October 7, 1999
Author: Bellare

The Goldreich-Levin Theorem

1 The problem

We fix a an integer n for the length of the strings involved. If a is an n-bit string and $1 \leq i \leq n$ then $a^{(i)}$ denotes the i-th bit of a. If a, b are n-bit strings then

$$\langle a, b \rangle = a^{(1)}b^{(1)} + a^{(2)}b^{(2)} + \cdots a^{(n)}b^{(n)}$$

denotes the inner product of a and b. The operations here are modulo two, meaning we work over the finite field of two elements, so the value above is a bit.

We are given an oracle B_x: $\{0,1\}^n \to \{0,1\}$ and a real number $\epsilon > 0$ such that

$$\Pr\left[\, \mathsf{B}_x(r) = \langle x, r \rangle \;:\; r \overset{R}{\leftarrow} \{0,1\}^n \,\right] = \frac{1}{2} + \frac{\epsilon}{2}\,. \tag{1}$$

We call ϵ the *advantage* of B_x. We are not directly given x.

We are also given another oracle GD_x: $\{0,1\}^n \to \{0,1\}$ with the property that $\mathsf{GD}_x(x) = 1$. We make no requirement on what GD_x returns on inputs different from x. Oracle GD_x is defining our task which becomes, given these two oracles, to find some x' such that $\mathsf{GD}_x(x') = 1$. We want to figure out how to do it and also what is the complexity.

A particular case is that $\mathsf{GD}_x = \mathsf{EQ}_x$ is the equality oracle which given any $y \in \{0,1\}^n$ returns 1 if $y = x$ and 0 otherwise. In this case, our task is to find x itself. One should have this in mind for intuition, so, as we proceed, think of the task as being to recover x.

Returning to our task, stated more precisely, we wish to design an algorithm A that given the above two oracles $\mathsf{B}_x, \mathsf{GD}_x$ returns a string x'. The *success probability* of A is the probability that $\mathsf{GD}_x(x') = 1$, taken over the coin tosses of A. We seek A having success probability at least 1/2. We let q_B denote the number of calls made by A to B_x and q_E the number of calls to GD_x. We let t be the running time plus the size of the code of A in some fixed RAM model of computation. (Alternatively, A is a circuit and t is the size of the circuit.) We want to figure out q_B, q_E, t as functions of n, ϵ. Certainly we want them all to be poly$(n, 1/\epsilon)$, but we want to know exactly what is the polynomial here. The algorithm should work for any $x \in \{0,1\}^n$.

We would like actually something slightly more general. We would like to view q_B, q_E, t as given, and lower bound the success probability of A as a function of n, ϵ, q_B, q_E, t. But this problem does not appear to have been studied.

2 Background

The context of Goldreich and Levin [5] is to find a hard-core predicate for any one-way function. Given a length-preserving one-way function f: $\{0,1\}^* \to \{0,1\}^*$, define $F(x,r) = (f(x), r)$ where $|x| = |r|$. This is also a one-way function. Now the claim is that $\langle x, r\rangle$ is a hard-core predicate for this function. This means that if there was an efficient algorithm to predict $\langle x, r\rangle$ given $f(x), r$, there is also an efficient algorithm to compute a pre-image of $f(x)$ given $f(x)$. Probabilities here are taken over the random choice of x and r. The technical part of the reduction amounts to the above problem. The given algorithm for predicting $\langle x, r\rangle$ is B_x, and we let $\mathsf{GD}_x(x')$ return 1 if $f(x') = f(x)$, and 0 otherwise. This application shows why, in general, GD_x can be different from EQ_x. They coincide when f is a permutation, but not otherwise.

The proof in Section 4 is due to Rackoff, using ideas of [1]. It is a simplification of the original proof of [5]. It is along the same lines as the proof in [7]. Two other excellent sources are Goldreich's survey [3] and book [4], which present a proof using the same ideas and also present security improvements. A recent paper of Levin [6] might have further security improvements. It would be nice to read [3, 4, 6] and figure out the improvements.

3 The high advantage case

The oracle B_x partitions the set of n-bit strings into two parts. The "good" strings are those inputs on which the oracle is correct and the bad strings are those inputs on which the oracle is wrong. It useful to name these sets:

$$\begin{aligned} \mathsf{GdS} &= \{\, s \in \{0,1\}^n \;:\; \mathsf{B}_x(s) = \langle x, s\rangle \,\} \\ \mathsf{BdS} &= \{\, s \in \{0,1\}^n \;:\; \mathsf{B}_x(s) \neq \langle x, s\rangle \,\} \,. \end{aligned}$$

Our assumption can then equivalently be stated as

$$|\mathsf{GdS}| = \frac{1+\epsilon}{2}\cdot 2^n \quad \text{and} \quad |\mathsf{BdS}| = \frac{1-\epsilon}{2}\cdot 2^n \,.$$

This will help us think about the problem.

Recall our problem is to find x' such that $\mathsf{GD}_x(x') = 1$, given oracle access to B_x and GD_x. We take our task as being to find x itself, which by assumption satisfies $\mathsf{GD}_x(x) = 1$. First assume that B_x is always correct, meaning has advantage $\epsilon = 1$. In other words $\mathsf{GdS} = \{0,1\}^n$, meaning we simply have an oracle which given any n-bit string r returns $\langle x, r\rangle$. How can we find x?

For $i = 1, \ldots, n$ let e_i denote the string having a one in position i and zeros elsewhere. Observe that $x^{(i)} = \langle x, e_i\rangle$. So it suffices to make the queries $e_1, \ldots, e_n$ to B_x to compute x. We did not even need the GD_x oracle.

Now suppose the advantage of B_x is less than 1, but still very close to 1. Let $\delta = 1 - \epsilon$. This by assumption is small, close to 0. A first thought is to proceed as above; we make queries $e_1, \ldots, e_n$ to B_x. But the probability of success here could be zero. Even though B_x is correct on most inputs, these particular inputs may not be among them. Meaning, even though GdS occupies a $1 - \delta$ fraction of $\{0,1\}^n$, it could still be true that some or all of the points $e_1, \ldots, e_n$ are in BdS.

If we want any chance of success, we must only invoke B_x on random points, so that we have a chance of falling in GdS. This leads to the idea of using *self-correction* (cf. [2]). The algorithm of

```
Algorithm $\mathrm{SC}^{\mathsf{B}_x}(z)$
    $r \overset{R}{\leftarrow} \{0,1\}^n$
    $b_1 \leftarrow \mathsf{B}_x(z+r)$ ; $b_2 \leftarrow \mathsf{B}_x(r)$
    Return $b_1 - b_2$
```

Figure 1: The SC algorithm that attempts to compute $\langle x, z \rangle$ given z.

Figure 1 takes as input any n-bit string z and attempts to compute $\langle x, z \rangle$ by invoking B_x only on random points, each individually unrelated to z. Remember that arithmetic operations are modulo two.

To analyze the algorithm, observe that the linearity of the inner product function tells us that $\langle x, z \rangle = \langle x, z+r \rangle - \langle x, r \rangle$ for any n-bit string r. If r is random, so is $z + r$. The two are not independent, but it is still true that both, individually, are uniformly distributed, and that's what we will use. The probability below is over the random choice of r made by the algorithm of Figure 1.

$$\begin{aligned}
\Pr\left[\, b_1 - b_2 \neq \langle x, z \rangle \,\right] &\leq \Pr\left[\, \mathsf{B}_x(z+r) \neq \langle x, z+r \rangle \text{ or } \mathsf{B}_x(r) \neq \langle x, r \rangle \,\right] \\
&= \Pr\left[\, z+r \in \mathsf{BdS} \text{ or } r \in \mathsf{BdS} \,\right] \\
&\leq \Pr\left[\, z+r \in \mathsf{BdS} \,\right] + \Pr\left[\, r \in \mathsf{BdS} \,\right] \\
&= 2 \cdot \left(\frac{1}{2} - \frac{\epsilon}{2} \right) \\
&= 1 - \epsilon \\
&= \delta \ .
\end{aligned}$$

In other words, our algorithm is correct except with probability δ. This is quite nice since its input z is not necessarily random. In particular z might be in BdS.

To find x we use the same observation as above, namely that it suffices to find the n bits $\langle x, e_i \rangle$ for $i = 1, \ldots, n$. Do this by calling $\mathrm{SC}^{\mathsf{B}_x}(e_i)$ for $i = 1, \ldots, n$. (Note that each call results in a new random choice of r.) The probability that all these n calls return the right answer is at least $1 - n\delta$. So as long as $\delta \leq 1/(2n)$, the success probability of our procedure is at least $1/2$.

The requirement $\delta \leq 1/(2n)$ translates to $\epsilon \geq 1 - 1/(2n)$, meaning ϵ is tending to 1 as n tends to infinity. We would like to do better and find x even when ϵ is not only a constant, but perhaps even an inverse polynomial in n.

Here's a thought. Above, we were sloppy in upper bounding the failure probability of the algorithm $\mathrm{SC}^{\mathsf{B}_x}(z)$. The way we did it is to say that we wanted both b_1 and b_2 to be correct; all other cases we took to be failure. But actually, the output of the algorithm is also correct when both b_1 and b_2 are wrong, because we are working mod two. In other words, the bad case is not that at least one of the two is wrong, but *exactly* one of the two is wrong, and this might have a smaller probability of happening. Thus

$$\Pr\left[\, b_1 - b_2 \neq \langle x, z \rangle \,\right] \;=\; \Pr\left[\, z+r \in \mathsf{BdS} \text{ and } r \in \mathsf{GdS} \,\right] + \Pr\left[\, z+r \in \mathsf{GdS} \text{ and } r \in \mathsf{BdS} \,\right] \ .$$

However r and $z + r$ are not independently distributed, so the value of the terms above is unclear. It turns out that there can be a value of z such that both probabilities above equal $(1 - \epsilon)/2$, in

```
Algorithm STRONG-SC$^{\mathsf{B}_x}(z; r_1, \ldots, r_m; b_1, \ldots, b_m)$
    $sum \leftarrow 0$
    For $i = 1, \ldots, 2^m$ do
        $b[S_i] \leftarrow \sum_{j \in S_i} b_j$
        $c_i \leftarrow \mathsf{B}_x(z + R[S_i]) - b[S_i]$
        $sum \leftarrow sum + c_i$
    End For
    If $sum \geq 2^m/2$ then $b \leftarrow 1$ else $b \leftarrow 0$
    Return $b$
```

Figure 2: The STRONG-SC algorithm that attempts to compute $\langle x, z \rangle$ given a random sequence of n-bit strings $R = (r_1, \ldots, r_m)$ and auxiliary bits $b_1, \ldots, b_m$.

which case the sum is $1 - \epsilon = \delta$ just as before. (You can try to build this example as an exercise). So this idea doesn't help after all. We need a different algorithm.

4 The general case

If k is any integer we let $[k] = \{1, \ldots, k\}$.

We introduce a parameter m which will eventually be set to $c \lg(n)$ for some constant c to be specified. If $R = (r_1, \ldots, r_m)$ is a sequence of n-bit strings and $S \subseteq [m]$ then we let $R[S] = \sum_{j \in S} r_j$. The sum here is performed componentwise modulo two, so the result is an n-bit string. Let $S_1, \ldots, S_{2^m}$ be a listing of all subsets of $[m]$ in some canonical order.

The goal of the STRONG-SC$^{\mathsf{B}_x}$ algorithm of Figure 2 is the same as that of SC$^{\mathsf{B}_x}$, namely to compute $\langle x, z \rangle$ for a given input $z \in \{0,1\}^n$. However our new algorithm has additional inputs. It takes a sequence $R = (r_1, \ldots, r_m)$ of n-bit strings which will be selected at random. It also takes a sequence $b_1, \ldots, b_m$ of bits. For the moment assume that $b_j = \langle x, r_j \rangle$ for $j = 1, \ldots, m$. How we can find these bits is a question we will address later; for now, just assume we managed to guess the "right" values of the m inner products $\langle x, r_1 \rangle, \ldots, \langle x, r_m \rangle$. In the algorithm, *sum* is an integer counter and the "+" in "$sum + c_i$" is integer addition; all other operations are the usual mod two ones.

The idea behind the algorithm is the following. The linearity of the inner-product function tells us that for any $i = 1, \ldots, 2^m$ we have

$$\langle x, z + R[S_i] \rangle \quad = \quad \langle x, z \rangle + \sum_{j \in S_i} \langle x, r_j \rangle \ .$$

If $b_j = \langle x, r_j \rangle$ then the right-hand side is $\langle x, z \rangle + \sum_{j \in S_i} b_j$. Denoting the sum here by $b[S_i]$ we can solve as follows:

$$\langle x, z \rangle \quad = \quad \langle x, z + R[S_i] \rangle - b[S_i] \ .$$

We want to use this equation to determine $\langle x, z \rangle$. We will attempt to compute $\langle x, z + R[S_i] \rangle$ by calling B_x on input $z + R[S_i]$. We will argue that with high enough probability over the choice of

```
Algorithm $\text{Recover}^{\mathsf{B}_x,\mathsf{GD}_x}(1^n)$
    For $j = 1, \ldots, m$ do $r_j \overset{R}{\leftarrow} \{0,1\}^n$ End For
    For $i = 1, \ldots, 2^m$ do
        Let $b_1 \ldots b_m$ be the binary representation of $i - 1$
        For $k = 1, \ldots, n$ do
            $y^{(k)} \leftarrow \text{Strong-SC}^{\mathsf{B}_x}(e_k; r_1, \ldots, r_m; b_1, \ldots, b_m)$
        End For
        $y \leftarrow y^{(1)} \ldots y^{(n)}$
        If $\mathsf{GD}_x(y) = 1$ then $x' \leftarrow y$
    End For
    Return $x'$
```

Figure 3: The RECOVER algorithm that attempts to compute x.

the sequence R we have

$$\langle x, z \rangle \quad = \quad \mathsf{B}_x(z + R[S_i]) - b[S_i]$$

for a majority of the values of $i \in [2^m]$. Thus, taking a majority vote over the values of $\mathsf{B}_x(z + R[S_i]) - b[S_i]$ as $i = 1, \ldots, 2^m$ will yield a bit that with high probability equals $\langle x, z \rangle$.

Once we have an algorithm that with high enough probability determines $\langle x, z \rangle$ for a given z, we can compute x as before. Namely we would call this algorithm on $e_1, \ldots, e_n$ and thus retrieve x bit by bit.

There are several issues to be dealt with in taking this high-level picture into an actual algorithm to recover x. First, we must pin down what we mean by "high enough" probabilities in the above, and analyze the STRONG-SC algorithm to see that it accomplishes its task with such probabilities. Second we have the issue of the bits $b_1, \ldots, b_m$ that above we assumed magically to be the "right" ones.

Let's deal with the second issue first. It is in solving this that we make use of the second oracle GD_x which, recall, tells us whether we have a solution. So far we have not used this.

The full recovery algorithm is depicted in Figure 3. We begin by picking $r_1, \ldots, r_m$ at random. The key point is that $m = O(\lg n)$. So there are only polynomially many vectors $b_1, \ldots, b_m$ to consider. We simply try them all. For each choice of the vector $b_1, \ldots, b_m$ we run the STRONG-SC algorithm n times, on the inputs $e_1, \ldots, e_n$, to generate candidates for the bits of x. Each candidate x is tested using GD_x. Some choice of $b_1, \ldots, b_m$ is correct —meaning $b_j = \langle x, r_j \rangle$ for $j = 1, \ldots, m$— so in that iteration of the loop we find x.

Notice the crucial role of the testing oracle GD_x. Had that not been present, we would have 2^m candidates for x but no way to telling which of these is a good one.

The main claim for the analysis thus reduces to a claim about the STRONG-SC algorithm when it gets the right choice of the auxiliary bits. In that case we can upper bound the probability that it fails to compute $\langle x, z \rangle$ as shown in the next lemma. Note the algorithm itself is deterministic; the only random choice below is $R = (r_1, \ldots, r_m)$.

Lemma 1 Let $M = 2^m$. Then for any $z \in \{0,1\}^n$ we have

$$\Pr\left[\, \textsc{Strong-SC}^{\mathsf{B}_x}(z; r_1, \ldots, r_m; \langle x, r_1\rangle, \ldots, \langle x, r_m\rangle) \;\neq\; \langle x, z\rangle \;:\; r_1, \ldots, r_m \stackrel{R}{\leftarrow} \{0,1\}^n \,\right] \leq \frac{1}{M\epsilon^2} \,. \;\blacksquare$$

We will prove this lemma later. Given this we can easily estimate the failure probability of the Recover algorithm. The coin tosses here are those of the algorithm itself.

Lemma 2 Let $M = 2^m$. Then

$$\Pr\left[\, \textsc{Recover}^{\mathsf{B}_x, \mathsf{GD}_x}(1^n) \;\neq\; x \,\right] \;\leq\; \frac{n}{M\epsilon^2} \,. \;\blacksquare$$

Proof of Lemma 2: Due to the loop considering all possible values of $b_1, \ldots, b_m$ we need only consider the case where $b_j = \langle x, r_j\rangle$ for $j = 1, \ldots, m$. In that case the Recover algorithm invokes Strong-SC a total of n times, using n different values of z but always the same values of $r_1, \ldots, r_m$ and $b_1, \ldots, b_m$. The probability that any of these calls returns the wrong answer is at most the sum over $k = 1, \ldots, n$ of the probability that that the k-th call returns the wrong answer. But the probability of a wrong answer on any call is bounded as per Lemma 1. $\blacksquare$

Evaluating the complexity of the above procedure yields the following conclusion.

Theorem 3 Let m be a parameter and $M = 2^m$. Then there is an algorithm A which makes at most $q_B = nM$ calls to its B_x oracle, at most $q_E = M$ calls to its GD_x oracle, has time-complexity (execution time plus size of code) at most $t = O(nM^2)$ and success probability at least $1 - \delta$ where $\delta = n\epsilon^{-2}/M$. $\blacksquare$

To get success probability of 1/2 we would set $M = 2n\epsilon^{-2}$. In that case $m = \lg(M) = \lg(n) + 2\log(\epsilon^{-1}) + 1$. The running time of A is $O(n^3\epsilon^{-4})$ and $q_B = O(n^2\epsilon^{-2})$ and $q_E = O(n\epsilon^{-2})$.

What remains is to prove Lemma 1. That's the bulk of the work. We will first sketch the main ideas. Then we will stop and recall some probability theory, and use that to conclude the proof.

We will define a random variable X_i for $i \in [M]$ that takes the value 1 when the value of $\mathsf{B}_x(z + R[S_i]) - b[S_i]$ is correct, meaning equals $\langle x, z\rangle$. (Under the assumption that $b_1, \ldots, b_m$ are correct.) The random variables $X_1, \ldots, X_M$ are not independent. However, they satisfy a certain limited type of independence: they are pairwise independent. This means that having the value of one of them doesn't help predict the value of another, even though having the value of two of them might help to predict others. This pairwise independent property is enough to prove Lemma 1 using Chebyshev's inequality.

To do all this we need to step back and recall some probability theory.

Definition 4 Let $X_1, \ldots, X_M$: $S \to \mathbf{R}$ be real-valued functions on some sample space S. The latter is equipped with a probability distribution under which $X_1, \ldots, X_M$ are viewed as random variables. We say that $X_1, \ldots, X_M$ are *pairwise independent* if for every $i, j \in [M]$ with $i \neq j$ and every $a, b \in \mathbf{R}$ we have

$$\Pr\left[\, X_i = a \text{ and } X_j = b \,\right] \;=\; \Pr\left[\, X_i = a \,\right] \cdot \Pr\left[\, X_j = b \,\right] \,. \;\blacksquare$$

To bring this into context, here's how we set up the random variables for the proof of Lemma 1. Let S be the set of all m-element sequences with entries from $\{0,1\}^n$. Put a uniform distribution on S. (That corresponds to picking $r_1, \ldots, r_m$ at random.) Now for $i = 1, \ldots, M$ define $X_i \colon S \to \{0,1\}$ as follows, on any input $R = (r_1, \ldots, r_m) \in S$–

$$X_i(R) = \begin{cases} 1 & \text{if } \mathsf{B}_x(z + R[S_i]) - \sum_{j \in S_i} \langle x, r_j \rangle \ = \ \langle x, z \rangle \\ 0 & \text{otherwise.} \end{cases}$$

This can be simplified by noting that the equality is true exactly when $\mathsf{B}_x(z+R[S_i]) = \langle x, z+R[S_i] \rangle$, which in turn happens exactly when $z + R[S_i]$ falls in the good set of inputs. Thus

$$X_i(R) = \begin{cases} 1 & \text{if } z + R[S_i] \in \mathsf{GdS} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Our claim is that the random variables $X_1, \ldots, X_M$ are pairwise independent. Why? If $S_i \neq S_j$ then there is some string r_k that belongs to one but not the other. Now given that operations are modulo two, a sum involving r_k is unpredictable from a sum not involving r_k. So if we know that $z + R[S_i]$ is in GdS, we still do not know whether $z + R[S_j]$ is in GdS– given $z + R[S_i]$, the value of $z + R[S_j]$ is still uniformly distributed.

You should probably play around a bit to convince yourself of this claim that $X_1, \ldots, X_M$ are pairwise independent, but this is the main idea.

Now let's go back to the general probability theory. Recall that if Y is a random variable then its variance is $\mathbf{Var}[Y] = \mathbf{E}[(Y-\mu)^2] = \mathbf{E}[Y^2] - \mu^2$ where $\mu = \mathbf{E}[Y]$ is the expectation of Y.

Lemma 5 Let $X_1, \ldots, X_M \colon S \to \mathbf{R}$ be pairwise independent random variables. Then

$$\mathbf{Var}[X_1 + \cdots + X_M] = \mathbf{Var}[X_1] + \cdots + \mathbf{Var}[X_M] \ . \ \blacksquare$$

Proof of Lemma 5: Use the formula for the variance and the linearity of expectation to get

$$\begin{aligned}
\mathbf{Var}[X_1 + \cdots + X_M] &= \mathbf{E}\left[(X_1 + \cdots + X_M)^2\right] - \mathbf{E}[X_1 + \cdots + X_M]^2 \\
&= \mathbf{E}[(X_1 + \cdots + X_M)(X_1 + \cdots + X_m)] - (\mathbf{E}[X_1] + \cdots + \mathbf{E}[X_M])^2 \\
&= \mathbf{E}\left[\textstyle\sum_{i,j} X_i X_j\right] - \sum_{i,j} \mathbf{E}[X_i] \cdot \mathbf{E}[X_j] \\
&= \sum_{i,j} \mathbf{E}[X_i X_j] - \sum_{i,j} \mathbf{E}[X_i] \cdot \mathbf{E}[X_j] \\
&= \sum_i \mathbf{E}\left[X_i^2\right] + \sum_{i \neq j} \mathbf{E}[X_i X_j] - \sum_i \mathbf{E}[X_i]^2 - \sum_{i \neq j} \mathbf{E}[X_i] \cdot \mathbf{E}[X_j] \\
&= \sum_i \left(\mathbf{E}\left[X_i^2\right] - \mathbf{E}[X_i]^2\right) + \sum_{i \neq j} (\mathbf{E}[X_i X_j] - \mathbf{E}[X_i] \cdot \mathbf{E}[X_j]) \\
&= \sum_i \mathbf{Var}[X_i] + \sum_{i \neq j} (\mathbf{E}[X_i X_j] - \mathbf{E}[X_i] \cdot \mathbf{E}[X_j]) \ .
\end{aligned}$$

The pairwise independence means that $\mathbf{E}[X_i X_j] = \mathbf{E}[X_i] \cdot \mathbf{E}[X_j]$ whenever $i \neq j$. Thus the second sum above is zero, and we are done. $\blacksquare$

Lemma 6 Let $X_1, \ldots, X_M\colon\ S \to \mathbf{R}$ be pairwise independent random variables, let $X = X_1 + \cdots + X_M$, let $A > 0$ be a real number, and let $\mu = \mathbf{E}[X_1] + \cdots + \mathbf{E}[X_M]$. Then

$$\Pr[\,|X-\mu| > A\,] \;\le\; \frac{\mathbf{Var}[X_1] + \cdots + \mathbf{Var}[X_M]}{A^2} \,. \quad \blacksquare$$

Proof of Lemma 6: Chebyshev's inequality tells us that

$$\Pr[\,|X-\mu| > A\,] \;\le\; \frac{\mathbf{Var}[X]}{A^2} \,.$$

Now apply Lemma 5. $\blacksquare$

That's it. Now we use Lemma 6. Recall that in Equation (2) above we defined the random variables $X_1, \ldots, X_M\colon\ S \to \{0,1\}$ that we need for the proof of Lemma 1, and said that they were pairwise independent. Now observe that

$$\begin{aligned}
\mathbf{E}[X_i] &= 1 \cdot \Pr[\,X_i = 1\,] + 0 \cdot \Pr[\,X_i = 0\,] \\
&= \Pr[\,X_i = 1\,] \\
&= \Pr[\,z + R[S_i] \in \mathsf{GdS}\,] \\
&= \frac{1+\epsilon}{2} \,.
\end{aligned}$$

This is true because $R[S_i]$ is uniformly distributed in $\{0,1\}^n$. Now

$$\begin{aligned}
\mathbf{Var}[X_i] &= \mathbf{E}\left[X_i^2\right] - \mathbf{E}[X_i]^2 \\
&= \mathbf{E}[X_i] - \mathbf{E}[X_i]^2 \\
&= \mathbf{E}[X_i] \cdot (1 - \mathbf{E}[X_i]) \\
&= \frac{1+\epsilon}{2} \cdot \frac{1-\epsilon}{2} \\
&= \frac{1-\epsilon^2}{4} \,.
\end{aligned}$$

Let $X = X_1 + \cdots + X_M$ and $\mu = \mathbf{E}[X]$. Linearity of expectation tells us that $\mu = M(1+\epsilon)/2$. Then observe that the probability that we want to bound in Lemma 1 is exactly

$$\begin{aligned}
\Pr[\,X < M/2\,] &\le \Pr\left[\,|X-\mu| > \frac{M\epsilon}{2}\,\right] \\
&\le \frac{\mathbf{Var}[X_1] + \cdots + \mathbf{Var}[X_M]}{(M\epsilon/2)^2} \\
&= \frac{M(1-\epsilon^2)/4}{M^2\epsilon^2/4} \\
&\le \frac{1}{M\epsilon^2}
\end{aligned}$$

as desired. That concludes the proof of Lemma 1.

Acknowledgments

Thanks to Ramarathnam Venkatesan for pointers and comments.

References

[1] W. Alexi, B. Chor, O. Goldreich and C. Schnorr, "RSA and Rabin Functions: Certain Parts Are as Hard as the Whole," *SIAM J. on Computing*, Vol. 17, No. 2, 1988, pp. 194–209.

[2] M. Blum, M. Luby and R. Rubinfeld, "Self-testing/correcting with applications to numerical problems," *Journal of Computer and System Sciences*, Vol. 47, 1993, pp. 549–595.

[3] O. Goldreich, "Three XOR lemmas: An exposition," Manuscript available at `http://www.wisdom.weizmann.ac.il/users/oded/papers.html`. See Chapter 3.

[4] O. Goldreich, *Modern cryptography, probabilistic proofs and pseudorandomness*, Springer, 1999. See Appendix C.2.

[5] O. Goldreich and L. Levin, "A hard predicate for all one-way functions," STOC 1989.

[6] L. Levin, "Randomness and non-determinism," Manuscript available at `http://www.cs.bu.edu/fac/lnd/research/publ.html`.

[7] M. Luby, *Pseudorandomness and cryptographic applications*, Princeton Computer Science Notes, 1996.