6 Section Week 6 - Hashing and Randomized Storage

This lecture assumes knowledge of what a probability distribution is.

First, let's finish up from last week. We talked about pseudorandom generators, and these are fine for extending randomness, but in many cryptographic applications they are insufficient. In particular, we defined the security of the PRG when given the uniform distribution as input, but we may care about security when the input is not uniform.

For example, consider any PRG $P : \{0,1\}^n \to \{0,1\}^{n+1}$ and construct another PRG P' as follows: $P'(0^n) = 0^{n+1}$, and $P'(x) = P(x)$ for all other $x \in \{0,1\}^n$. We can easily show that P' is a PRG, since P is a PRG and if we see some polynomial $p(n)$ samples from our PRGs then the behavior of P' is identical to that of P with probability $(1 - \frac{1}{2^n})^{p(n)} \approx e^{-\frac{p(n)}{2^n}} = 1 - o(\frac{1}{p(n)})$. However, if the algorithm is allowed to choose inputs, we can simply plug in 0^n and distinguish between P' and U_{n+1} with high probability.

6.1 PRFs

This leads us to the definition of a PRF, which is as follows.

A PRF has a key $k \in \{0,1\}^\ell$ and is $\{0,1\}^n \to \{0,1\}$ (later we'll talk about length extension). A PRF F is secure if for all efficient algorithms A,

$$\left| \Pr_{k \leftarrow_R \{0,1\}^\ell}[A^{F_k(\cdot)}(1^n) = 1] - \Pr_{H \leftarrow_R \{0,1\}^n \to \{0,1\}}[A^{H(\cdot)}(1^n) = 1] \right| < \epsilon(\ell)$$

Where H is a fully randomly chosen function and $F_k(\cdot)$ and $H(\cdot)$ denote oracle access to F_k and H, respectively. In other words, for a randomly seeded PRF, for any efficient algorithm that (possibly adaptively) chooses inputs to test it on, the PRF looks like a random function with security dependent on the length of the seed.

It turns out that the length extension ideas work here too but with more

work, and the result is that we can get PRFs with any desired polynomial output length.

To construct a PRF $\{0,1\}^n \to \{0,1\}^n$ from a PRG, suppose we have a $PRG : \{0,1\}^n \to \{0,1\}^{2n}$. We can use this to build a depth n binary tree, where the top node is labelled by the PRF key and at each level we apply the PRG to the label above and split the output in two to get the two labels of the children.

Draw the tree for this

6.2 Hash Functions

Now we get to hash functions, which are a common term in CS but are often confused theoretically. The most important first point is that HASH FUNCTIONS ARE NOT necessarily random looking. The definition of a hash function is:

A family $\mathcal{H}$ of functions is a collision resistant family of efficiently computable hash functions $h_k : \{0,1\}^* \to \{0,1\}^n$ parameterized by $k \in \{0,1\}^n$, where for any efficient algorithm A,

$$\Pr_{k \leftarrow_R \{0,1\}^n}[A(k) = (x, x') \text{ s.t. } h_k(x) = h_k(x')] < \epsilon(n)$$

In other words, even given the random key, no efficient algorithm can find a collision.

Some things to note here: nothing is secret about this construction. If we compare to the PRFs from before, we allow everyone to see everything going on in the hash function, but it is still hard to find a collision given k.

Now clearly we cannot have an actual one to one map from $\{0,1\}^*$ to $\{0,1\}^n$, so for a k there must exist x, x' s.t. $h_k(x) = h_k(x')$. So what about the algorithm that hardcodes and outputs these (x, x')? The answer is because that particular value of k is only chosen $\frac{1}{2^n}$ of the time. Then what about

an algorithm that has a lookup table, stores a pair for every k, and binary searches for the correct pair? I'm not actually sure; I think this would technically violate the definition of an efficient algorithm, since the algorithm uses exponential storage space or have to take in some advice or if we're talking about circuits you need to store the entire table in gates which would be not time efficient.

Another observation is that by the birthday paradox if our algorithm just tries $2^{n/2}$ queries of a fully random function then with constant probability we find a collision, and collisions in this sense are easier to find than inputs with a specified output. Thus usually hash functions have double the security parameters of other cryptographic protocols.

Now consider this non random looking hash function. Suppose we have any $H : \{0,1\}^{4n} \to \{0,1\}^n$. We can construct a $H' : \{0,1\}^{4n} \to \{0,1\}^{2n}$ by simply saying $H'(x) = 0^n \circ H(x)$. This does not look random at all but has the same collision resistance as H.

Now we are going to sweep some of the hash formalisms under the rug to present two cool data structures.

6.3 Merkle Trees

Here's a cool application of hash functions. Suppose you have some huge data set that you can't keep on your own computer and you want to store it on Dropbox, but you want to make sure Dropbox is actually storing your thing. Here's a way to do verified virtual storage.

First, suppose we have a collision resistant hash family and restrict its inputs to get a family of functions $\{0,1\}^{2n} \to \{0,1\}^n$. Now suppose our file is $2^k n$ bits long for some large k. We can build the k height Merkle hash tree as follows:

Draw a Merkle tree

Note that after we hash all the way down the result is a single string of length

n which we call the data's Merkle hash. What happens is Dropbox takes your gigantic data set, computes the Merkle hash, and sends it to you.

Now suppose you want some piece of your data. Dropbox sends the block containing what you need as well as all of the adjacent children in the Merkle tree, for a total of $O(kn)$ bits. Call this the Merkle proof of storage. Using all of this info, you can use the hash function to verify that everything they sent back hashes to your data's Merkle hash.

Security is given by the collision freeness of our hash function. In particular, let H be our hash function and consider the Merkle security game where Dropbox lies and tries to send you the wrong block of data. Then (informally) the starting hash inputs are different but end up the same, so somewhere the Merkle proof of storage goes from incorrect to correct. Say the correct hash output at this point is some z, the correct input is some x, and the incorrect input is y. We know that $H(x) = z$ by the original construction of the tree, but since we go from incorrect to correct it must also be the case that $H(y) = z$, so we have a collision.

Proof by picture

Thus, if we have an algorithm that can efficiently lie about your Merkle data query, then the same algorithm can find a collision in our hash function.

(Omitting full proof for brevity)

6.4 Bloom Filters

Bloom filters well studied by Michael Mitzenmacher here at Harvard!

Let us now make a simplifying assumption: we have a hash family $\mathcal{H}$ of functions $h_1, \ldots h_k$ where each h_i is a fully random function $U \to [n]$ for some universe U and integer n. This is a reasonable assumption to make and is an example of "wishful thinking to guide future work"; for now we'll roll with it and later discuss how this does in practice.

Now we wish to solve the problem: I have some set $S \subset U$, where $|S| = m$. I

will send you the elements of S one time, and then make queries of the form "Is q an element of S?" If you only have an n bit array as your memory and the hash functions, how can you succeed with high probability?

The idea is as follows. Initialize your array to all 0s. For each element s of S, when you receive it you compute $h_1(s), h_2(s), \ldots h_i(s), \ldots h_k(s)$ and set the bits at these indices to be 1 (if it was already a 1, just keep it to be a 1).

Now suppose I make some query q. To test if q was in S, simply compute $h_1(q), h_2(q), \ldots h_i(q), \ldots h_k(q)$ and check all of those indices. If all of the indices are 1, then output 1, otherwise output 0.

If q is in S then clearly we must output 1, since when we put the contents of S into our bloom filter we also put q into our bloom filter. Thus, we have no false negatives.

Clearly it is possible to have false positives, so we wish to compute the probability that each of $h_i(q)$ are set to 1.

We first compute the probability that the bit at some index j was set to a 1. We had a total of km hashes, so the probability that every single one of them missed j is $\left(1 - \frac{1}{n}\right)^{km}$, and the probability that j was set to 1 is $1 - \left(1 - \frac{1}{n}\right)^{km} \approx 1 - e^{-km/n}$. Thus, the probability that all k of the indices $h_1(q), h_2(q), \ldots h_i(q), \ldots h_k(q)$ are set to 1 is $\left(1 - e^{-km/n}\right)^k$.

After some calculus, we get that the false positive rate is minimized when $k = \frac{n}{m}(\ln 2)$ and the false positive rate is $(0.6185)^{n/m}$ (this is worked out in Mitzenmacher and Upfal).

The main advantage here is that it doesn't matter how ridiculously complex each item in the universe is to represent - as long as the ratio of our space and the subset of the universe are constant from each other, we get membership testing with constant error (consider for contrast each item in the universe requires 2^n bits to represent as the universe has 2^{2^n} items total).

Going back to our assumption about the hash functions, in Ramakrishna '89 they show that "perfect transformations with a filter size of 46,152 bits yield

the same performance as the actual transformations with 65,536 bits," so we're already close to getting the theoretical performance, an as we improve in our hash technology our bloom filters will get correspondingly better.