ALGORITHMICS, CRYPTOGRAPHY & COMPUTER SECURITY, graduate programs 2023
CRYPTOGRAPHY, 2022, assignments list # 1, 7.3.2023

1. Given a good hashing function $H : \{0,1\}^k \longrightarrow \{0,1\}^k$ that behaves like a random oracle. However, we need a function F that maps a sequence of 2^m bit strings of length k to a string of length k and also behaves as a random oracle.

   The following construction has been proposed: we create a tree with 2^m leaves. We label the leaves with the bit strings from the input. Then we label the internal nodes in the following way: a node a with children b and c gets the label $\mathsf{label}(a) := H(\mathsf{label}(b)) \oplus H(\mathsf{label}(c))$, where $\oplus$ stands for bitwise XOR operation. The label of the root of the tree is the value of F.

   Is it a sound construction? If not, then how to improve it?

2. A hash function must mimic a random oracle. Assume that we are given H_1 and H_2 designed independently which are claimed to be pretty good as a hash function.

   An intuitive way to create an even better function is to define $H(x) = H_1(H_2(x))$. Is it a good idea? Does H inherit all good properties of H_1 and H_2? Is it better than H_1 or H_2 alone?

   Check all properties that we have mentioned. Consider also the cardinality of the image of H.

3. Assume that you have a random oracle $\mathcal{O}$ that returns k bit numbers. However, what you need are the m bit numbers with exactly t ones. Present a procedure based on $\mathcal{O}$ that mimics a random oracle that returns only numbers with this property.

4. Consider a random walk performed on an array D of size 2^k, where each entry is a bit generated uniformly at random. If i is the current position, then in one step the pointer jumps to the position $\mathsf{Hash}(D[i]..D[i+m]])$. What is the probability that all the positions of D will be visited by the pointer when it starts at the position 0? (An estimation is enough.) What about visiting, say, at least half of the positions?

5. For authentication based on secret key K, we may have the problem of leaking the key to third parties. There might be different ways of such leakage, e.g. via delays in communication.

   To protect ourselves against such leakage, we use a very large key K – say, 100MB key. Then leaking it bit by bit would take years. But how to use a large key so that:

   - the adversary has a negligible chance to impersonate if he knows the bits of K on a fraction of 50% positions,
   - computing the authentication token (sent from the prover to the verifier) is very easy and fast.

   (Hint: we are going to make use of the results of Question 4.)

6. Frequently, we consider the distinguishability concept: The game is that we have a black box that hides either a random oracle $\mathcal{O}$ or a hash function H (each option with probability 0.5). Alice wins if she correctly guesses what is inside after sending a number of queries to the black box. We say that H is indistinguishable from $\mathcal{O}$, if Alice wins with probability $\frac{1}{2} + \epsilon$, where ϵ is negligibly small for any efficient strategy A (say, A is a polynomial-time algorithm).

   One common approach to evaluate candidates for good hash functions is to create a strategy called *distinguisher*, so that Alice using the distinguisher would win the above game with probability $\frac{1}{2} + \epsilon$, where ϵ is NOT negligible.

   Is the following sentence true:

function H has all the properties discussed in the lecture iff there is no distinguisher for H

Consider separately the $\Rightarrow$ and $\Leftarrow$ implications.

7. There is a good example of useful data structures based on hash functions: DHT structures (see also Bloom filters). To insert a file F this data structure we compute $a_1 := \mathsf{Hash}_1(ID(F))$ and $a_2 := \mathsf{Hash}_2(ID(F))$, then insert F at the address a_1 or a_2, whichever place is free. In order to fetch F with the identifier $ID(F)$ we simply look at the locations a_1 and a_2.

   Estimate the probability of failure to insert F into the DHT structure if already n files have been inserted and the addresses have k bits.

   A Cuckoo filter is an improvement of the above construction. If both places for F are already occupied, then we take F' already stored in, say, position a_1 and move it to its alternative position, while placing F in a_1. Of course, the alternative location for F' might already be taken, so we repeat the same procedure.

   Try examples to see how good this improvement is. Try to estimate analytically what the improvement is.

/-/ Mirosław Kutyłowski