Sudipta Halder (2021202011)

Build a provably secure PRF from PRG

Below is the proof of If the function G is a pseudorandom generator with expansion factor l(n) = 2n, then how can we make PRF from that PRG.

THEOREM 6.22 *If the function G is a pseudorandom generator with expansion factor $\ell(n) = 2n$, then Construction 6.21 is an efficiently computable pseudorandom function.*

PROOF (Sketch) The intuition behind the proof of this theorem follows from the motivating examples given above. Namely, the hybrid distribution provided for the case of a 2-bit input can be extended to inputs of length n. This extension works by just continuing to build the binary tree of Figure **??** (note that the case of 1 input bit gives the root and the next level of the tree and the case of 2 input bits extends this to an additional level of the tree). The reason why it is possible to continue extending the construction is that if the intermediate result is pseudorandom, then it can be replaced by truly random strings, that can then be used as seeds to the pseudorandom generator once again.

The actual proof of the theorem works by a hybrid argument (see the proof of Theorem 6.20), and we only sketch it here. We define a hybrid random variable H_n^i to be a full binary tree of depth n where the nodes of levels 0 to i are labelled with independent truly random values, and the nodes of levels $i+1$ to n are constructed as in Construction 6.21 (given the labels of level i). We note that in H_n^i, the labels in nodes 0 to $i-1$ are actually irrelevant. The function associated with this tree is obtained as in Construction 6.21 by outputting the appropriate values in the leaves.

Notice that H_n^n is a truly random function, because all of the leaves are given truly random and independent values. On the other hand, H_n^0 is exactly Construction 6.21 (because only the key is random and everything else is pseudorandom, as in the construction). Using a standard hybrid argument as made in the proof of Theorem 6.20, we obtain that if a polynomial-time distinguisher D can distinguish Construction 6.21 from a truly random function with non-negligible probability, then there must be values i for which H_n^i can be distinguished from H_n^{i+1} with non-negligible probability. We use this to distinguish the pseudorandom generator from random. Intuitively this follows because the only difference between the neighboring hybrid distributions H_n^i and H_n^{i+1} is that in H_n^{i+1} the pseudorandom generator G is applied one more time on the way from the root to the leaves of the tree. The actual proof is more tricky than this because we cannot hold the entire $(i+1)^{\text{th}}$ level of the tree (it may be exponential in size). Rather, let $t(n)$ be the maximum running-time of the distinguisher D who manages to distinguish Construction 6.21 from a random function. It follows that D makes at most $t(n)$ oracle queries to its oracle function. Now, let D' be a distinguisher for G that receives an input of length $2n \cdot t(n)$ that is either truly random or $t(n)$ invocations of $G(s)$ with independent random values of s each time. (Although we have not shown it here, it is not difficult to show that all of these samples together constitute a pseudorandom string of length $2n \cdot t(n)$.) Then, D' chooses a random $i \leftarrow \{0, \ldots, n-1\}$ and answers D's oracle queries as follows, initially holding an empty binary tree. Upon receiving a query $x = x_1 \cdots x_n$ from D, distinguisher D' uses $x_1 \cdots x_i$ to reach a node on the i^{th} level (filling all values to that point with arbitrary values – they are of no consequence). Then, D' takes one of its input samples (of length $2n$) and labels the left son of the reached node with the first half of the sample and the right son with the second half of the sample. D' then continues to compute the output as in Construction 6.21. Note that in future queries, if the input x brings D' to a node that has already been filled, then D' answers consistently to the value that already exists there. Otherwise, D' uses a new sample from its input. (Notice that D' works by filling the tree dynamically, depending on D's queries. It does this because the full tree is too large to hold.)

The important observations are as follows:

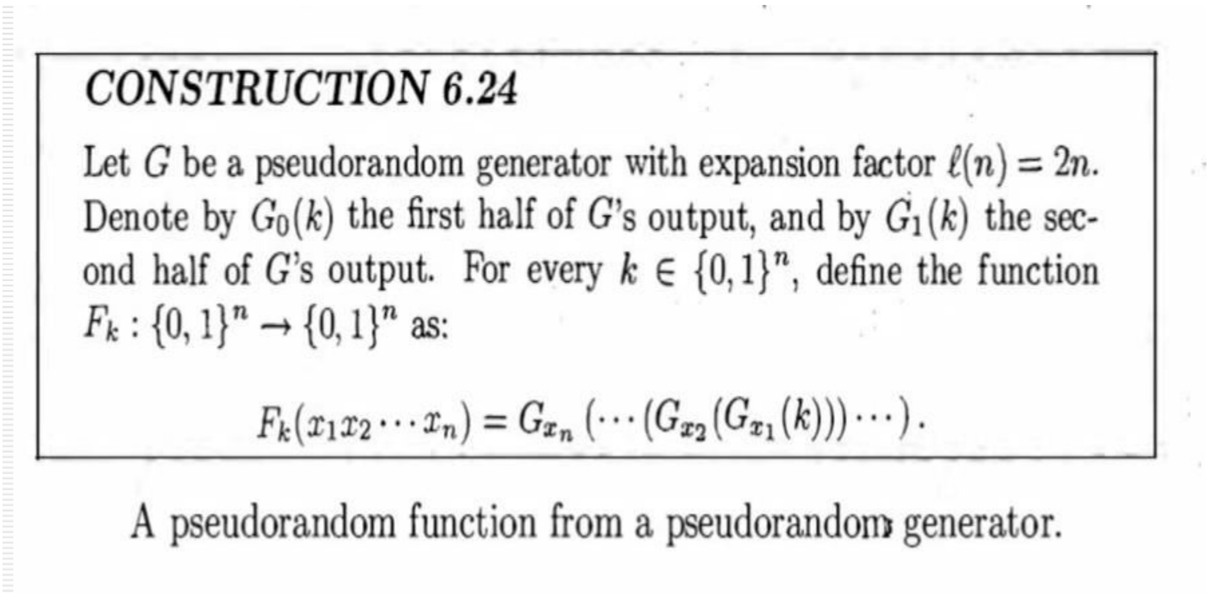
1. If D' receives a truly random string of length $2n \cdot t(n)$, then it answers D' exactly according to the distribution H_n^{i+1}. This holds because all the values in level $i+1$ in the tree that are (dynamically) constructed by D' are random.

2. If D' receives pseudorandom input (i.e., $t(n)$ invocations of $G(s)$ with independent values of s each time), then it answers D' exactly according to H_n^i. This holds because the values in level $i+1$ are pseudorandom and generated by G, exactly as defined. (Notice that the seeds to these pseudorandom values are not known to D' but this makes no difference to the result.)

By carrying out a similar hybrid analysis as in the proof of Theorem 6.20, we obtain that if D distinguishes Construction 6.21 from a truly random function with non-negligible probability $\varepsilon(n)$, then D' distinguishes $t(n)$ invocations of $G(s)$ from a truly random string of length $2n \cdot t(n)$ with probability $\varepsilon(n)/n$. Since $\varepsilon(n)$ is non-negligible, this contradicts the assumption that G is a pseudorandom generator. ■

[1]

Now, let's see the construction of PRF from length doubling PRG.

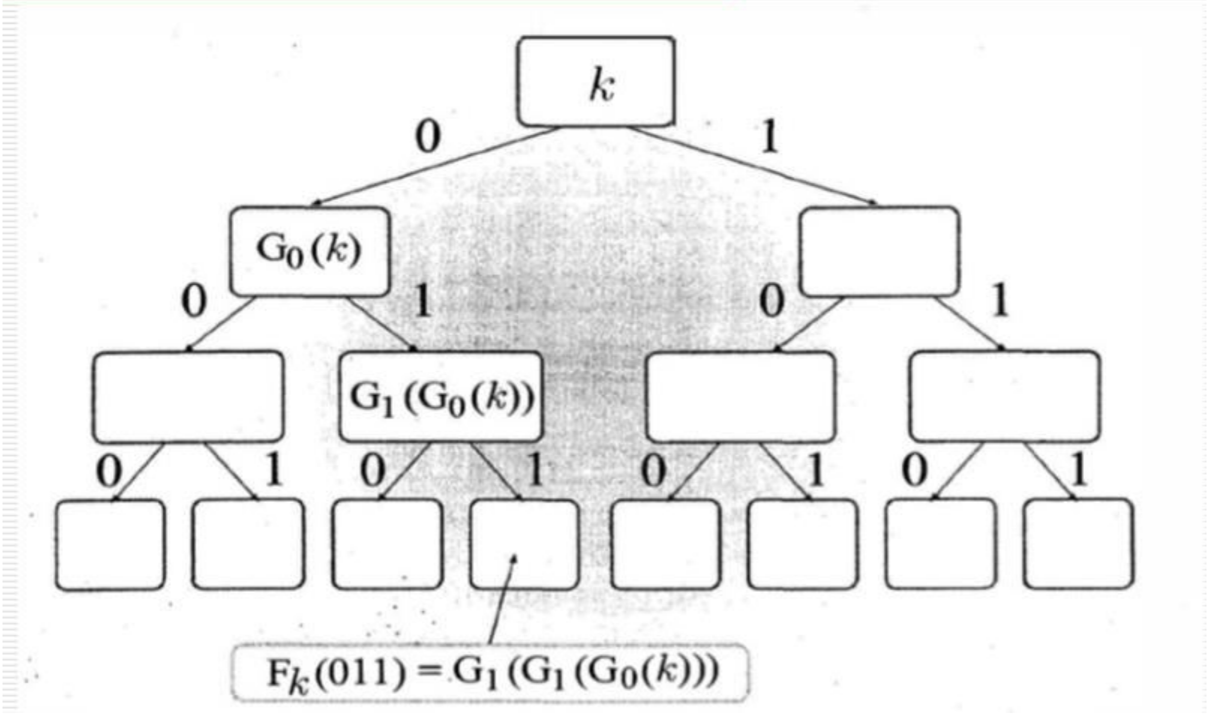


CONSTRUCTION 6.24

Let G be a pseudorandom generator with expansion factor $\ell(n) = 2n$. Denote by $G_0(k)$ the first half of G's output, and by $G_1(k)$ the second half of G's output. For every $k \in \{0,1\}^n$, define the function $F_k : \{0,1\}^n \to \{0,1\}^n$ as:

$$F_k(x_1x_2\cdots x_n) = G_{x_n}\left(\cdots(G_{x_2}(G_{x_1}(k)))\cdots\right).$$

A pseudorandom function from a pseudorandom generator.



1. So, we first took input of prime(p), generator(g), key(k), data. Now, we run a loop will run n times, where n = length of the input data in binary. For each iteration, the key will go inside the length doubling PRG, and a 2x length pseudorandom number comes out from PRG where x = length of the key. Now, it will check the bit of the data. If it's 0, it'll choose first x bits (left half), or else, it'll choose the next x bits (right half). And it will be sent as input for the next iteration in PRG.
   So, at the end the output length will be equal to length of key in binary.
   This is how we made PRF from length doubling PRG.

References

[1] J. K. a. Y. Lindell, Introduction to Modern Cryptography.