CS426 Combinatorial Group Testing Needed for Project Spring 2013

Question. Suppose you have n data items $d_1, \ldots, d_n$. You also have a *checksum* function H that takes as input an x that is an item or a concatenation of items, and produces a k-bit output $H(x)$ such that any modification to the input x will almost certainly cause a different output $H(x)$. Note that x may be huge, but $H(x)$ is always k bits long. A commonly used method for detecting whether an item or concatenation of items x has undergone an unauthorized modification is to compute its checksum $H(x)$ and store it in a safe place; later on, the integrity of x can be verified by computing $H(x)$ and comparing it to the stored checksum value (if they do not match then x has been corrupted). Suppose that you are told that at most one of the n items can be corrupted, and you are asked to store enough checksum values to make it possible to pinpoint which d_i (if any) is the corrupted one. This can obviously be achieved by storing in a safe place each of the n checksum values $H(d_1), \ldots, H(d_n)$. On the other hand, storing only one checksum value, such as

$$H(d_1||d_2||\ldots||d_n)$$

where "||" denotes concatenation, would detect *whether* there is corruption of an item or not, but would not be enough to pinpoint which item is the corrupted one. We need a scheme where we store only $1 + \log n$ checksum values, that allow us to pinpoint which item has been corrupted. For simplicity, we assume n is a power of 2.

Answer. One of the checksums is for the concatenation of all the items – it serves to determine whether there is corruption or not. The remaining $\log n$ checksums are for determining which one of the items is corrupted, and are as follows. For $j = 1, 2, \ldots, \log n$, the jth checksum is for the concatenation of those d_i for which the integer i has a 1 in the jth least significant bit of its binary representation; i.e., an item d_i is in the jth checksum if, in the binary representation of the integer i, the jth least significant bit is a 1.

To determine which d_i is corrupted, the binary representation of integer i is constructed one bit at a time, as follows: For $j = 0, \ldots, \log n - 1$ in turn, if the jth computed checksum matches the stored checksum then the jth bit of i is 0, and if it does not match then the bit is 1.

To illustrate, consider the case of 2000 items $d_1, d_2, \ldots, d_{2000}$, of which only one is corrupted (assume it is the item d_{676}). In that case the checksum of the concatenation of all the 2000 items does not match its expected value, which indicates that there is a corruption. The other 11 ($= \log n$) auxiliary checksums reveal which item is corrupted, as follows. The 11-bit binary representation of 676 is 01010100100, and the item d_{676} is therefore a part of the checksums for bit positions 2, 5, 7, 9, and it is not a part of the checksums for bit positions 0, 1, 3, 4, 6, 8, 10. The four checksums that contain d_{676} will not match their expected values, whereas the other seven checksums will match their expected values. This implies that the corrupted item d_i has (i) a 1 in bit positions 2, 5, 7, and 9, of the 11-bit binary representation of i; and (ii) a 0 in bit positions 0, 1, 3, 4, 6, 8, 10 (otherwise the 7 corresponding checksums would have matched their expected values).

Note 1. Of course replacing each occurrence of "least significant bit" with "most significant bit" in the above, would still work.

Note 2. In situations where the checksums are stored in the same place as the data, the checksum function is a hash-based message authentication code (HMAC), whose computation requires a secret key. The purpose of the key is to prevent an adversary from first carrying out a modification, and then replacing the checksums with newly computed ones that are consistent with the new value (thereby making the malicious modification undetectable). But no secret key is needed if the hashes are stored in a secure place that is separate from the data, and in that case a cryptographic hash function would have been enough. It is not safe to use an H that is a non-cryptographic hash function, because it would then be possible to make modifications that cause no change in the value of the hash.