CS 198 Codebreaking at Cal
Spring 2023 Course Notes

Note 6

Discrete Logarithm Problem

Normally, when one wants to recover the value of an exponent given the base and final value (say, 32 with base 2 is $\log_2(32) = 5$), it is simple to just take the logarithm of that number. Over finite fields (in our case, modular arithmetic $\mod n$), however, it becomes considerably harder. This is due to the fact that we have possibly 'wrapped around' the modulus, much like how we can't just take square or cube roots easily either. This problem is called the discrete logarithm problem.

Definition 1: Discrete Logarithm Problem

The **discrete logarithm problem** over the finite field $\mathbb{F}_N$ is the problem of finding x in the equation $g^x = h \pmod{N}$, given g and N. It is considered computationally intractable for large values of N.

Why is the discrete log important? We rely on the discrete logarithm problem (hereafter DLP) being hard to compute in order to ensure the security of Diffie-Hellman, alongside other cryptosystems.

Recall our Diffie-Hellman system in which Alice encrypts $g^a \mod p$ and Bob encrypts $g^b \mod p$ using their secret values of a and b. Were Eve able to compute the discrete log, she could find a and b and construct $g^{ab} \pmod{n}$ herself, breaking our system.

The current record for discrete logarithm is of a 1024-bit prime (very large number), so it is not impossible to break, just very expensive. The authors of Logjam estimated that a discrete log over a 1024-bit prime could cost roughly $100MM, which is certainly feasible for nation-state actors. This is why most systems use at least 2048-bit, and sometimes higher. There exist various methods of "cherrypicking" easy/hard primes for discrete logarithm as well, as we will see later.

Let's consider what a naive implementation of a discrete log finding algorithm may look like. We can try $g^k \mod n$ for $k = 1, 2, 3, \ldots n-1$, for a running time exponential in the number of bits of n.

We can do better – but not significantly so.

Baby Step Giant Step

Baby Step Giant Step is a (relatively) efficient algorithm for computing the discrete logarithm. BSGS runs in $O(\sqrt{N})$ time for a modulus N.

Consider some $m = \lceil \sqrt{n} \rceil$. For any $x \in [0, n-1]$, we can use the Division Algorithm to write it as $x = am + b$ for some integers $a, b < m$.

Rewriting our discrete log equation of $g^x \equiv h \mod n$:

$$\begin{aligned} g^x &\equiv h \mod n \\ g^{am+b} &\equiv h \mod n \\ g^b &\equiv h(g^{-(am)}) \mod n \\ g^b &\equiv h(g^{-m})^a \mod n \end{aligned}$$

If this equivalence holds for some a, b, then we can reconstruct our x! Therefore, we just need to try all possible a and b combinations.

First, for the "baby step", we find and store all g^b for $0 \leq b < m$ into some efficient hash table that gives us constant-time lookup. The key will be the g^b and the value will be b.

Next, for the "giant step", we compute $h(g^{-m})^a$ for $0 \leq a < m$. This can be made more efficient by precomputing $c = g^{-m}$ (it is constant) and repeatedly multiplying h by c in a for loop from 0 to m. At each step, check whether the current value is a key in the earlier hash table. If so, we've found our match and can immediately return $am + b$.

Integer Factorization

First, let's revisit the fact that factoring the RSA modulus N is equivalent to compromising the entire system. If the attack was able to find either p or q, they can find the other via division. Once they have p, q, they can compute d just like the original keyholder did ($e^{-1} \mod (p-1)(q-1)$).

Consider the naive implementation of a factorization algorithm:

```python
def naiveFactorization(N: int) -> List:
    k = 2 # Our factor
    factors = set()
    while k <= math.sqrt(N): # Only need to go up to sqrt(N) for this
        if (N % k == 0):
            factors.add(k)
            factors.add(N//k)
        k+=1
    return list(factors)
```

This takes in an integer N and returns its factors. The complexity of this algorithm is thus $O(\sqrt{N})$, where N is the number.

You might think this is a polynomial-time algorithm, but it is only psuedo-polynomial (polynomial in terms of the value), whereas it is exponential in the number of bits.

Consider an n-bit number. Taking its square root reduces it to $\frac{n}{2}$ bits, as $\sqrt{2^n} = 2^{\frac{n}{2}}$. For every extra bit added, we have $\sqrt{2}$ more operations than before. Therefore, the algorithm is $O(\sqrt{2}^b)$ for a b-bit number, or exponential time.

Put more intuitively, every extra 2 bits double the search space of the algorithm, so factoring even 512 bit numbers becomes unfeasible extremely quickly. (This would be 2^{256} operations, equivalent to just guessing an AES-256 key, which would take billions of years).

The best known integer factorization algorithm on a classical (non-quantum) computer is the general number field sieve, which is sub-exponential but only practical for numbers larger than 10^{100} or so. (Don't worry about this algorithm, it's out of scope).

Pollard's Rho Algorithm

We are lucky that this is considered hard for large, correctly selected values of N. But what about smaller, less carefully selected ones? Say an amateur cryptographer was trying to send a message via RSA and chose N to be some very larger prime multiplied by a small prime. In that case, it is very efficient to factor by Pollard's rho algorithm.

The algorithm utilizes a key fact of modular arithmetic, in which $k \equiv 0 \pmod{N}$ implies that

$$\begin{aligned} k &\equiv 0 \pmod{n_1} \\ k &\equiv 0 \pmod{n_2} \\ &\vdots \\ k &\equiv 0 \pmod{n_j} \end{aligned}$$

for $N = \prod_{i=0}^{j} n_i$ (each n_i is a prime factor of N). This means that if a number is a multiple of the modulus, it is also a multiple of every prime factor.

Exercise: Prove this for yourself!

Knowing this, if we have two numbers k_1 and k_2 such that $|k_1 - k_2| \equiv 0 \pmod{N}$, then $|k_1 - k_2|$ is a multiple of the prime factors of N. Remember that we can efficiently take the GCD of two numbers using Euclid's algorithm. If $|k_1 - k_2|$ is a multiple of n_1, then $gcd(|k_1 - k_2|, n_1) = n_1$. We've just factored the integer altogether, as we can now find the rest of the integers by division or repeating the algorithm.

So, if we manage to find k_1 and k_2 such that $\gcd(|k_1 - k_2|, N) \neq 1$, we will have successfully factored our integer. You might now be wondering how this is any better than the naive implementation, as we have to guess these numbers anyway. The key lies in the fact that we only need to find a duplicate mod n_1, since if $k_1 \equiv k_2 \mod n_1$, then $|k_1 - k_2|$ is a multiple of n_1, and $\gcd(|k_1 - k_2|, N) = n_1$.

Therefore we our "sample space" is really just as big as the smallest factor. By the birthday paradox, we expect to hit a duplicate in $O(\sqrt{n_1})$ guesses (off by a small constant factor).

To run the actual algorithm, we basically find two random numbers, check if $\gcd(|k_1 - k_2|, N) \neq 1$, and repeat if not – otherwise return $\gcd(|k_1 - k_2|, N)$ as our non-trivial factor.

In practice, this algorithm is extremely adept at factoring numbers of the form $N = pq$ for a small p and large q.

There also exists an algorithm by the name of **Pollard's p-1 algorithm** that is especially good at factoring numbers with many small prime factors. Said algorithm is out of scope, but closely related.