Lecture 26: Randomized Fingerprinting

Comparing Databases

Alice and Bob are far apart, say, in Australia and Brazil respectively. Each has a copy of a database (of n bits), a and b respectively. They want to check that their copies are the same, i.e., that $a = b$. However, they have only a (costly) telephone line between them over which they can communicate.

Obviously Alice could send her entire database a to Bob, and he could compare then it to b. But this requires transmission of n bits, which for realistic values of n is costly and error-prone. Instead, suppose Alice first computes a much smaller fingerprint $\mathbb{F}(a)$ and sends this to Bob. He then computes $\mathbb{F}(b)$ and compares it with $\mathbb{F}(a)$. If the fingerprints are equal, he announces that the copies are identical.

What kind of properties do we want the fingerprint function to have? Clearly, $\mathbb{F}(x)$ should be much shorter than the length of x. Further, we would like $\mathbb{F}(a)$ and $\mathbb{F}(b)$ to be the same if a and b are identical, and different otherwise. In the following discussion, we will try to construct one such function so that if $a \neq b$, $\mathbb{F}(a) \neq \mathbb{F}(b)$ with constant probability.

Let's view a copy of the database as an n-bit binary number, i.e., $a = \sum_{i=0}^{n-1} a_i 2^i$ and $b = \sum_{i=0}^{n-1} b_i 2^i$. Now define $\mathbb{F}(a) = a \bmod p$, where p is a prime number chosen at random from the range $\{1, \ldots, k\}$, for some suitable k. Remember that we want $\Pr[\mathbb{F}(a) = \mathbb{F}(b)]$ to be small if $a \neq b$.

Suppose $a \neq b$. When is $\mathbb{F}(a) = \mathbb{F}(b)$? Well, for this to happen, we need that $a \bmod p = b \bmod p$, i.e., that p divides $d = a - b \neq 0$. But d is an (at most) n-bit number, so the size of d is less than 2^n. This means that at most n different primes can divide d.

Why is this so? Recall that we can write d as the product of its prime factors $p_1 \cdot p_2 \cdot \ldots \cdot p_t$. Since each of these primes must be at least 2, and d is at most 2^n, it must be the case that $t \leq n$, and so at most n primes divide d.

So: as long as we make k large enough so that the number of primes in the range $\{1, \ldots, k\}$ is much larger than n we will be in good shape. To ensure this, we use the *Prime Number Theorem* from Lecture 10.

Theorem 26.1: *[Prime Number Theorem] Let $\pi(k)$ denote the number of primes less than k. Then $\pi(k) \sim \frac{k}{\ln k}$ as $k \to \infty$.*

Now all we need to do is set $k = cn\ln(cn)$ for any c we like. By the Prime Number Theorem, with this choice of k,

$$\Pr[\mathbb{F}(a) = \mathbb{F}(b) | a \neq b] \leq \frac{n}{\pi(k)} \sim \frac{1}{c}.$$

So, if we take $c = 10$, say, then we will achieve an error probability less than $1/10$.

Finally, note that Alice only needs to send to Bob the numbers $a \bmod p$ and p (so that Bob knows which fingerprint to compute), both of which are at most k. So the number of bits sent by Alice is at most $2\log_2 k = O(\log n)$.

Reducing the error: Note that the protocol above is guaranteed to work 90% of the time. But this is not very good for crucial applications – we would like our error probability to be much smaller. One way to do this is to *perform independent trials*. I.e., we repeat the protocol many times independently of each other and answer that the databases are the same if and only if $\mathbb{F}(a) = \mathbb{F}(b)$ in all the runs of the protocol.

What is the error probability if we repeat t times? Well, we make an error when $a \neq b$ and $\mathbb{F}(a) = \mathbb{F}(b)$ on all the runs. But the chance of this happening is at most $(1/c)^t$. If $c = 10$, and we make $t = 100$, the chance of error is at most $(1/10)^{100}$, which is incredibly small.

We did not explain how Alice selects a random prime $p \in \{1,\dots,k\}$. This can be done, as in the case of RSA, by choosing a random number in $\{1,\dots,k\}$, testing if it is prime, and if not, throwing it away and trying again.

Randomized Pattern Matching

Consider the classical problem of searching for a pattern Y in a string X. I.e., we want to know whether the string $Y = y_1 y_2 \dots y_m$ (of length m) occurs as a contiguous substring of $X = x_1 x_2 \dots x_n$ (which is of length n). This kind of operation is regularly done in almost every piece of software used, e.g., word-processors, web browsers and databases. Furthermore, it is used frequently enough that finding fast ways of doing this have been studied for a long time.

The naïve approach of trying every possible match takes $O(nm)$ time. (Why?) There is a rather complicated deterministic algorithm that runs in $O(n+m)$ time (which is clearly best possible, since just reading the two strings takes $O(m+n)$ time). A beautifully simple randomized algorithm, due to Karp and Rabin, also runs in $O(n+m)$ time and is based on the same idea as in the above example.

Let us assume (for simplicity) that the alphabet is binary. Let $X(j) = x_j x_{j+1} \dots x_{j+m-1}$ denote the substring of X of length m starting at position j. For example, if $X = 101100001$ and $m = 5$, then $X(2) = 01100$.

Algorithm Match-String

pick a random prime p in the range $\{1,\dots,k\}$
for $j = 1$ **to** $n-m+1$ **do**
    if $X(j) = Y \bmod p$ **then** report match and stop

Note that the test in the **if**-statement here is the same as checking if $\mathbb{F}(X(j)) = \mathbb{F}(Y)$, for the same fingerprint function $\mathbb{F}$ as in the Alice and Bob problem in the previous section.

If the algorithm runs to completion without reporting a match, then Y definitely does not occur in X. To see this, note that if Y occured in X, then $Y = X(i)$ for some i, and hence the algorithm would have reported a match at the i-th step. Hence the only error the algorithm can make is to report a false match.

What is the probability that this happens? By the same analysis as above, for each j if $X(j) \neq Y$ then

$$\Pr[\mathbb{F}(X(j)) = \mathbb{F}(Y)] \leq \frac{m}{\pi(k)}.$$

This is the error probability at the j-th step. Therefore we can bound the chance that the algorithm makes an error in its entire run by applying the union bound. In fact,

$$\Pr[\text{algorithm reports a match}] \leq \frac{nm}{\pi(k)} \sim \frac{1}{c}$$

if we choose $k = cnm\ln(cnm)$ (exactly as before). As before, we can choose c to be, say, 10 to make the error probability at most $1/10$.

What about the running time? Well, in a simple implementation, each iteration of the loop requires the computation of a fingerprint of an m-bit number, which takes $O(m)$ time, giving a total running time of $O(nm)$. This seems to imply that we are doing worse than the naïve implementation, since we take the same time after doing more involved things, and now we even have the possibility of making errors!

However, things can be substantially improved by noticing that

$$\mathbb{F}(X(j+1)) = 2\Big(\mathbb{F}(X(j)) - 2^{m-1}x_j\Big) + x_{j+m} \bmod p.$$

(Check this.) So, given $X(j)$, computing $X(j+1)$ requires one subtraction, one multiplication (or a shift), one addition, and finally taking remainders modulo a prime p. Hence, under the realistic assumption that arithmetic operations on fingerprints — which are small — can be done in constant time, each iteration actually takes only <u>constant</u> time (except the first iteration, which takes $O(m)$ time to compute $\mathbb{F}(X(1))$ and $\mathbb{F}(Y)$). Thus the overall running time of the algorithm is therefore $O(n+m)$ as claimed earlier, and the error probability is a constant.

We can again make repeated runs of the algorithm to make the error probability vanishingly small. However, in practice, we would want to eliminate the possibility of false matches entirely. To do this, we could make the algorithm <u>test</u> any match before reporting it. If it is found to be a false match, the algorithm could simply restart with a new random prime p. The resulting algorithm never makes an error.

Ex: Show that the expected running time of this new algorithm is at most $\frac{c}{c-1}T \approx T$, where T is the running time of the original algorithm, and that the probability it runs for at least $(\ell+1)T$ time is at most $c^{-\ell}$.