Extracts from

Modern Cryptography, Probabilistic Proofs and Pseudorandomness

Oded Goldreich

Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Rehovot, ISRAEL.

March 25, 1998

1 The context

Loosely speaking, a *polynomial-time* function f is called one-way if any efficient algorithm can invert it only with negligible success probability. For simplicity we consider throughout this section only length preserving one-way functions.

Definition 1 (one-way function): *A* one-way function*, f, is a polynomial-time computable function such that for every probabilistic polynomial-time algorithm A', every positive polynomial $p(\cdot)$, and all sufficiently large k's*

$$\Pr\left[A'(f(U_k)) \in f^{-1}(f(U_k))\right] < \frac{1}{p(k)}$$

We stress that both occurrences of U_k refer to the same random variable. That is, the above asserts that

$$\sum_{x\in\{0,1\}^k} 2^{-k} \cdot \Pr\left[A'(f(x)) \in f^{-1}(f(x))\right] < \frac{1}{p(k)}$$

Popular candidates for one-way functions are based on the conjectured intractability of Integer Factorization (cf., [Ofactor] for state of the art), the Discrete Logarithm Problem (cf., [Odlp] analogously), and decoding of random linear code [GKL]. The infeasibility of inverting f yields a weak notion of unpredictability: For every probabilistic polynomial-time algorithm A (and sufficiently large k), it must be the case that $\Pr_i[A(i, f(U_k)) \neq b_i(U_k)] > 1/2k$, where the probability is taken uniformly over $i \in \{1, ..., k\}$ (and U_k), and $b_i(x)$ denotes the i^{th} bit of x. A stronger (and in fact strongest possible) notion of unpredictability is that of a hard-core predicate. Loosely speaking, a *polynomial-time* predicate b is called a hard-core of a function f if all efficient algorithm, given $f(x)$, can guess $b(x)$ only with success probability which is negligible better than half.

Definition 2 (hard-core predicate [BM]): *A polynomial-time computable predicate* $b : \{0,1\}^* \mapsto \{0,1\}$ *is called a* hard-core *of a function* f *if for every probabilistic polynomial-time algorithm* A'*, every polynomial* $p(\cdot)$*, and all sufficiently large* k*'s*

$$\Pr\left(A'(f(U_k)) = b(U_k)\right) < \frac{1}{2} + \frac{1}{p(k)}$$

Clearly, if b is a hard-core of a 1-1 polynomial-time computable function f then f must be one-way.[1] It turns out that any one-way function can be slightly modified so that it has a hard-core predicate.

Theorem 3 (A generic hard-core [GL]): *Let* f *be an arbitrary 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*.*

A proof is presented below. Our presentation of the proof of Theorem 3 differs from what appears in the original text [GL].

2 Proof of Theorem 3

Theorem 3, due to Goldreich and Levin [GL], relates two computational tasks: The first task is inverting a function f; namely given y find an x so that $f(x) = y$. The second task is predicting, with non-negligible advantage, the exclusive-or of a subset of the bits of x when only given $f(x)$. More precisely, it has been proved that if f cannot be efficiently inverted then given $f(x)$ and r it is infeasible to predict the inner-product mod 2 of x and r better than obvious.

The proof presented here is not the original one presented in [GL] (see generalization in [GRS]), but rather an alternative suggested by Charlie Rackoff. The alternative proof, inspired by [ACGS], has two main advantages over the original one: It is simpler to explain, and it provides better security (i.e., a more efficient reduction of inverting f to predicting the inner-product).

[1] Functions which are not 1-1 may have hard-core predicates of information theoretic nature; but these are of no use to us here. For example, for $\sigma \in \{0,1\}$, $f(\sigma, x) = 0f'(x)$ has an "information theoretic" hard-core predicate $b(\sigma, x) = \sigma$.

Theorem 4 (Theorem 3 – restated): *Suppose we have oracle access to a random process $b_x : \{0,1\}^n \mapsto \{0,1\}$, so that*

$$\Pr_{r\in\{0,1\}^n}[b_x(r) = b(x,r)] \geq \frac{1}{2} + \epsilon$$

where the probability is taken uniformly over the internal coin tosses of b_x and all possible choices of $r \in \{0,1\}^n$, and $b(x,r)$ denote the inner-product mod 2 of the binary vectors x and r. Then, we can in time polynomial in n/ϵ output a list of strings which with probability at least $\frac{1}{2}$ contains x.

Theorem 3 is derived from the above by using standard arguments. We prove this fact first.

Proposition 5 *Theorem 4 implies Theorem 3.*

Proof: The proof uses a "reducibility argument" – inverting the function f is reduced to predicting $b(x,r)$ from $(f(x),r)$. Hence, we assume (for contradiction) the existence of an efficient algorithm predicting the inner-product with advantage which is not negligible, and derive an algorithm that inverts f with related (i.e., not negligible) success probability. This contradicts the hypothesis that f is a one-way function.

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

$$\epsilon_G(n) \stackrel{\text{def}}{=} \Pr\left(G(f(X_n), R_n) = b(X_n, R_n)\right) - \frac{1}{2}$$

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 exists an efficient algorithm G, a polynomial $p(\cdot)$ and an infinite set N so that for every $n\in N$ it holds that $\epsilon_G(n) > \frac{1}{p(n)}$. 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 $\frac{\epsilon(n)}{2}$ fraction of the x's of length n, algorithm G has an $\frac{\epsilon(n)}{2}$ advantage in predicting $b(x, R_n)$ from $f(x)$ and R_n. Namely,

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

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

This time 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 observation follows by an averaging argument. Namely, write $\text{Exp}(s(X_n)) = \frac{1}{2} + \epsilon(n)$, and apply Markov Inequality.$\Box$

Thus, we restrict our attention to x's in S_n. For each such x, the conditions of Theorem 4 hold, and so within time $\text{poly}(n/\epsilon(n))$ and with probability at least $1/2$ we retrieve a list of strings containing x. Contradiction to the one-wayness of f follows, since the probability we invert f on uniformly selected x is at least $\frac{1}{2} \cdot \Pr(U_n \in S_n) \geq \frac{\epsilon(n)}{4}$. ■

2.1 A motivating discussion

Let $s(x) \stackrel{\text{def}}{=} \Pr[b_x(r) = b(x,r)]$, where r is uniformly distributed in $\{0,1\}^{|x|}$. Then, by the hypothesis of Theorem 4, $s(x) \geq \frac{1}{2} + \epsilon$. Suppose, for a moment, that $s(x) > \frac{3}{4} + \epsilon$. In this case, retrieving x by querying the oracle b_x is quite easy. To retrieve the i^{th} bit of x, denoted x_i, we uniformly select $r \in \{0,1\}^n$, and obtain $b_x(r)$ and $b_x(r \oplus e^i)$, 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. Clearly, if both $b_x(r) = b(x,r)$ and $b_x(r \oplus e^i) = b(x, r \oplus e^i)$ then

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

The probability that both equalities hold (i.e., both $b_x(r) = b(x,r)$ and $b_x(r \oplus e^i) = b(x, r \oplus e^i)$) is at least $1 - 2 \cdot (\frac{1}{4} - \epsilon) = \frac{1}{2} + 2\epsilon$. 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 obtain x itself. However, the entire analysis was conducted under (the unjustifiable) assumption that $s(x) > \frac{3}{4} + \epsilon$, whereas we only know that $s(x) > \frac{1}{2} + \epsilon$.

The problem with the above procedure is that it doubles the original error probability of the oracle b_x on random queries. Under the unrealistic assumption, that the b_x's error on such inputs is significantly smaller than $\frac{1}{4}$, the "error-doubling" phenomenon raises no problems. However, in general (and even in the special case where b_x's error is exactly $\frac{1}{4}$) the above procedure is unlikely to yield x. Note that the error probability of b_x can not be decreased by querying b_x several times on the same instance (e.g., b_x 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* b_x – a way which does not double the original error probability of b_x. The key idea is to generate the r's in a way which requires querying b_x only once per each r (and x_i), instead of twice. The good news are that the error probability is no longer doubled, since we will only use b_x to get an "estimate" of $b(x, r \oplus e^i)$. The bad news are that we still need to know $b(x,r)$, and it is not clear how we can know $b(x,r)$ without querying b_x. 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) $b(x,r)$ for polynomially many r's. An obvious way of guessing these $b(x,r)$'s yields an exponentially vanishing success probability. The solution is to 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 non-negligible 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 the $b(x,r)$'s), we can retrieve x with high probability. Hence, we retrieve x with non-negligible 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 $m = \mathrm{poly}(n/\epsilon)$ many r's, we uniformly (and independently) select $l \stackrel{\mathrm{def}}{=} \log_2(m+1)$ 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 use 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 $2^{-l} = \frac{1}{\mathrm{poly}(n/\epsilon)}$. The different r's correspond to the different non-empty subsets of $\{1,2,...,l\}$. We compute $r^J \stackrel{\mathrm{def}}{=} \oplus_{j\in J} s^j$. 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 non-negligible probability all our guesses are correct.

2.2 Back to the formal argument

Following is a formal description of the recovering algorithm, denoted A. On input n and ϵ (and oracle access to b_x), algorithm A sets $l \stackrel{\mathrm{def}}{=} \lceil \log_2(n\cdot\epsilon^{-2}+1)\rceil$. Algorithm A uniformly and independently select $s^1,...,s^l \in \{0,1\}^n$, and $\sigma^1,...,\sigma^l \in \{0,1\}$. It then computes, for every non-empty set $J \subseteq \{1,2,...,l\}$, a string $r^J \leftarrow \oplus_{j\in J} s^j$ and a bit $\rho^J \leftarrow \oplus_{j\in J}\sigma^j$. For every $i \in \{1,...,n\}$ and every *non-empty* $J \subseteq \{1,..,l\}$, algorithm A computes $z_i^J \leftarrow \rho^J \oplus b_x(r^J \oplus e^i)$. Finally, algorithm A sets z_i to be the majority of the z_i^J values, and outputs $z = z_1 \cdots z_n$.

Comment: An alternative implementation of the above ideas results in an algorithm, denoted A', which fits the conclusion of the theorem. Rather than selecting at random a setting of $\sigma^1,...,\sigma^l \in \{0,1\}$, algorithm A' tries all possible values for $\sigma^1,...,\sigma^l$. It outputs a list of 2^l candidates z's, one per each of the possible settings of $\sigma^1,...,\sigma^l \in \{0,1\}$.

Clearly, A makes $n\cdot 2^l = n^2/\epsilon^2$ oracle calls to b_x, and the same amount of other elementary computations. Algorithm A' makes the same queries, but conducts a total of $(n/\epsilon^2)\cdot(n^2/\epsilon^2)$ elementary computations.

Following is a detailed analysis of the success probability of algorithm A. We start by showing that, in case the σ^j's are correct, then 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.

Claim: For every $1 \leq i \leq n$,

$$\Pr\left(|\{J : b(x, r^J) \oplus b_x(r^J \oplus e^i) = x_i\}| > \frac{1}{2} \cdot (2^l - 1) \right) > 1 - \frac{1}{4n}$$

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 $b(x, r^J) \oplus b_x(r^J \oplus e^i) = x_i$. The reader can easily verify that each r^J is uniformly distributed in $\{0,1\}^n$. It follows that each ζ^J equals 1 with probability $\frac{1}{2} + \epsilon$. We show that the ζ^J's are pairwise independent by showing that the r^J's are pairwise independent. For every $J \neq K$ we have, without loss of generality, $j \in J$ and $k \in K \setminus J$. Hence, for every $\alpha, \beta \in \{0,1\}^n$, we have

$$\begin{aligned}
\Pr\left(r^K = \beta \mid r^J = \alpha\right) &= \Pr\left(s^k = \beta \mid s^j = \alpha\right) \\
&= \Pr\left(s^k = \beta\right) \\
&= \Pr\left(r^K = \beta\right)
\end{aligned}$$

and pairwise independence of the r^J's follows. Let $m \stackrel{\text{def}}{=} 2^l - 1$. Using Chebyshev's Inequality, we get

$$\begin{aligned}
\Pr\left(\sum_J \zeta^J \leq \frac{1}{2} \cdot m\right) &\leq \Pr\left(\left|\sum_J \zeta^J - (0.5+\epsilon) \cdot m\right| \geq \epsilon \cdot m\right) \\
&< \frac{\mathrm{Var}(\zeta^{\{1\}})}{\epsilon^{-2} \cdot (n/\epsilon^2)} \\
&< \frac{1}{4n}
\end{aligned}$$

The claim now follows. $\square$

Recall that if $\sigma^j = b(x, s^j)$, for all j's, then $\rho^J = b(x, r^J)$ for all non-empty J's. In this case z output by algorithm A equals x, with probability at least $3/4$. However, the first event happens with probability $2^{-l} = \frac{1}{n/\epsilon^2}$ independently of the events analyzed in the Claim. Hence, algorithm A recovers x with probability at least $\frac{3}{4} \cdot \frac{\epsilon^2}{n}$ (whereas, the modified algorithm, A', succeeds with probability at least $\frac{3}{4}$). Theorem 4 follows. $\blacksquare$

2.3 Improved Implementation of Algorithm A'

In continuation to the proof of Theorem 4, we present guidelines for a more efficient implementation of Algorithm A'. In the sequel it will be more convenient to use arithmetic of reals instead of that of Boolean values. Hence, we denote $b'(x,r) = (-1)^{b(r,x)}$ and $b'_x(r) = (-1)^{b_x(r)}$.

1. Prove that $\mathrm{Exp}_r(b'(x,r) \cdot b'_x(r+e^i)) = 2\epsilon \cdot (-1)^{x_i}$, where $\epsilon = \Pr_r(b_x(r) = b(x,r)) - 0.5$.
2. Let v be an l-dimensional Boolean vector, and let R be a uniformly chosen l-by-n Boolean matrix. Prove that 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 $\{0,1\}^n$.

   (Note that each such vR corresponds to a r^J above, with $J = \{j : v_j = 1\}$.)
3. Prove that $b'(x, vR) = b'(xR^T, v)$, for every $x \in \{0,1\}^n$ and $v \in \{0,1\}^l$.

   (This enables to compute the $b'(x, vR)$'s via the $b'(xR^T, v)$'s.)
4. Prove that, with probability at least $\frac{1}{2}$ over the choices of R, there exists $u \in \{0,1\}^l$ so that for every $1 \le i \le n$ the sign of $\sum_{v \in \{0,1\}^l} b'(u,v) b'_x(vR + e^i))$ equals the sign of $(-1)^{x_i}$.

   (Hint: Re-do the proof of the Claim of subsection 2.2, using $u \stackrel{\rm def}{=} xR^T$.)
5. Let B be a fixed 2^l-by-2^l matrix with the (u,v)-entry being $b'(u,v)$, and denote by $\overline{\sigma}^i$ an 2^l-dimensional vector with the $v^{\rm th}$ entry equal $b'_x(vR + e^i)$. Consider an algorithm that computes $\overline{z}_i \leftarrow B\overline{\sigma}^i$, for all i's, and forms a 2^l-by-n matrix Z in which the columns are the $\overline{z}_i$'s. The output is the list of rows in Z.

   (Notice that the algorithm makes $2^l \cdot n$ queries to obtain all entries in the $\overline{\sigma}^i$'s, that all these queries can be computed within $2^l n$ time, and so all that remains is to multiply the fixed matrix B by n vectors.)

   (a) Using Item 4, evaluate the success probability of the algorithm (i.e., the probability that x is in the output list). .

   (b) Using the special structure of the fixed matrix B, show that the product $B\overline{\sigma}^i$ can be computed in time $l \cdot 2^l$.

   Hint: B is the Sylvester matrix, which 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.

It follows that algorithm A' can be implemented in time $n \cdot l2^l$, which is $\widetilde{O}(n^2/\epsilon^2)$.

Further Improvement. We may further improve algorithm A' by observing that it suffices to let $2^l = O(1/\epsilon^2)$ rather than $2^l = O(n/\epsilon^2)$. 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 encoding of some w, s.t. $|x| = O(|w|)$, under an asymptotically good error-correcting code, this would suffice. To remove this assumption, we may modify the hardcore so that $\mathsf{b}(w,r)$ is the inner-product of the encoding of w, denoted $C(w)$, and r (where $|r| = |C(w)|$). Furthermore, using a linear error-correcting code $C(w) = Aw$, we can write $\mathsf{b}(w,r) = b(Aw,r) = b(w, A^T r)$, and so the entire algorithm can be emulated in terms of an oracle b_w which is ϵ-correlated with $b(w,\cdot)$. Thus, given such an oracle b_w, and an additional oracle χ_w so that $\chi_w(y) = 1$ iff $y = w$, we can recover w using $O(|w|/\epsilon^2)$ oracle queries (and a similar amount of other elementary operations). This is *optimal* in the sense that each oracle answer provides only $O(\epsilon^2)$ bits of information.