1 Hard-Core Bits

Even though a function may be one-way, given \( f(x) \) it may be possible to learn a great deal about \( x \). (Consider, for example, the subset sum candidate one-way function \( f(x_1, \ldots, x_n, I) = (x_1, \ldots, x_n, \sum_{i \in I} x_i) \).

**Definition 1.1.** A function \( B : \{0, 1\}^* \rightarrow \{0, 1\} \) is a hard-core bit for a function \( f \) if and only if for every PPT \( A \), the function \( \epsilon : \mathbb{N} \rightarrow \mathbb{N} \) is negligible where

\[
\epsilon(n) = \Pr[A(f(x)) = B(x) \mid x \leftarrow U_n] - 1/2.
\]

That is, \( B \) is a hard-core bit if and only if it is computationally infeasible to predict \( B(x) \) given \( f(x) \) with probability significantly better than \( 1/2 \). It is trivial to predict any value with probability \( \geq 1/2 \) since random guessing ensures that the success rate is exactly \( 1/2 \). Observe also that by this definition, if \( B \) is efficiently computable, the value of \( B(x) \) must be very close to being balanced on inputs in \( U_n \) since otherwise a guess that succeeds with probability above \( 1/2 \) can be made by evaluating \( B \) on a number of random \( y \) in \( U_n \) and outputting the majority answer of the \( B(y) \).

If a function \( f \) loses information about \( x \) then it can be easy to produce a hard-core bit for \( f \). For example, suppose that \( f(x) \) produces all but the last bit of \( x \) and \( B(x) \) is that last bit. That is not the kind of case we will be interesting in. We will typically consider functions \( f \) that do not lose this any information (for example functions that are permutations on the set of inputs of length \( n \)) and in this case, in order for \( B \) to be hard-core for \( f \), \( f \) must be a one-way function.

The following alternative definition of hard-core bit can be seen to be equivalent to the original definition and, although it is more complicated, it is more convenient for analysis.

**Definition 1.2.** A function \( B : \{0, 1\}^* \rightarrow \{0, 1\} \) is a hard-core bit for a function \( f \) if and only if for every PPT \( A' \), the function \( \epsilon' : \mathbb{N} \rightarrow \mathbb{N} \) is negligible where

\[
\epsilon'(n) = \Pr[A'(f(x), B(x)) = 1 \mid x \leftarrow U_n] - \Pr[A'(f(x), b) = 1 \mid x \leftarrow U_n, b \leftarrow U_1].
\]

Clearly if we define \( A'(y, b) \) to run \( A \) on input \( y \) and output 1 if and only if \( A(y) \) outputs \( b \), then \( \Pr[A'(f(x), b) = 1 \mid x \leftarrow U_n, b \leftarrow U_1] = 1/2 \) and \( \Pr[A'(f(x), B(x)) = 1 \mid x \leftarrow U_n] = \Pr[A(f(x)) = B(x) \mid x \leftarrow U_n] \) so \( \epsilon'(n) \) from this definition is precisely the same as \( \epsilon(n) \) from the previous definition so this definition is at least as strong as the earlier one. One can also show the reverse implication by observing that \( \Pr[A'(f(x), b)] = 1 \mid x \leftarrow U_n, b \leftarrow U_1 \) is the average of the distributions conditioned on \( b = B(x) \) and \( b = 1 - B(x) \).
This latter definition looks very much our definitions of statistical indistinguishability, except that in trying to distinguish $B(x)$ from a random $b$, $A'$ is given $f(x)$ as advice. Using this latter definition we can extend the notion of hard-core bits to hard-core functions.

**Definition 1.3.** A function $H : \{0,1\}^* \to \{0,1\}^{n(n)}$ is hard-core for a function $f$ if and only if for every PPT $A'$, the function $\epsilon' : \mathbb{N} \to \mathbb{N}$ is negligible where

$$\epsilon'(n) = \Pr[A'(f(x), H(x)) = 1 | x \leftarrow U_n] - \Pr[A'(f(x), \vec{b}) = 1 | x \leftarrow U_n, \vec{b} \leftarrow U_{m(n)}].$$

Similar notions of hard-core bits and hard-core functions can be defined for collections of functions but for simplicity we do not state them formally. As we will see, if our candidate collections of one-way functions are indeed one-way then each has a natural hard-core bit.

### 1.1 Hard-Core Bits for Candidate Functions

Define $\text{LSB}_k(x)$ to be the $k$ least-significant bits of $x \in \{0,1\}^n$ and define $\text{LSB}(x) = \text{LSB}_1(x)$. Similarly for $p$ a prime and $x \in \mathbb{Z}_{p-1}$ define the most significant bit of $x$,

$$\text{MSB}_p(x) = \begin{cases} 1 & (p-1)/2 \leq x \leq p-2 \\ 0 & 0 \leq x < (p-1)/2 \end{cases}.$$ 

Observe that for $g$ a generator of $\mathbb{Z}_p^*$, $1 = g^{(p-1)} \mod p = (g^{(p-1)/2})^2 \mod p$ but $g^{(p-1)/2} \not\equiv 1 (\mod p)$. Thus $g^{(p-1)/2} \equiv -1 (\mod p)$ and we can write

$$\mathbb{Z}_p^* = \{1, g, g^2, \ldots, g^{(p-1)/2} = -1, -g, -g^2, \ldots, -g^{p/2-1}\}.$$

**Lemma 1.4** (Blum-Micali 1982). If $\text{EXP}_{(p,g)}$ (a.k.a. $\text{DLP}_{(p,g)}$) is one-way then $\text{MSB}(x)$ is a hard-core bit for $\text{EXP}_{(p,g)}$.

**Proof Sketch.** The basic idea of the argument is that if one has an algorithm that can determine $\text{MSB}_p(x)$ from $\text{EXP}_{(p,g)} = g^x \mod p$ then one actually invert $\text{EXP}_{(p,g)}$. We use the following two facts:

- Given $z$ such that $z$ is a square modulo $p$, there is a randomized algorithm that will find an $w$ such that $w^2 \equiv z \pmod{p}$. (This is known as the Tonelli-Shanks algorithm.)

- $y$ is a square modulo $p$ if and only if $y = g^{2k} \pmod{p}$ for some integer $k$ and thus if and only if $y^{(p-1)/2} \equiv 1 \pmod{p}$.

We now describe the algorithm. Given $y = g^x \pmod{p}$, we can determine the low order bit of $x$ simply by determining whether $y$ is a square modulo $p$. Now define

$$z = \begin{cases} y & \text{if } y \text{ is a square } \pmod{p} \\ g^{-1}y & \text{if } y \text{ is not a square } \pmod{p} \end{cases}.$$
Clearly $z$ is always square mod $p$ and $z = g^{2k} \mod p$ where $k$ is the integer given by the bits of $x$ shifted right by one bit.

Now, when the square root algorithm is run on $z$ we get one of two square roots of $z$, either $w = g^k$ or $w = -g^k = g^{(p-1)/2+k}$. Thus $w = g^v$ where $v$ is either $k$ or $(p-1)/2 + k$. We really want the former one but just given $w$ we don’t know which case we have. However, if given can find the $\text{MSB}_p(v)$ given $w = g^v$ then we can tell which case we have and simply multiply by $-1$ to obtain $g^k$. This can be repeated to cover each bit of $x$ in turn for a total of $n$ calls where $n$ is the number of bits in $x$.

Similar properties hold for other one-way candidate functions.

**Lemma 1.5** (Blum, Blum, Schub 1982). If $\text{Blum}_N$ is one-way then $\text{LSB}(x)$ is hard-core bit for $\text{Blum}_N$.

**Lemma 1.6** (Alexi, Chor, Goldreich, Schnorr 1983). $\text{LSB}(x)$ is a hard-core bit for $\text{RSA}_{(N,e)}$, $\text{Blum}_N$, $\text{Rabin}_N$ if the corresponding function is one-way. Moreover, For $m = O(\log \log N)$, $\text{LSB}_m(x)$ is hard-core for $\text{RSA}_{(N,e)}$, $\text{Blum}_N$, $\text{Rabin}_N$ if the corresponding function is one-way.

In each of the above cases the number of calls to the hard-core predicate in order to invert the function is $O(n)$ where $n$ is the number of bits in the parameters. As a result the advantage at predicting the hard-core bit must be at most $O(n)$ times the inverting probability for the underlying one-way function. The following result is more recent, much more general, but a fair bit less efficient.

**Lemma 1.7** (Høastad, Naslund 2004). Any block of $\log \log N$ bits of $\text{RSA}_{(N,e)}$ are simultaneously secure.

### 1.2 A Hard-core Bit from any One-Way Function

The following is a general method for deriving hard-core bits from one-way functions.

**Theorem 1.8** (Goldreich-Levin). If $f : \{0,1\}^* \rightarrow \{0,1\}^*$ is a one-way function that is length-preserving (maps $\{0,1\}^n$ to $\{0,1\}^n$) then $B : \{0,1\}^* \rightarrow \{0,1\}^*$ defined by $B(xr) = x \cdot r \mod 2$ where $|x| = |r|$ and $x \cdot r$ is the inner product of $x$ and $r$ is a hard-core bit for the function $g(x,r) = (f(x),r)$.

This theorem is very general and useful although the difference in the predictability of $B$ versus the invertibility of $f$ is cubic and so not as efficient as the specific candidates functions above.

### 2 Pseudorandom Number Generators from One-Way Permutations

Recall that a pseudorandom generator (PRNG) is a deterministic polynomial-time computable function $G : \{0,1\}^* \rightarrow \{0,1\}^*$ that is length-increasing (mapping $n$ bits to $\ell(n)$ bits) and such that for all PPT $A$, 

$$\text{Adv}_{A}^{\text{PRNG},G}(n) = \text{Pr}[A(G(U_n)) = 1] - \text{Pr}[A(U_{\ell(n)}) = 1]$$
is negligible.

The following is a general method for using one-way permutations to build PRNGs. We will prove part (b) next time. Part (a) is an exercise.

**Theorem 2.1.** Let \( f : \{0, 1\}^* \rightarrow \{0, 1\}^* \) such that for all \( n, f : \{0, 1\}^n \rightarrow \{0, 1\}^n \) is a permutation and \( B \) is a hard-core bit for \( f \) that is polynomial-time computable then

(a) \( G : \{0, 1\}^* \rightarrow \{0, 1\}^* \) given by \( G(x) = f(x)B(x) \) is a PRNG with \( \ell(n) = n + 1 \).

(b) For every polynomial \( \ell(n) > n \) the function \( G^\ell : \{0, 1\}^* \rightarrow \{0, 1\}^* \) given by \( G(x) = B(x)B(f(x))B(f(f(x))) \cdots B(f^{\ell(|x|)-1}(x)) \) is a PRNG.