In the theorem statement and what follows, for simplicity of analysis we use $\approx$ to hide factors which are polynomial in $d$ and $N^{\varepsilon}$ for some arbitrarily small constant $\varepsilon > 0$.[1] The remainder of this section is dedicated to the proof of the theorem.

Last time we considered adding multiples of $N \cdot x^i$ to $f(x)$, which preserves the roots of $f(x)$ modulo $N$. But this only let us obtain a bound of $B \approx N^{2/d^2}$. To do better, we consider higher powers of $f(x)$ and $N$. That is, our strategy will be to use LLL to efficiently find a nonzero polynomial $h(x) = \sum_i h_i x^i \in \mathbb{Z}[x]$ of degree at most $n = d(m+1)$, for some positive integer $m$ to be determined, such that:

1. Any mod-$N$ root of $f$ is a mod-$N^m$ root of $h$, i.e., if $f(x_0) = 0 \bmod N$ then $h(x_0) = 0 \bmod N^m$.
2. The polynomial $h(Bx)$ is "short," i.e., its coefficients $h_i B^i$ are all less than $N^m/(n+1)$ in magnitude.

From the second property it follows that if $|x_0| \leq B$, then

$$|h(x_0)| \leq \sum_i |h_i B^i| < N^m.$$

Therefore, by the first property, any small mod-$N$ root of $f$ is a root of $h(x)$ *over the integers* (not modulo anything). So, having found $h(x)$, we can efficiently factor it over the integers and check whether each of its small roots is indeed a root of $f(x)$ modulo $N$.

To construct a lattice basis that lets us find such an $h(x)$, the first helpful insight is that $f(x_0) = 0 \bmod N$ implies $f(x_0)^k = 0 \bmod N^k$ for any positive integer $k$. To create our lattice basis, we define $n = d(m+1)$ polynomials $g_{u,v}(x)$ whose mod-$N^m$ roots will include all of the mod-$N$ roots of $f(x)$. Concretely,

$$g_{u,v}(x) = N^{m-v} f(x)^v x^u, \quad \text{for } u \in \{0, \ldots, d-1\}, v \in \{0, \ldots, m\}.$$

We use two important facts about these polynomials. First, $f(x)$ is monic and of degree $d$, so $g_{u,v}(x)$ has leading coefficient $N^{m-v}$ and is of degree exactly $u + vd$. Second, if $x_0$ is a mod-$N$ root of $f(x)$, then $x_0$ is a mod-$N^m$ root of $g_{u,v}(x)$ for all $u, v$, because $N^m$ divides $N^{m-v} f(x_0)^v$.

The basis vectors for our lattice are defined by the coefficients of $g_{u,v}(Bx)$ where, as above, $B$ corresponds to the bound on the absolute value of the roots. Specifically, the basis is $\mathbf{B} = [\mathbf{b}_0, \ldots, \mathbf{b}_{n-1}]$, where $\mathbf{b}_{u+vd}$ is the coefficient vector of $g_{u,v}(Bx)$ represented as a polynomial in $x$. Since $g_{u,v}(x)$ is of degree $u + vd$ with leading coefficient $N^{m-v}$, and $u + vd$ runs over $\{0, \ldots, n-1\}$ as $u, v$ run over their respective ranges, the basis is triangular, with diagonal entries $N^{m-v} B^{u+vd}$. A simple calculation then reveals that

$$\det(\mathbf{B}) = B^{n(n-1)/2} \cdot N^{dm(m+1)/2}.$$

Running the LLL algorithm on $\mathbf{B}$ yields a nonzero vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ of length

$$\begin{aligned}\|\mathbf{v}\| \leq 2^{(n-1)/2} \det(\mathbf{B})^{1/n} &= 2^{(n-1)/2} \left(B^{n(n-1)/2} \cdot N^{dm(m+1)/2}\right)^{1/n} \\ &\leq (2B)^{n/2} \cdot N^{m/2}.\end{aligned}$$

[1] This yields a true bound of $B = N^{1/d-\varepsilon}$, though with more work the theorem can be proved for $B = N^{1/d}$.

Setting $B \approx (N^{1/d})^{m/(m+1)}$, which is $N^{1/d-\varepsilon}$ for large enough (but still polynomially bounded) $m$, we can ensure that $(2B)^{n/2} < N^{m/2}/(n+1)$ and therefore that $\|\mathbf{v}\| < N^m/(n+1)$, as required. Reading off the entries of $\mathbf{v}$ as the coefficients of $h(Bx)$ yields a satisfactory polynomial $h(x)$.

## 2 Cryptanalysis of RSA Variants

One of the most interesting applications of Coppersmith's algorithm is to attack variants of RSA.

### 2.1 RSA Recap

The RSA function and cryptosystem (named after its inventors Rivest, Shamir and Adleman) is one of the most widely used public-key encryption and digital signature schemes in practice.

It is important to differentiate between the RSA *function* and an RSA-based cryptosystem. The RSA function is defined as follows. Let $N = pq$, where $p$ and $q$ are distinct very large primes (according to current security recommendations, they should be at least 1,000 bits each). Let $e$ be a public "encryption exponent" such that $\gcd(e, \varphi(N)) = 1$, where $\varphi(N) = |\mathbb{Z}_N^*| = (p-1)(q-1)$ is the totient function. The RSA function $f_{e,N} \colon \mathbb{Z}_N^* \to \mathbb{Z}_N^*$ is defined by the public values $(N, e)$ as follows:

$$f_{e,N}(x) := x^e \bmod N.$$

It is conjectured that, given just $N, e$, and $y = f_{e,N}(x)$ for a uniformly random $x \in \mathbb{Z}_N^*$, it is computationally infeasible to find $x$. However, one can efficiently invert the RSA function using some extra "trapdoor" information. Let $d$ be the "decryption exponent" defined by $e \cdot d = 1 \bmod \varphi(N)$, which is efficiently computable given the factorization of $N$. Then the inverse RSA function is defined as

$$f_{e,N}^{-1}(y) := y^d \bmod N.$$

This indeed inverts the function because, for $y = f_{e,N}(x) = x^e \bmod N$, we have

$$y^d = (x^e)^d = (x^{e \cdot d \bmod \varphi(N)}) = x \bmod N,$$

where the first inequality holds because the order of the group $\mathbb{Z}_N^*$ is $\varphi(N)$. It follows that $f_{e,N}$ is a bijection (also called a *permutation*) on $\mathbb{Z}_N^*$.

### 2.2 Low-Exponent Attacks

Because exponentiation modulo huge integers is somewhat computationally expensive, many early proposals advocated using encryption exponents as small as $e = 3$ with RSA, due to some significant efficiency benefits.[2] For example, using an encryption exponent of the form $e = 2^k + 1$, one can compute $x^e$ using only $k + 1$ modular multiplications via the standard "repeated squaring" method. Moreover, the basic RSA function still appears to be hard to invert (on uniformly random $y \in \mathbb{Z}_N^*$) for small $e$.

Like any other deterministic public-key encryption scheme, the RSA function itself is not a secure cryptosystem. This is because if an attacker had a guess about the message in a ciphertext $c$ (say, because the number of possible messages was small), it could easily test whether its guess was correct by encrypting the message and checking whether the resulting ciphertext matched $c$. Therefore, one needs to use a *randomized*

[2]Note that one cannot use $e = 2$ (or any other even $e$) because it is never coprime with $\varphi(N)$, which is also even.

encryption algorithm. The most obvious way to randomize the RSA function is to append some "padding" of random bits to the end of the message before encrypting. However, devising a secure padding scheme is quite difficult and subtle, due to the existence of clever attacks using tools like LLL. Here we describe one such attack: when $e$ is small and the padding is not long enough, the attack efficiently recovers an encrypted message $M$ given just two encryptions of $M$ with different paddings.[3]

We start with a simple "related-message attack," which is interesting in its own right and will also be a key component of the padding attack. Note that this attack is on the deterministic version of the RSA function.

**Lemma 2.1.** *Let $e = 3$ and $M_1, M_2 \in \mathbb{Z}_N^*$ satisfy $M_1 = f(M_2) \bmod N$, where $f(x) = ax + b$ for some $b \neq 0$ is a known linear function. Then one can efficiently recover $M_1, M_2$ from $C_1 = M_1^e \bmod N$ and $C_2 = M_2^e \bmod N$ (and the other public information $N, e, f$).*[4]

*Proof.* Define polynomials $g_1(x) = f(x)^e - C_1, g_2(x) = x^e - C_2 \in \mathbb{Z}_N[x]$. Notice that $M_2$ is a root of both $g_1$ and $g_2$, and thus $(x - M_2)$ is a common divisor of $g_1$ and $g_2$. If it is in fact the greatest common divisor (i.e., the common divisor of largest degree), then we can find it using Euclid's algorithm.[5] We show next that $(x - M_2)$ is indeed the greatest common divisor.

First observe that for any $C \in \mathbb{Z}_N^*$, the polynomial $x^3 - C$ has only one root modulo $N$, because the RSA function is a bijection. As a result, we have $g_2(x) = x^3 - C_2 = (x - M_2)(x^2 + \beta_1 x + \beta_0)$, where the quadratic term is irreducible, so $\gcd(g_1, g_2)$ is either $(x - M_2)$ or $g_2$. Since $b \neq 0$, we have that $g_2(x) \nmid g_1(x)$, and therefore the greatest common divisor is indeed $x - M_2$. $\square$

We now consider a potential padding method: to encrypt a message $M$, one appends $m$ uniformly random bits $r \in \{0, 1\}^m$ (for some value of $m$), and applies the RSA function:

$$C = f_{N,e}(M \| r).$$

(Upon decryption, the pad bits are ignored.) Mathematically, this corresponds with transforming $M$ to $2^m \cdot M + r$ for uniformly random $r \in \{0, \ldots, 2^m - 1\}$. Note that $M$ must be short enough, and $m$ small enough, so that the result can be interpreted as an element of $\mathbb{Z}_N^*$ and unambiguously represents $M$.

The following theorem shows that if the pad length is too short (as determined by the size of $e$), then it is possible to efficiently recover the message $M$ from two distinct encryptions of it. Notice that for $e = 3$, a pad length of $n/9 \approx 2{,}000/9 \approx 222$ might seem quite large, yet it provides essentially no security.

**Lemma 2.2.** *Let $N$ have bit length $n$, and let $m \leq \lfloor n/e^2 \rfloor$ be the bit length of the pad. Given two encryptions $C_1, C_2$ of the same message $M$ with arbitrary distinct pads $r_1, r_2 \in \{0, \ldots, 2^m - 1\}$, one can efficiently recover $M$.*

*Proof.* We have $M_1 = 2^m \cdot M + r_1$ and $M_2 = 2^m \cdot M + r_2$ for some distinct $r_1, r_2 \in \{0, \ldots, 2^m - 1\}$. We define two bivariate polynomials $g_1(x, y), g_2(x, y) \in \mathbb{Z}_N[x]$ as

$$\begin{aligned} g_1(x, y) &= x^e - C_1 = x^e - M_1^e \\ g_2(x, y) &= (x + y)^e - C_2 = (x + y)^e - M_2^e \end{aligned}$$

[3]In real-life applications it is quite common to encrypt the same message more than once, e.g., via common protocol headers or retransmission.

[4]It turns out that theorem is "usually" true even for $e > 3$, but there are rare exceptions.

[5]Strictly speaking, Euclid's algorithm would normally requires $\mathbb{Z}_N$ to be a field, which it is not. However, if Euclid's algorithm fails in this setting then it reveals the factorization of $N$ as a side effect, which lets us compute the decryption exponent and the messages.

Essentially, $x$ represents the unknown message, and $y$ represents the unknown pads. Since $g_1$ is independent of $y$, we have that $(x = M_1, y = \star)$ is a root of $g_1$ for any value of $y$. Similarly, $(x = M_1, y = r_2 - r_1)$ is a root of $g_2$.

To take the next step, we need a concept called the *resultant* of two polynomials in a variable $x$, which is defined as the product of all the differences between their respective roots:

$$\mathrm{res}_x(p(x), q(x)) = \prod_{p(x_0)=q(x_1)=0} (x_0 - x_1).$$

In our setting, we treat the bivariate polynomials $g_i(x, y)$ as polynomials in $x$ whose coefficients are polynomials in $y$ (i.e., elements of $\mathbb{Z}_N[y]$), which is why $\mathrm{res}_x(g_1, g_2)$ is a polynomial in $y$.

We use a few important facts about the resultant. First, it is clear that $\mathrm{res}_x(p(x), q(x)) = 0$ when $p, q$ have a common root. Second, $\mathrm{res}_x(p, q) = \det(\mathbf{S}_{p,q})$, where $\mathbf{S}_{p,q}$ is a square $(\deg p + \deg q)$-dimensional matrix called the Sylvester matrix, whose entries are made up of various shifts of the coefficient vectors of $p$ and $q$. Therefore, the resultant can be computed efficiently. Finally, in our setting the $x$-coefficients of $g_1$ are degree-0 polynomials in $y$, while the $x$-coefficients of $g_2$ are polynomials of degree at most $e$ in $y$. By definition of the Sylvester matrix, the resultant $h(y) = \mathrm{res}_x(g_1, g_2)$ has degree at most $e^2$ in $y$.

We claim that $\Delta = r_2 - r_1 \neq 0$, which has absolute value $|\Delta| \leq 2^m < N^{1/e^2}$, is a root of the resultant $h(y) = \mathrm{res}_x(g_1, g_2)$. This is because the univariate polynomials $g_1(x, \Delta)$ and $g_2(x, \Delta)$ have a common root $x = M_1$.

Armed with this information, our attack proceeds as follows. We construct the polynomials $g_1, g_2$, and compute the resultant $h(y) = \mathrm{res}_x(g_1, g_2)$. Then $\deg(h(y)) \leq e^2$, and we know that $h(y)$ has $\Delta = r_2 - r_1 \neq 0$ as a root modulo $N$. We run Coppersmith's algorithm on $h(y)$, and since $|\Delta| \leq 2^m < N^{1/e^2}$, we get a polynomial-length list containing $\Delta$. Trying each element of the list as a candidate $\Delta$, we have a known (candidate) linear function $f(x) = x - \Delta$ such that $M_1 = f(M_2)$, and we can run the related message attack from Lemma 2.1. One of these runs involves the correct value of $\Delta$ and thus reveals $M_1, M_2$, and we can confirm which run does so by re-encrypting and checking against $C_1, C_2$. $\square$

The above is just one of countless Coppersmith-style attacks against variants of RSA and problems related to factoring, such as:

- Small decryption exponent $d$: so far the best known attack recovers $d$ if it is less than $N^{.292}$. This uses a bivariate version of Coppersmith that lacks a rigorous proof of correctness, but seems to work well in practice. Important open questions are whether $d < N^{1/2-\varepsilon}$ is attackable (the conjecture is that it should be), and whether there are rigorously provable variants of Coppersmith for bivariate or multivariate polynomials.
- Partial secret key exposure: when certain bits of $d$ or the factors $p, q$ of $N$ are exposed, it is often possible to recover them completely.