CS 830: Randomness and Complexity 1/25/05

Lecture 3: More on One-way Function Reductions

instructor: Jin-Yi Cai Scribe: Matthew Anderson

1 Strength of the Reduction

In the first two examples of reduction we randomly inserted the value we were trying to invert into a sequence of other values:

$$f(x_1), f(x_2), ..., f(x_{i-1}), f(x) = y_i, f(x_{i+1}), ...f(x_N). \tag{1}$$

This allowed us to show that if we could invert g then we could also invert f, because eventually we could use the adversary that could break the "hardcore" of g to break the "hardcore" of f. Since we assumed f was secure then no such adversary could exists, so g must also be secure.

There are some common properties to the reductions we have seen so far, which we have used for arguments of transfer of breakability across the reduction:

1. Given an $f(x)$ we can find a $g(z)$ where z is a uniform random neighbor of x.
2. Given the inverse of $g(z)$ we can find the inverse of $f(x)$ and hence find x.

We mapped the previous two reductions onto a bipartite graph where the left partition of the graph was the set of inputs to f (with cardinality 2^n) and the right partition was inputs to g (with cardinality $2^{n(N-1)}$). The edge relation was defined by the neighbor relation between x and z in the structure of our reduction.

The "generic" view that we would like to have is "if f is impossible to invert in polytime, then g is impossible to invert in polytime". Unfortunately, we have not, as of yet, been able to prove that something is impossible to invert. Our only crutch, however, is that we can show "if f is impossible, then g is impossible"; this is only an implication and not an absolute statement.

We can discuss how the difficulty of inversion transforms across the reduction. If g can be inverted in $\frac{T'(n)}{\delta'(n)} \leq R'(n)$, then f can be inverted in $\frac{T(n)}{\delta(n)} \leq R(n)$. Recall $T(n)$ is the worst case running time of the adversary and $\delta(n)$ is the success probability. Usually we have $R(n) \geq R'(n)$ because there is some overhead associated with doing the reduction. In the best case we would like $R(n) \sim R'(n)$, which gives the sense of efficiently injected the "hardness" of f into g. We can discuss several orders of strength when comparing how much hardness is injected into g:

1. Linear reduction - $R(n) \leq n^{c_1} R'(n)$
2. Polynomial reduction - $R(n) \leq n^{c_1} (R'(n))^{c_2}$
3. Weak reduction - $R(n) \leq n^{c_1} (R'(n^{c_3}))^{c_2}$

As the right side of the inequality increases we get asymptotically further from our hope of $R(n) \sim R'(n)$. For protocols that are polynomial time secure, these measures of reduction strength are largely irrelevant because $R(n)$ and $R'(n)$ are both polynomial time. However, in the exponential security domain the result is different, consider the following examples:

1. Linear reduction
   If f is $(n^c 2^{n^{1/2}})$-secure then g is $(2^{n^{1/2}})$-secure. This is not a strong difference in security.

2. Polynomial reduction
   If f is $(2^{2n^{1/2}})$-secure, then g is $(2^{n^{1/2}})$-secure.

3. Weak reduction
   If f is (2^{2n})-secure, then g is $(2^{n^{1/2}})$-secure. But f cannot be (2^{2n})-secure, because then you can try all the cases for inversion. If f is $(2^{n^{1/2}})$-secure, then g is $(2^{n^{1/4}})$-secure.

Note that n, in the above examples, was the security parameter, but more generally, n should be replace by some polynomial function $s(n)$ as the security parameter. In general, we prefer algorithms where $s(n)$ does not depend badly on n.

2 A Linear Reduction with Less Public Randomness

We are going to present another example that is slightly more complex then the previous examples using less public randomness. The process for going about proving the security is the same as with the previous reductions. [Note: that this material will be used in our discussion of Nissan's generator at a later date.]

Let $l = O\left(\log \frac{1}{w(n)}\right)$. Recall that $w(n)$ is the fraction of instances of f over 2^n that are "hard" to invert. If we have $w(n) = \frac{1}{n^c}$ then $l = O(\log n)$.

Define the set of public random values to be:

$$\Pi =< \Pi_1, \Pi_2, ..., \Pi_l >, \Pi_i \epsilon_U \{0,1\}^n. \tag{2}$$

Now will now construct our generator g using the public random input Π. We define g in two stages:

$$\begin{aligned} s :& 2^n \times 2^{l \cdot n} \to 2^n \\ & < x, \Pi > \mapsto s_l(x, \Pi) \\ \\ s_1 &= f(f(x) \oplus \Pi_1) \\ &\vdots \\ s_i &= s_i(x, \Pi_{\{1,...,i\}}) = s_{i-1}(s_{i-1}(x, \Pi\{1,...,i-1\}) \oplus \Pi_i, \Pi_{\{1,...,i-1\}}) \end{aligned} \tag{3}$$

$$g(x, \Pi) = (s(x, \Pi); \Pi) \tag{4}$$

Here are a few diagrams on computation tree to illustrate the computation of s as depth-first traversal of the tree. The computation of s_1, s_2, and s_i are shown in Figures 1,2, and 3 respectively.

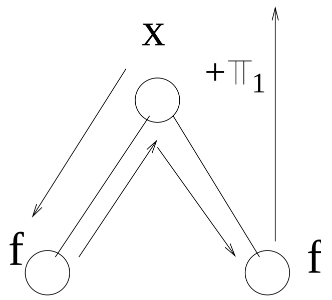

Figure 1: Evaluation of $s_1 = f(f(x) \oplus \Pi_1)$

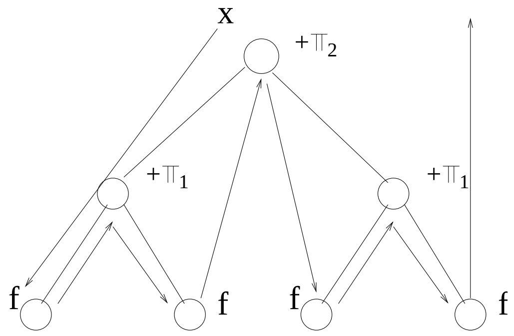

Figure 2: Evaluation of $s_2 = f(f(f(f(x) \oplus \Pi_1) \oplus \Pi_2) \oplus \Pi_1)$

We can also define a sequential algorithm that produces the same result as the above construction of g.

Let $N = 2^l$.
Set $y_1 = x$.
For i from 1 to $2^l - 1$:
    Let i be written as $i_l ... i_2 i_1$.
    $m_i = \min \{j \epsilon \{1, ..., l\} | i_j = 1\}$
    Set $y_{i+1} = f(y_i) \oplus \Pi_{m_i}$.
return $< f(y_N), \Pi_{1...l} >$.

As an exercise, try to prove that the two methods are equivalent. [Hint: For $N/2 < i < N$, $m_i = m_{i-N/2}$. Also $f(y_N) = s_l$ and $f(y_{N-1}) = s_{l-1}$.]

Now that we have an algorithm, we proceed as before and construct a mapping on a bipartite graph (L, R, E) (shown in Figure 4). The left side of the graph will be the inputs to f, $|L| = 2^n$ and the right side will be inputs to g, $|R| = 2^n \cdot 2^{l \cdot n}$. There will be an edge connection $x \epsilon L$ and $< z, \Pi > \epsilon R$ if and only if $x = y_i$ for some i in the sequence of y's $(y_1, y_2, ..., y_N)$ defined by $< z, \Pi >$. Notice, that this implies that the left degree of the bipartite graph is $N \cdot 2^{l \cdot n}$ and the right degree is N. We can use this to compute g of a uniform random neighbor of x.

Lemma 2.1 *Our bipartite graph (L, R, E) is $(\frac{w(n)}{2}, \frac{3}{4})$ - forward expanding. (i.e. $\forall S \subseteq L$, if $\mu(S) \geq \frac{w(n)}{2}$ then $\mu(\Gamma(S)) \geq \frac{1}{4}$.)*

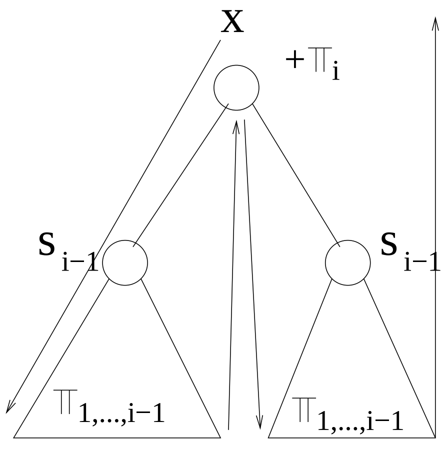

Figure 3: Evaluation of s_i

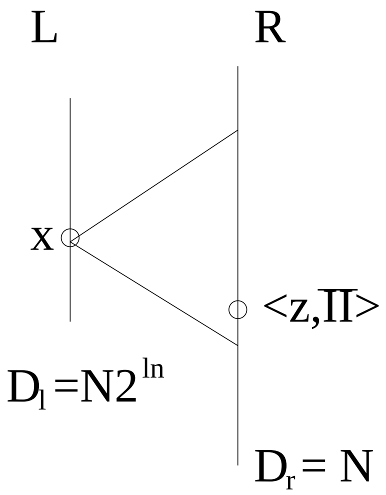

Figure 4: Mapping onto bipartite graph

We will prove this lemma in the next class.

The question that we would like to answer is "What is the set of $x\epsilon\{0,1\}^n$ such that if we have $g(x,\Pi)$ has at least one i such that $y_i\epsilon S$?" Define $F_{\Pi_{\{1,\ldots,i\}}}$ as $\{x|s_i(x,\Pi_{1,\ldots,i})\}$. This is the set of x when we compute $g(x,\Pi_{\{1,\ldots,i})$ such that at least one $y_i\epsilon S$. $F_{\Pi_\phi} = S$. F_i is the set at the i^{th} level of the tree, we have the recurrence relation $F_i = F_{i-1} \cup (\Pi_i \oplus F_{i-1})$. While the size of $|F_i|$ is relatively small, we expect that the intersection between F_{i-1} and F_{i-1} randomly shifted by Π_i to be of smaller order than the set themselves.

We will later show that this quantity is $\lceil 6\log_{5/4}\frac{1}{w(n)}\rceil$