A Generic Hardcore Predicate for any One-Way Function
Extracts from Foundations of Cryptography[1]

Oded Goldreich
Department of Computer Science
Weizmann Institute of Science
Rehovot, ISRAEL.

February 8, 2001

[1] This is Section 2.5 of the book *Foundations of Cryptography*. For further details on the book, see webpage `http://www.wisdom.weizmann.ac.il/~oded/foc.html`.

2.5 Hard-Core Predicates

Loosely speaking, saying that a function f is one-way implies that given y it is infeasible to find a preimage of y under f. This does not mean that it is infeasible to find out partial information about the preimage of y under f. Specifically it may be easy to retrieve half of the bits of the preimage (e.g., given a one-way function f consider the function g defined by $g(x,r) \stackrel{\text{def}}{=} (f(x),r)$, for every $|x|=|r|$). The fact that one-way functions do not necessarily hide partial information about their preimage limits their "direct applicability" to tasks as secure encryption. Fortunately, assuming the existence of one-way functions, it is possible to construct one-way functions that hide specific partial information about their preimage (which is easy to compute from the preimage itself). This partial information can be considered as a "hard core" of the difficulty of inverting f.

2.5.1 Definition

Loosely speaking, a *polynomial-time* predicate b, is called a hard-core of a function f if every efficient algorithm, given $f(x)$, can guess $b(x)$ only with success probability that is negligibly better than one half.

Definition 2.5.1 (hard-core predicate): *A polynomial-time computable predicate $b : \{0,1\}^* \to \{0,1\}$ is called a* hard-core *of a function f if for every probabilistic polynomial-time algorithm A', every positive polynomial $p(\cdot)$, and all sufficiently large n's*

$$\mathsf{Pr}\left[A'(f(U_n))=b(U_n)\right] < \frac{1}{2} + \frac{1}{p(n)}$$

Note that for every $b : \{0,1\}^* \to \{0,1\}$ and $f : \{0,1\}^* \to \{0,1\}^*$, there exist obvious algorithms that guess $b(U_n)$ from $f(U_n)$ with success probability at least one half (e.g., the algorithm that, obliviously of its input, outputs a uniformly chosen bit). Also, if b is a hard-core predicate (for any function) then $b(U_n)$ must be almost unbiased (i.e., $|\mathsf{Pr}[b(U_n)=0] - \mathsf{Pr}[b(U_n)=1]|$ must be a negligible function in n).

Since b itself is polynomial-time computable, the failure of efficient algorithms to approximate $b(x)$ from $f(x)$ (with success probability non-negligibly higher than one half) must be due to either an information loss of f (i.e., f not being one-to-one) or to the difficulty of inverting f. For example, the predicate $b(\sigma\alpha) = \sigma$ is a hard-core of the function $f(\sigma\alpha) \stackrel{\text{def}}{=} 0\alpha$, where $\sigma \in \{0,1\}$ and $\alpha \in \{0,1\}^*$. Hence, in this case the fact that b is a hard-core of the function f is due to the fact that f loses information (specifically, the first bit σ). On the other hand, in case f loses no information (i.e., f is one-to-one) hard-cores for f exist only if f is one-way (see Exercise **??**). We will be interested in the case where the hardness of approximating $b(x)$ from $f(x)$ is due to computational reasons and not to information theoretic ones (i.e., information loss).

Hard-core predicates for collections of one-way functions are defined in an analogous way. Typically, the predicate may depend on the index of the function, and both algorithms (i.e., the one for evaluating it as well as the one for predicting it based on the function value) are also given this index. That is, a polynomial-time algorithm $B : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ is called a hard-core of the one-way collection (I, D, F) if for every probabilistic polynomial-time algorithm, A', every positive polynomial $p(\cdot)$, and all sufficiently large n's

$$\mathsf{Pr}\left[A'(I_n, f_{I_n}(X_n))=B(I_n, X_n)\right] < \frac{1}{2} + \frac{1}{p(n)}$$

where $I_n \stackrel{\text{def}}{=} I(1^n)$ and $X_n \stackrel{\text{def}}{=} D(I_n)$.

Some natural candidates: Simple hard-core predicates are known for the RSA, Rabin, and DLP collections (presented in Subsection **??**), provided that the corresponding collections are one-way. Specifically, the least significant bit is a hard-core for the RSA collection, provided that the RSA collection is one-way. Namely, assuming that the RSA collection is one-way, it is infeasible to guess (with success probability significantly greater than half) the least significant bit of x from $RSA_{N,e}(x) = x^e \bmod N$. Similarly, assuming the intractability of integer factorization, it is infeasible to guess the least significant bit of $x \in Q_N$ from $Rabin_N(x) = x^2 \bmod N$, where N is a Blum integer (and Q_N denotes the set of quadratic residues modulo N). Finally, assuming that the DLP collection is one-way, it is infeasible to guess whether $x < \frac{P}{2}$ when given $DLP_{P,G}(x) = G^x \bmod P$. In the next subsection we present a general result of this type.

2.5.2 Hard-Core Predicates for any One-Way Function

Actually, the title is inaccurate: we are going to present hard-core predicates only for (strong) one-way functions of a special form. However, every (strong) one-way function can be easily transformed into a function of the required form, with no substantial loss in either "security" or "efficiency".

Theorem 2.5.2 *Let f be an arbitrary strong one-way function, and let g be defined by $g(x,r) \stackrel{\text{def}}{=} (f(x),r)$, where $|x|=|r|$. Let $b(x,r)$ denote the inner-product mod 2 of the binary vectors x and r. Then the predicate b is a hard-core of the function g.*

In other words, the theorem states that if f is strongly one-way then it is infeasible to guess the exclusive-or of a random subset of the bits of x when given $f(x)$ and the subset itself. We stress that the theorem requires that f is strongly one-way and that the conclusion is false if f is only weakly one-way (see Exercise **??**). Clearly, g is also strongly one-way. We point out that g maintains other properties of f such as being length-preserving and being one-to-one. Furthermore, an analogous statement holds for collections of one-way functions with/without trapdoor etc.

The rest of this section is devoted to proving Theorem 2.5.2. Again we use a reducibility argument: here inverting the function f is reduced to guessing $b(x,r)$ from $(f(x),r)$. Hence, we assume (for contradiction) the existence of an efficient algorithm guessing the inner-product with advantage that is non-negligible, and derive an algorithm that inverts f with related (i.e., non-negligible) success probability. This contradicts the hypothesis that f is a one-way function.

We start with some preliminary observations and a motivating discussion, and then turn to the main part of the actual proof. We conclude with more efficient implementations of the reducibility argument, which assert "higher levels of security".

2.5.2.1 Preliminaries

Let G be a (probabilistic polynomial-time) algorithm that on input $f(x)$ and r tries to guess the inner-product (mod 2) of x and r. Denote by $\varepsilon_G(n)$ the (overall) advantage of algorithm G in guessing $b(x,r)$ from $f(x)$ and r, where x and r are uniformly chosen in $\{0,1\}^n$. Namely,

$$\varepsilon_G(n) \stackrel{\text{def}}{=} \mathsf{Pr}\left[G(f(X_n),R_n) = b(X_n,R_n)\right] - \frac{1}{2} \tag{2.1}$$

where here and in the sequel X_n and R_n denote two independent random variables, each uniformly distributed over $\{0,1\}^n$. Assuming, to the contradiction, that b is not a hard-core of g means that there exists an efficient algorithm G, a polynomial $p(\cdot)$ and an infinite set N so that for every $n \in N$ it holds that $\varepsilon_G(n) > \frac{1}{p(n)}$. We restrict our attention to this algorithm G and to n's in this set N. In the sequel we shorthand ε_G by ε.

Our first observation is that, on at least an $\frac{\varepsilon(n)}{2}$ fraction of the x's of length n, algorithm G has at least an $\frac{\varepsilon(n)}{2}$ advantage in guessing $b(x, R_n)$ from $f(x)$ and R_n. Namely,

Claim 2.5.2.1: there exists a set $S_n \subseteq \{0,1\}^n$ of cardinality at least $\frac{\varepsilon(n)}{2} \cdot 2^n$ such that for every $x \in S_n$, it holds that

$$s(x) \stackrel{\text{def}}{=} \mathsf{Pr}[G(f(x), R_n) = b(x, R_n)] \geq \frac{1}{2} + \frac{\varepsilon(n)}{2}$$

Here the probability is taken over all possible values of R_n and all internal coin tosses of algorithm G, whereas x is fixed.

Proof: The claim follows by an averaging argument. Namely, write $\mathsf{E}(s(X_n)) = \frac{1}{2} + \varepsilon(n)$, and apply Markov's Inequality. $\square$

In the sequel we restrict our attention to x's in S_n. We will show an efficient algorithm that on every input y, with $y = f(x)$ and $x \in S_n$, finds x with very high probability. Contradiction to the (strong) one-wayness of f will follow by recalling that $\mathsf{Pr}[U_n \in S_n] \geq \frac{\varepsilon(n)}{2}$.

We start with a motivating discussion. The inverting algorithm, that uses algorithm G as subroutine, will be formally described and analyzed later.

2.5.2.2 A motivating discussion

Consider a fixed $x \in S_n$. By definition $s(x) \geq \frac{1}{2} + \frac{\varepsilon(n)}{2} > \frac{1}{2} + \frac{1}{2p(n)}$. Suppose, for a moment, that $s(x) > \frac{3}{4} + \frac{1}{2p(n)}$. Of course there is no reason to believe that this is the case, we are just doing a mental experiment. Still, in this case (i.e., of $s(x) > \frac{3}{4} + \frac{1}{\text{poly}(|x|)}$) retrieving x from $f(x)$ is quite easy. To retrieve the i^{th} bit of x, denoted x_i, we randomly select $r \in \{0,1\}^n$, and compute $G(f(x), r)$ and $G(f(x), r \oplus e^i)$, where e^i is an n-dimensional binary vector with 1 in the i^{th} component and 0 in all the others, and $v \oplus u$ denotes the addition mod 2 of the binary vectors v and u. (The process is actually repeated polynomially-many times, using independent random choices of such r's, and x_i is determined by a majority vote.)
If both $G(f(x), r) = b(x, r)$ and $G(f(x), r \oplus e^i) = b(x, r \oplus e^i)$, then

$$\begin{aligned} G(f(x), r) \oplus G(f(x), r \oplus e^i) &= b(x, r) \oplus b(x, r \oplus e^i) \\ &= b(x, e^i) \\ &= x_i \end{aligned}$$

where the second equality uses

$$b(x, r) \oplus b(x, s) \equiv \sum_{i=1}^{n} x_i r_i + \sum_{i=1}^{n} x_i s_i \equiv \sum_{i=1}^{n} x_i (r_i + s_i) \equiv b(x, r \oplus s) \pmod 2 .$$

The probability that both $G(f(x), r) = b(x, r)$ and $G(f(x), r \oplus e^i) = b(x, r \oplus e^i)$ hold, for a random r, is at least $1 - 2 \cdot (\frac{1}{4} - \frac{1}{\text{poly}(|x|)}) > \frac{1}{2} + \frac{1}{\text{poly}(|x|)}$. Hence, repeating the above procedure sufficiently many times and ruling by majority, we retrieve x_i with very high probability. Similarly, we can retrieve all the bits of x, and hence invert f on $f(x)$. However, the entire analysis was conducted under (the unjustifiable) assumption that $s(x) > \frac{3}{4} + \frac{1}{2p(|x|)}$, whereas we only know that $s(x) > \frac{1}{2} + \frac{1}{2p(|x|)}$.

The problem with the above procedure is that it doubles the original error probability of algorithm G on inputs of the form $(f(x), \cdot)$. Under the unrealistic assumption, that G's average error on such inputs is non-negligibly smaller than $\frac{1}{4}$, the "error-doubling" phenomenon raises no problems.

However, in general (and even in the special case where G's error is exactly $\frac{1}{4}$) the above procedure is unlikely to invert f. Note that the *average* error probability of G (which is averaged over all possible inputs of the form $(f(x),\cdot)$) can not be decreased by repeating G several times (e.g., G may always answer correctly on three quarters of the inputs, and always err on the remaining quarter). What is required is an *alternative way of using* the algorithm G, a way that does not double the original error probability of G. The key idea is to generate the r's in a way that requires applying algorithm G only once per each r (and i), instead of twice. Specifically, we will use algorithm G to obtain a "guess" for $b(x, r \oplus e^i)$, and obtain $b(x,r)$ in a different way. The good news is that the error probability is no longer doubled, since we only use G to get a "guess" of $b(x, r \oplus e^i)$. The bad news is that we still need to know $b(x,r)$, and it is not clear how we can know $b(x,r)$ without applying G. The answer is that we can guess $b(x,r)$ by ourselves. This is fine if we only need to guess $b(x,r)$ for one r (or logarithmically in $|x|$ many r's), but the problem is that we need to know (and hence guess) the value of $b(x,r)$ for polynomially many r's. An obvious way of guessing these $b(x,r)$'s yields an exponentially vanishing success probability. Instead, we generate these polynomially many r's so that, on one hand they are "sufficiently random" whereas, on the other hand, we can guess all the $b(x,r)$'s with noticeable success probability. Specifically, generating the r's in a particular *pairwise independent* manner will satisfy both (seemingly contradictory) requirements. We stress that in case we are successful (in our guesses for all the $b(x,r)$'s), we can retrieve x with high probability. Hence, we retrieve x with noticeable probability.

A word about the way in which the pairwise independent r's are generated (and the corresponding $b(x,r)$'s are guessed) is indeed in place. To generate $m = \mathrm{poly}(n)$ many r's, we uniformly (and independently) select $l \stackrel{\text{def}}{=} \log_2(m+1)$ strings in $\{0,1\}^n$. Let us denote these strings by $s^1, ..., s^l$. We then guess $b(x, s^1)$ through $b(x, s^l)$. Let us denote these guesses, which are uniformly (and independently) chosen in $\{0,1\}$, by σ^1 through σ^l. Hence, the probability that all our guesses for the $b(x, s^i)$'s are correct is $2^{-l} = \frac{1}{\mathrm{poly}(n)}$. The different r's correspond to the different *non-empty* subsets of $\{1, 2, ..., l\}$. Specifically, we let $r^J \stackrel{\text{def}}{=} \oplus_{j \in J} s^j$. The reader can easily verify that the r^J's are pairwise independent and each is uniformly distributed in $\{0,1\}^n$. The key observation is that

$$b(x, r^J) = b(x, \oplus_{j \in J} s^j) = \oplus_{j \in J} b(x, s^j)$$

Hence, our guess for the $b(x, r^J)$'s is $\oplus_{j \in J} \sigma^j$, and with noticeable probability all our guesses are correct.

2.5.2.3 Back to the actual proof

Following is a formal description of the inverting algorithm, denoted A. We assume, for simplicity that f is length preserving (yet this assumption is not essential). On input y (supposedly in the range of f), algorithm A sets $n \stackrel{\text{def}}{=} |y|$, and $l \stackrel{\text{def}}{=} \lceil \log_2(2n \cdot p(n)^2 + 1) \rceil$, where $p(\cdot)$ is the polynomial guaranteed above (i.e., $\epsilon(n) > \frac{1}{p(n)}$ for the infinitely many n's in N). Algorithm A proceeds as follows:

(1) Uniformly and independently selects $s^1, ..., s^l \in \{0,1\}^n$, and $\sigma^1, ..., \sigma^l \in \{0,1\}$.
(2) For every non-empty set $J \subseteq \{1, 2, ..., l\}$,
    computes a string $r^J \leftarrow \oplus_{j \in J} s^j$ and a bit $\rho^J \leftarrow \oplus_{j \in J} \sigma^j$.
(3) For every $i \in \{1, ..., n\}$ and every non-empty $J \subseteq \{1, .., l\}$,
    computes $z_i^J \leftarrow \rho^J \oplus G(y, r^J \oplus e^i)$.
(4) For every $i \in \{1, ..., n\}$, sets z_i to be the majority of the z_i^J values.
(5) Outputs $z = z_1 \cdots z_n$.

Remark: an alternative implementation. In an alternative implementation of the above ideas, the inverting algorithm tries all possible values for $\sigma^1, ..., \sigma^l$, computes a string z for each of these 2^l possibilities, and outputs only one of the resulting z's, with an obvious preference to a string z satisfying $f(z) = y$. For later reference, this alternative algorithm is denoted A'. (See further discussion in the next subsection.)

Following is a detailed analysis of the success probability of algorithm A on inputs of the form $f(x)$, for $x \in S_n$, where $n \in N$. One key observation, which is extensively used, is that for $x, \alpha, \beta \in \{0,1\}^n$, it holds that

$$b(x, \alpha \oplus \beta) = b(x, \alpha) \oplus b(x, \beta)$$

It follows that $b(x, r^J) = b(x, \oplus_{j\in J} s^j) = \oplus_{j\in J} b(x, s^j)$. The main part of the analysis is showing that, in case the σ^j's are correct (i.e., $\sigma^j = b(x, s^j)$ for all $j \in \{1, ..., l\}$), with constant probability, $z_i = x_i$ for all $i \in \{1, ..., n\}$. This is proven by bounding from below the probability that the majority of the z_i^J's equals x_i, where $z_i^J = b(x, r^J) \oplus G(f(x), r^J \oplus e^i)$ (due to the hypothesis that $\sigma^j = b(x, s^j)$ for all $j \in \{1, ..., l\}$).

Claim 2.5.2.2: For every $x \in S_n$ and every $1 \le i \le n$,

$$\mathsf{Pr}\left[\left|\left\{J : b(x, r^J) \oplus G(f(x), r^J \oplus e^i) = x_i\right\}\right| > \frac{1}{2} \cdot (2^l - 1)\right] \; > \; 1 - \frac{1}{2n}$$

where $r^J \stackrel{\text{def}}{=} \oplus_{j\in J} s^j$ and the s^j's are independently and uniformly chosen in $\{0,1\}^n$.

Proof: For every J, define a 0-1 random variable ζ^J, so that ζ^J equals 1 if and only if $b(x, r^J) \oplus G(f(x), r^J \oplus e^i) = x_i$. Since $b(x, r^J) \oplus b(x, r^J \oplus e^i) = x_i$, it follows that $\zeta^J = 1$ if and only if $G(f(x), r^J \oplus e^i) = b(x, r^J \oplus e^i)$.

The reader can easily verify that each r^J is uniformly distributed in $\{0,1\}^n$, and the same holds for each $r^J \oplus e^i$. It follows that each ζ^J equals 1 with probability $s(x)$, which by $x \in S_n$, is at least $\frac{1}{2} + \frac{1}{2p(n)}$. We show that the ζ^J's are pairwise independent by showing that the r^J's are pairwise independent. For every $J \ne K$, without loss of generality, there exist $j \in J$ and $k \in K - J$. Hence, for every $\alpha, \beta \in \{0,1\}^n$, we have

$$\begin{aligned}
\mathsf{Pr}\left[r^K = \beta \,\middle|\, r^J = \alpha\right] &= \mathsf{Pr}\left[s^k = \beta \,\middle|\, s^j = \alpha\right] \\
&= \mathsf{Pr}\left[s^k = \beta\right] \\
&= \mathsf{Pr}\left[r^K = \beta\right]
\end{aligned}$$

and pairwise independence of the r^J's follows. Let $m \stackrel{\text{def}}{=} 2^l - 1$ and ζ represent a generic ζ^J (which are all identically distributed). Using Chebyshev's Inequality (and $m \ge 2n \cdot p(n)^2$), we get

$$\begin{aligned}
\mathsf{Pr}\left[\sum_J \zeta^J \le \frac{1}{2} \cdot m\right] &\le \mathsf{Pr}\left[\left|\sum_J \zeta^J - \left(\frac{1}{2} + \frac{1}{2p(n)}\right) \cdot m\right| \ge \frac{1}{2p(n)} \cdot m\right] \\
&\le \frac{m \cdot \mathrm{Var}[\zeta]}{\left(\frac{1}{2p(n)} \cdot m\right)^2} \\
&= \frac{\mathrm{Var}[\zeta]}{\left(\frac{1}{2p(n)}\right)^2 \cdot (2n \cdot p(n)^2)}
\end{aligned}$$

$$
\begin{aligned}
&< \frac{\frac{1}{4}}{\left(\frac{1}{2p(n)}\right)^2 \cdot (2n \cdot p(n)^2)} \\
&= \frac{1}{2n}
\end{aligned}
$$

The claim follows. □

Recall that if $\sigma^j = b(x, s^j)$, for all j's, then $\rho^J = \oplus_{j\in J}\sigma^j = \oplus_{j\in J} b(x, s^j) = b(x, r^J)$, for all non-empty J's. In this case, with probability at least one half, the string z output by algorithm A equals x. However, the first event (i.e., $\sigma^j = b(x, s^j)$ for all j's) happens with probability $2^{-l} = \frac{1}{2n\cdot p(n)^2+1}$ independently of the events analyzed in Claim 2.5.2.2. Hence, in case $x \in S_n$, algorithm A inverts f on $f(x)$ with probability at least $\frac{1}{2} \cdot 2^{-l} = \frac{1}{4n\cdot p(|x|)^2+2}$ (whereas, the alternative algorithm, A', succeeds with probability at least $\frac{1}{2}$). Recalling that (by Claim 2.5.2.1) $|S_n| > \frac{1}{2p(n)} \cdot 2^n$, we conclude that, for every $n \in N$, algorithm A inverts f on $f(U_n)$ with probability at least $\frac{1}{8n\cdot p(n)^3+4p(n)}$. Noting that A is polynomial-time (i.e., it merely invokes G for $2n \cdot p(n)^2 = \mathrm{poly}(n)$ times in addition to making a polynomial amount of other computations), a contradiction to our hypothesis that f is strongly one-way follows. ■

2.5.2.4 * More efficient reductions

The above proof actually establishes the following

Proposition 2.5.3 *Let G be a probabilistic algorithm with running-time $t_G : \mathbb{N} \to \mathbb{N}$ and advantage $\varepsilon_G : \mathbb{N} \to [0,1]$ in guessing b* (cf. Eq. (2.1)). *Then there exists an algorithm A that runs in time $O(n^2/\varepsilon_G(n)^2) \cdot t_G(n)$ so that*

$$
\Pr[A(f(U_n)) = U_n] \;\geq\; \frac{\varepsilon_G(n)}{2} \cdot \frac{\varepsilon_G(n)^2}{4n}
$$

The alternative implementation, A', mentioned above (i.e., trying all possible values of the σ^j's rather than guessing one of them), runs in time $O(n^3/\varepsilon_G(n)^4) \cdot t_G(n)$ and satisfies

$$
\Pr[A'(f(U_n)) = U_n] \;\geq\; \frac{\varepsilon_G(n)}{2} \cdot \frac{1}{2}
$$

Below, we provide a more efficient implementation of A'. Combining it with a more refined averaging argument than the one used in Claim 2.5.2.1, we obtain:

Proposition 2.5.4 *Let G, $t_G : \mathbb{N} \to \mathbb{N}$ and $\varepsilon_G : \mathbb{N} \to [0,1]$ be as above, and define $\ell(n) \stackrel{\text{def}}{=} \log_2(1/\varepsilon_G(n))$. Then there exists an algorithm A'' that runs in* expected *time $O(n^2 \cdot \ell(n)^3) \cdot t_G(n)$ and satisfies*

$$
\Pr[A''(f(U_n)) = U_n] \;=\; \Omega(\varepsilon_G(n)^2)
$$

Thus, the *time over success ratio* of A'' is $\mathrm{poly}(n)/\varepsilon_G(n)^2$, which (in some sense) is optimal up to a $\mathrm{poly}(n)$ factor; see Exercise **??**.

Proof Sketch: Let $\varepsilon(n) \stackrel{\text{def}}{=} \varepsilon_G(n)$, and $\ell \stackrel{\text{def}}{=} \log_2(1/\varepsilon(n))$. Recall that $\mathsf{E}[s(X_n)] = 0.5 + \varepsilon(n)$, where $s(x) \stackrel{\text{def}}{=} \Pr[G(f(x), R_n) = b(x, R_n)]$ (as in Claim 2.5.2.1). We first replace Claim 2.5.2.1 by a more refined analysis.

Claim 2.5.4.1: There exists an $i \in \{1, ..., \ell\}$ and a set $S_n \subseteq \{0,1\}^n$ of cardinality at least $(2^{i-1} \cdot \varepsilon(n)) \cdot 2^n$ such that for every $x \in S_n$, it holds that

$$s(x) = \Pr[G(f(x), R_n) = b(x, R_n)] \ \geq \ \frac{1}{2} + \frac{1}{2^{i+1} \cdot \ell}$$

Proof: Let $A_i \stackrel{\text{def}}{=} \{x : s(x) \geq \frac{1}{2} + \frac{1}{2^{i+1}\ell}\}$. For any non-empty set $S \subseteq \{0,1\}^n$, we let $a(S) \stackrel{\text{def}}{=} \max_{x \in S}\{s(x) - 0.5\}$, and $a(\emptyset) \stackrel{\text{def}}{=} 0$. Assuming to the contradiction that the claim does not hold (i.e., $|A_i| < (2^{i-1} \cdot \varepsilon(n)) \cdot 2^n$ for $i = 1, ..., \ell$), we get

$$\begin{aligned}
\mathsf{E}[s(X_n) - 0.5] \ &\leq \ \Pr[X_n \in A_1] \cdot a(A_1) + \sum_{i=2}^{\ell} \Pr[X_n \in (A_i \setminus A_{i-1})] \cdot a(A_i \setminus A_{i-1}) \\
&\quad + \Pr[X_n \in (\{0,1\}^n \setminus A_\ell)] \cdot a(\{0,1\}^n \setminus A_\ell) \\
&< \ \varepsilon(n) \cdot \frac{1}{2} + \sum_{i=2}^{\ell} (2^{i-1} \cdot \varepsilon(n)) \cdot \frac{1}{2^i \ell} + 1 \cdot \frac{1}{2^{\ell+1}\ell} \\
&= \ \frac{\varepsilon(n)}{2} + (\ell - 1) \cdot \frac{\varepsilon(n)}{2\ell} + \frac{2^{-\ell}}{2\ell} \ = \ \varepsilon(n)
\end{aligned}$$

which contradicts $\mathsf{E}[s(X_n) - 0.5] = \varepsilon(n)$. □

Fixing any i that satisfies Claim 2.5.4.1, we let $\epsilon \stackrel{\text{def}}{=} 2^{-i-1}/\ell$, and consider the corresponding set $S_n \stackrel{\text{def}}{=} \{x : s(x) \geq 0.5 + \epsilon\}$. By suitable setting of parameters, we obtain that for every $x \in S_n$, algorithm A' runs in time $O(n^3/\epsilon^4) \cdot t_G(n)$ and retrieves x from $f(x)$ with probability at least 1/2. Our next goal is to provide a more efficient implementation of A'; specifically, one running in time $O(n^2/\epsilon^2) \cdot (t_G(n) + \log(n/\epsilon))$.

The modified algorithm A' is given input $y = f(x)$ and a parameter ϵ, and sets $l = \log((n/\epsilon^2) + 1)$. In the actual description (presented below), it will be more convenient to use arithmetic of reals instead of that of Boolean. Hence, we denote $b'(x, r) = (-1)^{b(x,r)}$ and $G'(y, r) = (-1)^{G(y,r)}$. The verification of the following facts is left as an exercise:

Fact 1: For every x it holds that $\mathsf{E}[b'(x, U_n) \cdot G'(f(x), U_n + e^i)] \ = \ s'(x) \cdot (-1)^{x_i}$, where $s'(x) \stackrel{\text{def}}{=} 2 \cdot (s(x) - \frac{1}{2})$. (Note that, for $x \in S_n$, we have $s'(x) \geq 2\epsilon$.)

Fact 2: Let R be a uniformly chosen l-by-n Boolean matrix. Then, for every $v \neq u \in \{0,1\}^l \setminus \{0\}^l$, it holds that vR and uR are pairwise independent and uniformly distributed in $\{0,1\}^n$.

Fact 3: For every $x \in \{0,1\}^n$ and $v \in \{0,1\}^l$, it holds that $b'(x, vR) = b'(xR^T, v)$.

Using these facts, we obtain

Claim 2.5.4.2: For any $x \in S_n$ and a uniformly chosen l-by-n Boolean matrix R, there exists $\sigma \in \{0,1\}^l$ so that, with probability at least $\frac{1}{2}$, for every $1 \leq i \leq n$, the sign of $\sum_{v \in \{0,1\}^l} b'(\sigma, v) \cdot G'(f(x), vR + e^i))$ equals the sign of $(-1)^{x_i}$.

Proof: Let $\sigma = xR^T$. Combining the above facts, for every $v \in \{0,1\}^l \setminus \{0\}^l$, we have $\mathsf{E}[b'(xR^T, v) \cdot G'(f(x), vR + e^i)] = s'(x) \cdot (-1)^{x_i}$. Thus, for every such v, it holds that $\Pr[b'(xR^T, v) \cdot G'(f(x), vR + e^i) = (-1)^{x_i}] = \frac{1+s'(x)}{2} = s(x)$. Using Fact 2, $l = \log((2n/\epsilon^2) + 1)$ and Chebyshev's Inequality, the claim follows. □

A last piece of notation: Let B be an 2^l-by-2^l matrix with the (σ, v)-entry being $b'(\sigma, v)$, and let $\overline{g}^i$ be an 2^l-dimensional vector with the v^{th} entry equal $G'(f(x), vR+e^i)$. Thus, the σ^{th} entry in the vector $B\overline{g}^i$ equals $\sum_{v\in\{0,1\}^l} b'(\sigma,v)\cdot G'(f(x), vR+e^i))$.

Efficient implementation of algorithm A'. On input $y = f(x)$ and a parameter ϵ, the inverting algorithm A' sets $l = \log((n/\epsilon^2)+1)$, and proceeds as follows:

(1) For $i = 1, ..., n$, computes the 2^l-dimensional vector $\overline{g}^i$ (as defined above).

(2) For $i = 1, ..., n$, computes $\overline{z}_i \leftarrow B\overline{g}^i$.
Let Z be an 2^l-by-n real matrix in which the i^{th} column equals $\overline{z}_i$.
Let Z' be an 2^l-by-n Boolean matrix representing the signs of the elements in Z:
specifically, the $(i,j)^{\text{th}}$ entry of Z' equals 1
if and only if the $(i,j)^{\text{th}}$ entry of Z is negative.

(3) Scanning all rows of Z', output the first row z so that $f(z) = y$.

By Claim 2.5.4.2, for $x \in S_n$, with probability at least 1/2, the above algorithm retrieves x from $y = f(x)$. The running time of the algorithm is dominated by Steps (1) and (2), which can be implemented in time $n\cdot 2^l \cdot O(t_G(n)) = O((n/\epsilon)^2\cdot t_G(n))$ and $n\cdot O(l\cdot 2^l) = O((n/\epsilon)^2\cdot\log(n/\epsilon))$, respectively.[1]

Finally, we define algorithm A''. On input $y = f(x)$, the algorithm selects $j \in \{1, ..., \ell\}$ with probability 2^{-2j+1} (and halts with no output otherwise). It invokes the above implementation of algorithm A' on input y with parameter $\epsilon \stackrel{\text{def}}{=} 2^{-j-1}/\ell$, and returns whatever A' does. The *expected* running time of A'' is

$$\sum_{j=1}^{\ell} 2^{-2j+1}\cdot O\left(\frac{n^2}{(2^{-j-1}/\ell)^2}\right)\cdot(t_G(n)+\log(n\cdot 2^{j+1}\ell)) \;=\; O(n^2\cdot\ell^3)\cdot t_G(n)$$

(Assuming $t_G(n) = \Omega(\ell\log n)$.) Letting $i \le \ell$ be an index satisfying Claim 2.5.4.1 (and S_n be the corresponding set), we consider the case in which j (selected by A'') is greater or equal to i. By Claim 2.5.4.2, in such a case and for $x \in S_n$, algorithm A' inverts f on $f(x)$ with probability at least one half. Using $i \le \ell$ $(= \log_2(1/\varepsilon(n)))$, we get

$$\begin{aligned}\mathsf{Pr}[A''(f(U_n)) = U_n] &\ge \mathsf{Pr}[U_n \in S_n]\cdot\mathsf{Pr}[j \ge i]\cdot\frac{1}{2}\\ &\ge 2^{i-1}\varepsilon(n)\cdot 2^{-2i+1}\cdot\frac{1}{2}\\ &\ge \varepsilon(n)\cdot 2^{-\ell}\cdot\frac{1}{2} \;=\; \frac{\varepsilon(n)^2}{2}\end{aligned}$$

The proposition follows. ■

[1] Using the special structure of matrix B, one may show that given a vector $\overline{w}$, the product $B\overline{w}$ can be computed in time $O(l\cdot 2^l)$. Hint: B (known as the Sylvester matrix) can be written recursively as

$$S_k = \begin{pmatrix} S_{k-1} & S_{k-1} \\ S_{k-1} & \overline{S_{k-1}} \end{pmatrix}$$

where $S_0 = +1$ and $\overline{M}$ means flipping the $+1$ entries of M to -1 and vice versa. So

$$\begin{pmatrix} S_{k-1} & S_{k-1} \\ S_{k-1} & \overline{S_{k-1}} \end{pmatrix}\begin{bmatrix} w' \\ w'' \end{bmatrix} = \begin{bmatrix} S_{k-1}w' + S_{k-1}w'' \\ S_{k-1}w' - S_{k-1}w'' \end{bmatrix}$$

Thus, letting $T(k)$ denote the time using in multiplying S_k by an 2^k-dimensional vector, we have $T(k) = 2\cdot T(k-1)+O(2^k)$, which solves to $T(k) = O(k2^k)$.

Comment: Using an additional trick,[2] one may save a factor of $\Theta(n)$ in the running time, resulting in *expected* running-time of $O(n \cdot \log^3(1/\varepsilon_G(n))) \cdot t_G(n)$.

2.5.3 * Hard-Core Functions

We have just seen that every one-way function can be easily modified to have a hard-core predicate. In other words, the result establishes one bit of information about the preimage that is hard to approximate from the value of the function. A stronger result may say that several bits of information about the preimage are hard to approximate. For example, we may want to say that a specific pair of bits is hard to approximate, in the sense that it is infeasible to guess this pair with probability non-negligibly larger than $\frac{1}{4}$. Actually, in general, we take a slightly different approach, and require that the true value of these bits be hard to distinguish from a random value. That is, a *polynomial-time* function, h, is called a hard-core of a function f if no efficient algorithm can distinguish $(f(x), h(x))$ from $(f(x), r)$, where r is a random string of length $|h(x)|$. For further discussion of the notion of efficient distinguishability the reader is referred to Section **??**. We assume for simplicity that h is length regular (see below).

Definition 2.5.5 (hard-core function): *Let $h : \{0,1\}^* \rightarrow \{0,1\}^*$ be a polynomial-time computable function, satisfying $|h(x)| = |h(y)|$ for all $|x| = |y|$, and let $l(n) \stackrel{\rm def}{=} |h(1^n)|$. The function h is called a* hard-core *of a function f if for every probabilistic polynomial-time algorithm D', every positive polynomial $p(\cdot)$, and all sufficiently large n's*

$$\left|\Pr\left[D'(f(X_n), h(X_n)) = 1\right] - \Pr\left[D'(f(X_n), R_{l(n)}) = 1\right]\right| < \frac{1}{p(n)}$$

where X_n and $R_{l(n)}$ are two independent random variables the first uniformly distributed over $\{0,1\}^n$, and the second uniformly distributed over $\{0,1\}^{l(n)}$,

For $l \equiv 1$, Definition 2.5.5 is equivalent to Definition 2.5.1; see discussion following Lemma 2.5.8. See also Exercise **??**.

Simple hard-core functions with logarithmic length (i.e., $l(n) = O(\log n)$) are known for the RSA, Rabin, and DLP collections provided that the corresponding collections are one-way. For example, the function which outputs logarithmically many least significant bit is a hard-core function for the RSA collection, provided that the RSA collection is one-way. Namely, assuming that the RSA collection is one-way, it is infeasible to distinguish, given $RSA_{N,e}(x) = x^e \bmod N$, the $O(\log |N|)$ least significant bit of x from a uniformly distributed $O(\log |N|)$-long bit string. (Similar statements hold for the Rabin and DLP collections.) A general result of this type follows.

Theorem 2.5.6 *Let f be an arbitrary strong one-way function, and let g_2 be defined by $g_2(x,s) \stackrel{\rm def}{=} (f(x), s)$, where $|s| = 2|x|$.*[3] *Let $b_i(x,s)$ denote the inner-product mod 2 of the binary vectors x*

[2] We further modify algorithm A' by setting $2^l = O(1/\varepsilon^2)$ (rather than $2^l = O(n/\varepsilon^2)$). Under the new setting, with constant probability, we recover correctly a constant fraction of the bits of x (rather than all of them). If x were an codeword under an asymptotically good error-correcting code (cf., [?]), this would suffice. To avoid this assumption, we modify algorithm A' so that it tries to recover certain XORs of bits of x (rather than individual bits of x). Specifically, we use an asymptotically good linear code (i.e., having constant rate, correcting a constant fraction of errors and having efficient decoding algorithm). Thus, the modified A' recovers correctly a constant fraction of the bits in the encoding of x, under such a code, and using the decoding algorithm – recovers x.

[3] In fact, we may use $|s| = |x| + l(|x|) - 1$, where $l(n) = O(\log n)$. In the current description, s_1 as well as $s_{n+l(n)+1}, ..., s_{2n}$ are not used. However, the current formulation allows not to to specify l when defining g_2.

and $(s_{i+1},...,s_{i+n})$, *where* $s = (s_1,...,s_{2n})$. *Then, for any constant* $c > 0$, *the function* $h(x,s) \stackrel{\text{def}}{=} b_1(x,s)\cdots b_{l(|x|)}(x,s)$ *is a hard-core of the function* g_2, *where* $l(n) \stackrel{\text{def}}{=} \min\{n, \lceil c\log_2 n\rceil\}$.

The proof of the theorem follows by combining a *proposition that capitalizes on the structure of the specific function* h and a *general lemma* concerning hard-core functions. Loosely speaking, the proposition "reduces" the problem of approximating $b(x,r)$ given $g(x,r)$ to the problem of approximating the exclusive-or of any non-empty set of the bits of $h(x,s)$ given $g_2(x,s)$, where b and g are the hard-core and the one-way function presented in the previous subsection. Since we know that the predicate $b(x,r)$ cannot be approximated from $g(x,r)$, we conclude that no exclusive-or of the bits of $h(x,s)$ can be approximated from $g_2(x,s)$. The general lemma implies that, for every "logarithmically shrinking" function h' (i.e., h' satisfying $|h'(x)| = O(\log|x|)$), the function h' is a hard-core of a function f' if and only if the exclusive-or of any non-empty subset of the bits of h' cannot be approximated from the value of f'. Following are the formal statements and proofs of both claims.

Proposition 2.5.7 *Let* f, g_2, l *and the* b_i*'s be as in Theorem 2.5.6. Let* $\{I_n \subseteq \{1,2,...,l(n)\}\}_{n\in\mathbb{N}}$ *be an arbitrary sequence of non-empty sets, and let* $b_{I_{|x|}}(x,s) \stackrel{\text{def}}{=} \oplus_{i\in I_{|x|}} b_i(x,s)$. *Then, for every probabilistic polynomial-time algorithm* A', *every positive polynomial* $p(\cdot)$, *and all sufficiently large* n*'s*

$$\Pr\left[A'(I_n, g_2(U_{3n})) = b_{I_n}(U_{3n})\right] < \frac{1}{2} + \frac{1}{p(n)}$$

where U_{3n} *is a random variable uniformly distributed over* $\{0,1\}^{3n}$.

Proof: The proof is by a reducibility argument. Let X_n, R_n and S_{2n} be independent random variable, uniformly distributed over $\{0,1\}^n$, $\{0,1\}^n$, and $\{0,1\}^{2n}$, respectively. We show that the problem of approximating $b(X_n,R_n)$ given $(f(X_n),R_n)$ is reducible to the problem of approximating $b_{I_n}(X_n,S_{2n})$ given $(f(X_n),S_{2n})$. The underlying observation is that, for every $|s| = 2\cdot|x|$ and every $I \subseteq \{1,...,l(n)\}$,

$$b_I(x,s) = \oplus_{i\in I} b_i(x,s) = b(x, \oplus_{i\in I}\mathrm{sub}_i(s))$$

where $\mathrm{sub}_i(s_1,...,s_{2n}) \stackrel{\text{def}}{=} (s_{i+1},...,s_{i+n})$. Furthermore, the reader can verify that for every non-empty $I \subseteq \{1,...,l(n)\}$, the random variable $\oplus_{i\in I}\mathrm{sub}_i(S_{2n})$ is uniformly distributed over $\{0,1\}^n$, and that given a string $r \in \{0,1\}^n$ and such a set I one can efficiently select a string uniformly in the set $\{s : \oplus_{i\in I}\mathrm{sub}_i(s) = r\}$. Verification of both claims is left as an exercise.[4]

Now, assume to the contradiction, that there exists an efficient algorithm A', a polynomial $p(\cdot)$, and an infinite sequence of sets (i.e., I_n's) and n's so that

$$\Pr\left[A'(I_n, g_2(U_{3n})) = b_{I_n}(U_{3n})\right] \geq \frac{1}{2} + \frac{1}{p(n)}$$

We first observe that for n's satisfying the above inequality we can easily find a set I satisfying

$$p_I \stackrel{\text{def}}{=} \Pr\left[A'(I, g_2(U_{3n})) = b_I(U_{3n})\right] \geq \frac{1}{2} + \frac{1}{2p(n)}$$

[4] Given any non-empty I and any $r = r_1\cdots r_n \in \{0,1\}^n$, consider the following procedure, where k is the largest element in I. First, uniformly select $s_1,...,s_k,s_{k+n+1},...,s_{2n} \in \{0,1\}$. Next, going from $i=1$ to $i=n$, determine s_{k+i} so that $\oplus_{j\in I}s_{i+j} = r_i$ (i.e., $s_{k+i} \leftarrow r_i \oplus (\oplus_{j\in I\setminus\{k\}}s_{j+i})$, where the relevant s_{i+j}'s are already determined, since $j<k$). This process determines a string $s_1\cdots s_{2n}$ uniformly among 2^n strings s that satisfy $\oplus_{i\in I}\mathrm{sub}_i(s) = r$. Since there are 2^n possible r's, both claims follow.

Specifically, we may try all possible I's and estimate p_I for each of them (via random experiments), picking an I for which the estimate is highest. (Note that using $\text{poly}(n)$ many experiments, we may approximate each of the possible $2^{l(n)} - 1 = \text{poly}(n)$ many p_I's up-to an additive deviation of $1/4p(n)$ and error probability of 2^{-n}.)

We now present an algorithm for approximating $b(x, r)$, from $y \stackrel{\text{def}}{=} f(x)$ and r. On input y and r, the algorithm first finds a set I as described above (this stage depends only on $n \stackrel{\text{def}}{=} |x|$ which equals $|r|$). Once I is found, the algorithm uniformly selects a string s so that $\oplus_{i \in I}\text{sub}_i(s) = r$, and returns $A'(I, (y, s))$.

Note that for uniformly distributed $r \in \{0,1\}^n$, the string s selected by our algorithm is uniformly distributed in $\{0,1\}^{2n}$, and $b(x, r) = b_I(x, s)$. Evaluation of the success probability of this algorithm is left as an exercise. ■

The following lemma provides a generic transformation of algorithms distinguishing between $(f(X_n), h(X_n))$ and $(f(X_n), R_{l(n)})$ to algorithms that given $f(X_n)$ and a random non-empty subset I of $\{1, ..., l(n)\}$ predict the XOR of the bits of X_n at locations I.

Lemma 2.5.8 (Computational XOR Lemma): *Let f and h be arbitrary length regular functions, and let $l(n) \stackrel{\text{def}}{=} |h(1^n)|$. Let D be any algorithm, and denote*

$$p \stackrel{\text{def}}{=} \Pr\left[D(f(X_n), h(X_n)) = 1\right] \quad \textit{and} \quad q \stackrel{\text{def}}{=} \Pr\left[D(f(X_n), R_{l(n)}) = 1\right]$$

where X_n and R_l are independent random variable uniformly distributed over $\{0,1\}^n$ and $\{0,1\}^{l(n)}$, respectively. We consider a specific algorithm, denoted $G \stackrel{\text{def}}{=} G_D$, that uses D as a subroutine. Specifically, on input y, $S \subseteq \{1, ..., l(n)\}$ (and $l(n)$), algorithm G selects $r = r_1 \cdots r_{l(n)}$ uniformly in $\{0,1\}^{l(n)}$, and outputs $D(y, r) \oplus 1 \oplus (\oplus_{i \in S} r_i)$. Then,

$$\Pr\left[G(f(X_n), I_l, l(n)) = \oplus_{i \in I_l}(h_i(X_n))\right] = \frac{1}{2} + \frac{p - q}{2^{l(n)} - 1}$$

where I_l is a randomly chosen non-empty subset of $\{1, ..., l(n)\}$ and $h_i(x)$ denotes the i^{th} bit of $h(x)$.

It follows that, for logarithmically shrinking h's, the existence of an efficient algorithm that distinguishes (with a gap that is not negligible in n) the random variables $(f(X_n), h(X_n))$ and $(f(X_n), R_{l(n)})$ implies the existence of an efficient algorithm that approximates the exclusive-or of a random non-empty subset of the bits of $h(X_n)$ from the value of $f(X_n)$ with an advantage that is not negligible. On the other hand, it is clear that any efficient algorithm, that approximates an exclusive-or of a random non-empty subset of the bits of h from the value of f, can be easily modified to distinguish $(f(X_n), h(X_n))$ from $(f(X_n), R_{l(n)})$. Hence, for logarithmically shrinking h's, the function h is a hard-core of a function f if and only if the exclusive-or of any non-empty subset of the bits of h cannot be approximated from the value of f.

Proof: All that is required is to evaluate the success probability of algorithm G (as a function of $p - q$). We start by fixing an $x \in \{0,1\}^n$ and evaluating $\Pr[G(f(x), I_l, l) = \oplus_{i \in I_l}(h_i(x))]$, where I_l is a uniformly chosen non-empty subset of $\{1, ..., l\}$ and $l \stackrel{\text{def}}{=} l(n)$. The rest is an easy averaging (over the x's).

Let $\mathcal{C}$ denote the set (or class) of all non-empty subsets of $\{1, ..., l\}$. Define, for every $S \in \mathcal{C}$, a relation $\equiv_S$ so that $y \equiv_S z$ if and only if $\oplus_{i \in S} y_i = \oplus_{i \in S} z_i$, where $y = y_1 \cdots y_l$ and $z = z_1 \cdots z_l$. Note that for every $S \in \mathcal{C}$ and $z \in \{0,1\}^l$, the relation $y \equiv_S z$ holds for exactly 2^{l-1} of the y's. Recall that by definition of G, on input $(f(x), S, l)$ and random choice $r = r_1 \cdots r_l \in \{0,1\}^l$, algorithm G outputs $D(f(x), r) \oplus 1 \oplus (\oplus_{i\in S} r_i)$. The latter equals $\oplus_{i\in S}(h_i(x))$ if and only if one of the following two disjoint events occurs:

event1: $D(f(x), r) = 1$ and $r \equiv_S h(x)$;

event2: $D(f(x), r) = 0$ and $r \not\equiv_S h(x)$;

By the above discussion and elementary manipulations, we get

$$\begin{aligned}
s(x) &\stackrel{\text{def}}{=} \mathsf{Pr}[G(f(x), I_l, l) = \oplus_{i\in I_l}(h_i(x))] \\
&= \frac{1}{|\mathcal{C}|} \cdot \sum_{S\in\mathcal{C}} \mathsf{Pr}[G(f(x), S, l) = \oplus_{i\in S}(h_i(x)] \\
&= \frac{1}{|\mathcal{C}|} \cdot \sum_{S\in\mathcal{C}} (\mathsf{Pr}[\text{event1}] + \mathsf{Pr}[\text{event2}]) \\
&= \frac{1}{2\cdot|\mathcal{C}|} \cdot \sum_{S\in\mathcal{C}} (\mathsf{Pr}[\Delta(R_l) = 1 \mid R_l \equiv_S h(x)] + \mathsf{Pr}[\Delta(R_l) = 0 \mid R_l \not\equiv_S h(x)])
\end{aligned}$$

where R_l is uniformly distributed over $\{0,1\}^l$ (representing the random choice of algorithm G), and $\Delta(r)$ is a shorthand for the random variable $D(f(x), r)$. The rest of the analysis is straightforward but tedious, and can be skipped with little loss.

$$\begin{aligned}
s(x) &= \frac{1}{2} + \frac{1}{2|\mathcal{C}|} \cdot \sum_{S\in\mathcal{C}} (\mathsf{Pr}[\Delta(R_l) = 1 \mid R_l \equiv_S h(x)] - \mathsf{Pr}[\Delta(R_l) = 1 \mid R_l \not\equiv_S h(x)]) \\
&= \frac{1}{2} + \frac{1}{2|\mathcal{C}|} \cdot \frac{1}{2^{l-1}} \cdot \left(\sum_{S\in\mathcal{C}} \sum_{r\equiv_S h(x)} \mathsf{Pr}[\Delta(r) = 1] - \sum_{S\in\mathcal{C}} \sum_{r\not\equiv_S h(x)} \mathsf{Pr}[\Delta(r) = 1] \right) \\
&= \frac{1}{2} + \frac{1}{2^l \cdot |\mathcal{C}|} \cdot \left(\sum_{r} \sum_{S\in EQ(r,h(x))} \mathsf{Pr}[\Delta(r) = 1] - \sum_{r} \sum_{S\in NE(r,h(x))} \mathsf{Pr}[\Delta(r) = 1] \right)
\end{aligned}$$

where $EQ(r,z) \stackrel{\text{def}}{=} \{S \in \mathcal{C} : r \equiv_S z\}$ and $NE(r,z) \stackrel{\text{def}}{=} \{S \in \mathcal{C} : r \not\equiv_S z\}$. Observe that for every $r \neq z$ it holds that $|NE(r,z)| = 2^{l-1}$ (and $|EQ(r,z)| = 2^{l-1} - 1$). On the other hand, $EQ(z,z) = \mathcal{C}$ (and $NE(z,z) = \emptyset$) holds for every z. Hence, we get

$$\begin{aligned}
s(x) &= \frac{1}{2} + \frac{1}{2^l|\mathcal{C}|} \sum_{r\neq h(x)} \left((2^{l-1} - 1) \cdot \mathsf{Pr}[\Delta(r) = 1] - 2^{l-1} \cdot \mathsf{Pr}[\Delta(r) = 1] \right) \\
&\quad + \frac{1}{2^l|\mathcal{C}|} \cdot |\mathcal{C}| \cdot \mathsf{Pr}[\Delta(h(x)) = 1] \\
&= \frac{1}{2} - \frac{1}{2^l|\mathcal{C}|} \sum_{r\neq h(x)} \mathsf{Pr}[\Delta(r) = 1] + \left(\frac{1}{|\mathcal{C}|} - \frac{1}{2^l|\mathcal{C}|} \right) \cdot \mathsf{Pr}[\Delta(h(x)) = 1]
\end{aligned}$$

where the last equality uses $|\mathcal{C}| = 2^l - 1$ (i.e., $\frac{1}{2^l} = \frac{1}{|\mathcal{C}|} - \frac{1}{2^l|\mathcal{C}|}$). Re-arranging the terms and substituting for Δ, we get

$$\begin{aligned}
s(x) &= \frac{1}{2} + \frac{1}{|\mathcal{C}|} \cdot \mathsf{Pr}[\Delta(h(x)) = 1] - \frac{1}{2^l|\mathcal{C}|} \sum_{r} \mathsf{Pr}[\Delta(r) = 1] \\
&= \frac{1}{2} + \frac{1}{|\mathcal{C}|} \cdot (\mathsf{Pr}[D(f(x), h(x)) = 1] - \mathsf{Pr}[D(f(x), R_l) = 1])
\end{aligned}$$

Finally, taking the expectation over the x's, we get

$$\begin{aligned}
\mathsf{E}[s(X_n)] &= \frac{1}{2} + \frac{1}{|\mathcal{C}|} \cdot (\mathsf{Pr}[D(f(X_n), h(X_n)) = 1] - \mathsf{Pr}[D(f(X_n), R_l) = 1]) \\
&= \frac{1}{2} + \frac{1}{2^l - 1} \cdot (p - q)
\end{aligned}$$

and the lemma follows. ■