Electronic Colloquium on Computational Complexity, Report No. 57 (2005)

# Hardness Amplification via Space-Efficient Direct Products

Venkatesan Guruswami[*]
Dept. of Computer Science & Engineering
University of Washington
Seattle, USA
venkat@cs.washington.edu

Valentine Kabanets[†]
School of Computing Science
Simon Fraser University
Vancouver, Canada
kabanets@cs.sfu.ca

May, 2005

**Abstract**

We prove a version of the derandomized Direct Product Lemma for deterministic space-bounded algorithms. Suppose a Boolean function $g : \{0,1\}^n \to \{0,1\}$ cannot be computed on more than $1-\delta$ fraction of inputs by any deterministic time $T$ and space $S$ algorithm, where $\delta \leqslant 1/t$ for some $t$. Then, for $t$-step walks $w = (v_1, \dots, v_t)$ in some explicit $d$-regular expander graph on $2^n$ vertices, the function $g'(w) \overset{\text{def}}{=} g(v_1) \dots g(v_t)$ cannot be computed on more than $1 - \Omega(t\delta)$ fraction of inputs by any deterministic time $\approx T/d^t - \mathsf{poly}(n)$ and space $\approx S - O(t)$. As an application, by iterating this construction, we get a deterministic linear-space "worst-case to constant average-case" hardness amplification reduction, as well as a family of logspace encodable/decodable error-correcting codes that can correct up to a constant fraction of errors. Logspace encodable/decodable codes (with linear-time encoding and decoding) were previously constructed by Spielman [Spi96]. Our codes have weaker parameters (encoding length is polynomial, rather than linear), but have a conceptually simpler construction. The proof of our Direct Product Lemma is inspired by Dinur's remarkable recent proof of the PCP theorem by gap amplification using expanders [Din05].

## 1 Introduction

### 1.1 Hardness amplification via Direct Products

Hardness amplification is, roughly, a procedure for converting a somewhat difficult computational problem into a much more difficult one. For example, one would like to convert a problem $A$ that is worst-case hard (i.e., cannot be computed within certain restricted computational model) into a new problem $B$ that is *average-case* hard (i.e., cannot be computed on a significant fraction of inputs).

The main motivation for hardness amplification comes from the desire to generate "pseudo-random" distributions on strings. Such distributions should be generated using very little true randomness, and yet *appear* random to any computationally bounded observer. The fundamental discovery of Blum, Micali, and Yao [BM84, Yao82] was that certain average-case hard problems

[*]Supported in part by NSF Career Award CCF-0343672.
[†]Supported in part by an NSERC Discovery grant

ISSN 1433-8092

(one-way functions) can be used to build pseudorandom generators. Later, Nisan and Wigderson [NW94] showed that Boolean functions of sufficiently high average-case circuit complexity can be used to derandomize (i.e., simulate efficiently deterministically) any probabilistic polynomial-time algorithm.

The construction of [NW94] requires an exponential-time computable Boolean function family $\{f_n : \{0,1\}^n \to \{0,1\}\}_{n>0}$ such that no Boolean circuit of size $s(n)$ can agree with $f_n$ on more than $1/2 + 1/s(n)$ fraction of inputs. The quality of derandomization depends on the lower bound $s(n)$ for the average-case complexity of $f_n$: the bigger the bound $s(n)$, the better the derandomization. For example, if $s(n) = 2^{\Omega(n)}$, then every probabilistic polynomial-time algorithm can be simulated in deterministic polynomial time.

Proving average-case circuit lower bounds (even for problems in deterministic exponential time) is a very difficult task. A natural question to ask is whether a Boolean function of high *worst-case* circuit complexity can be used for derandomization (the hope is that a worst-case circuit lower bound may be easier to prove). The answer turns out to be Yes. In fact, worst-case hard Boolean functions can be efficiently converted into average-case hard ones via an appropriate hardness amplification procedure.

The first such "worst-case to average-case" reduction was given by Babai, Fortnow, Nisan, and Wigderson [BFNW93]. They used algebraic error-correcting codes to go from a worst-case hard function $f$ to a weakly average-case hard function $g$. They further amplified the average-case hardness of $g$ via the following Direct Product construction. Given $g : \{0,1\}^n \to \{0,1\}$, define $g^k : (\{0,1\}^n)^k \to \{0,1\}^k$ as $g^k(x_1, \dots, x_k) = g(x_1) \dots g(x_k)$. Intuitively, computing $g$ on $k$ independent inputs $x_1, \dots, x_k$ should be significantly harder than computing $g$ on a single input. In particular, if $g$ cannot be computed by circuits of certain size $s$ on more than $1 - \delta$ fraction of inputs (i.e., $g$ is $\delta$-hard for circuit size $s$), then one would expect that $g^k$ should not be computable (by circuits of approximately the same size $s$) on more than $(1-\delta)^k$ fraction of inputs. The result establishing the correctness of this intuition is known as Yao's Direct Product Lemma [Yao82], and has a number of different proofs [Lev87, GNW95, Imp95, IW97].

## 1.2 Derandomized Direct Products and Error-Correcting Codes

Impagliazzo and Wigderson [Imp95, IW97] consider a "derandomized" version of the Direct Product lemma. Instead of evaluating a given $n$-variable Boolean function $g$ on $k$ *independent* inputs $x_1, \dots, x_k$, they generate the inputs using a certain deterministic function $F : \{0,1\}^r \to (\{0,1\}^n)^k$ such that the input size $r$ of $F$ is much smaller than the output size $kn$. They give several examples of the function $F$ for which the function $g'(y)$ defined as $g(F(y)_1) \dots g(F(y)_k)$, where $F(y)_i$ denote the $i$th $n$-bit string output by $F(y)$ for $y \in \{0,1\}^r$, has average-case hardness about the same as that of $g^k(x_1, \dots, x_k)$ for completely independent inputs $x_i$. In particular, Impagliazzo [Imp95] shows that if $g$ is $\delta$-hard (for certain size circuits) for $\delta < 1/O(n)$, then, for a pairwise independent $F : \{0,1\}^{2n} \to (\{0,1\}^n)^n$, the function $g'(y) = g(F(y)_1) \dots g(F(y)_n)$ is $\Omega(\delta n)$-hard (for slightly smaller circuits).

Trevisan [Tre03] observes that any Direct Product Lemma proved via "black-box" reductions can be interpreted as an error-correcting code mapping binary messages into codewords over a larger alphabet. Think of an $N = 2^n$-bit message *Msg* as a truth table of an $n$-variable Boolean function $g$. The encoding *Code* of this message will be the table of values of the direct-product function $g^k$. That is, the codeword *Code* is indexed by $k$-tuples of $n$-bit strings $(x_1, \dots, x_k)$, and the value of *Code* at position $(x_1, \dots, x_k)$ is the $k$-tuple $(g(x_1), \dots, g(x_k))$. The Direct Product Lemma says that if $g$ is $\delta$-hard, then $g^k$ is $\epsilon \approx 1 - (1-\delta)^k$-hard. In the language of codes, this means that given (oracle access to) a string $w$ over the alphabet $\Sigma = \{0,1\}^k$ such that $w$ and *Code* disagree in less than $\epsilon$ fraction of positions, we can construct an $N$-bit string $Msg'$ such that $Msg$ and $Msg'$ disagree in less than $\delta$ fraction of positions.

Note that the error-correcting code derived from a Direct Product Lemma maps $N$-bit messages to $N^k$-symbol codewords over the larger alphabet $\Sigma = \{0,1\}^k$. A derandomized Direct Product Lemma, using a function $F : \{0,1\}^r \to (\{0,1\}^n)^k$ as described above, yields an error-correcting code with encoding length $2^r$. For example, the pairwise-independent function $F$ from Impagliazzo's derandomized Direct Product Lemma would yield codes with encoding length $N^2$, which is a significant improvement over the length $N^k$.

The complexity of the reduction used to prove a Direct Product Lemma determines the complexity of the decoding procedure for the corresponding error-correcting code. In particular, if a reduction uses some non-uniformity (say, $m$ bits of advice), then the corresponding error-correcting code will be only *list-decodable* with the list size at most $2^m$. If one wants to get codes with $\epsilon$ being asymptotically close to 1, then list-decoding is indeed necessary. However, for a constant $\epsilon$, unique decoding is possible, and so, in principle, there must be a proof of this weaker Direct Product Lemma that uses only uniform reductions (i.e., no advice).

### 1.3 Derandomized Direct Products via uniform reductions

The derandomized Direct Product lemmas of [Imp95, IW97] are proved using nonuniform reductions. Using graph-based construction of error-correcting codes of [GI01], Trevisan [Tre03] proves a variant of a derandomized Direct Product lemma with a *uniform deterministic* reduction.

More precisely, for certain $k$-regular expander graphs $G_n$ on $2^n$ vertices (labeled by $n$-bit strings), Trevisan [Tre03] defines the function $F : \{0,1\}^n \to (\{0,1\}^n)^k$ as $F(y) = y_1, \dots, y_k$, where $y_i$s are the neighbors of the vertex $y$ in the graph $G_n$. He then argues that, for a Boolean function $g : \{0,1\}^n \to \{0,1\}$, if there is a deterministic algorithm running in time $t(n)$ that solves $g'(y) = g(F(y)_1) \dots g(F(y)_k)$ on $\Omega(1)$ fraction of inputs, then there is a deterministic algorithm running in time $O(t\mathsf{poly}(n,k))$ that solves $g$ on $1-\delta$ fraction of inputs, for $\delta = O(1/k)$. That is, if $g$ is $\delta$-hard with respect to deterministic time algorithms, then $g'$ is $\Omega(1)$-hard with respect to deterministic algorithms running in slightly less time. Note that the input size of $g'$ is $n$, which is the same as the input size of $g$.

The given non-Boolean function $g' : \{0,1\}^n \to \{0,1\}^k$ can be converted into a Boolean function $g''$ on $n + O(\log k)$ input variables that has almost the same $\Omega(1)$ hardness with respect to deterministic algorithms. The idea is to use some binary error-correcting code $\mathcal{C}$ mapping $k$-bit messages to $O(k)$-bit codewords, and define $g''(x,i)$ to be the $i$th bit of $\mathcal{C}(g'(x))$.

### 1.4 Our results

In this paper, we analyze a different derandomized Direct Product construction. Let $G_n$ be a $d$-regular expander graph on $2^n$ vertices, for some constant $d$. Denote by $[d]$ the set $\{1,2,\dots,d\}$. For any $t$ and any given $n$-variable Boolean function $g$, we define $g'$ to be the value of $g$ along a $t$-step walk in $G_n$. That is, we define $g' : \{0,1\}^n \times [d]^t \to \{0,1\}^{t+1}$ as $g'(x, i_1, \dots, i_t) = g(x_0)g(x_1)\dots g(x_t)$, where $x_0 = x$, and each $x_j$ (for $1 \leqslant j \leqslant t$) is the $i_j$th neighbor of $x_{j-1}$ in the graph $G_n$. We show that if $g$ is $\delta$-hard to compute by deterministic uniform algorithms running in time $T$ and space $S$ for $\delta < 1/t$, then $g'$ is $\Omega(t\delta)$-hard with respect to deterministic algorithms running in time $\approx T/d^t$ and space $\approx S - O(t)$. Our proof of this Direct Product lemma is inspired by Dinur's remarkable recent proof of the PCP theorem by gap amplification using expanders [Din05].

Note that if $g$ is $\delta$-hard, then we expect $g^t(x_1, \ldots, x_t) = g(x_1) \ldots g(x_t)$ (on $t$ independent inputs) to be $\delta' = 1-(1-\delta)^t$-hard. For $\delta \ll 1/t$, we have $\delta' \approx t\delta$, and so our derandomized Direct Product construction described above achieves asymptotically correct hardness amplification.

Combining the function $g'$ with any linear error-correcting code $\mathcal{C}$ (with constant relative distance) mapping $(t+1)$-bit messages into $O(t)$-bit codewords, we can get from $g'$ a Boolean function on $n + O(t)$ variables that also has hardness $\Omega(t\delta)$. Applying these two steps (our expander-walk Direct Product followed by an encoding using the error-correcting code $\mathcal{C}$) to a given $\delta$-hard $n$-variable Boolean function $g$ for $\log 1/\delta$ iterations, we can obtain a new Boolean function $g''$ on $n + O(t \log 1/\delta)$ variables that is $\Omega(1)$-hard. If $g$ is $\delta$-hard for deterministic time $T$ and space $S$, then $g''$ is $\Omega(1)$-hard for deterministic time $\approx T\mathsf{poly}(\delta)$ and space $\approx S - O(\log 1/\delta)$.

In terms of running time, this iterated Direct Product construction matches the parameters of Trevisan's Direct Product construction described earlier. Both constructions are proved with uniform deterministic reductions. The main difference seems to be in the usage of space. Our reduction uses at most $O(n + \log 1/\delta)$ space, which is at most $O(n)$ even for $\delta = 2^{-n}$. Thus we get a deterministic uniform "worst-case to constant average-case" reduction computable in linear space. The space usage in Trevisan's construction is determined by the space complexity of encoding/decoding of the "inner" error correcting code $\mathcal{C}$ from $k$ to $O(k)$ bits, for $k = O(1/\delta)$. A simple deterministically encodable/decodable code would use space $\Omega(k) = \Omega(1/\delta)$.

We also show that constant-degree expanders which have expansion better than degree/2 can be used to obtain a simple space-efficient hardness amplification. However, it is not known how to construct such expanders explicitly.

**Related work.** Our deterministic linear-space hardness amplification result is not new. A deterministic linear-space "worst-case to constant average-case" reduction can be also achieved by using Spielman's expander-based error-correcting codes [Spi96]. His codes have encoding/decoding algorithms of space complexity $O(\log N)$ for messages of length $N$, which translates into $O(n)$-space reductions for $n$-variable Boolean functions.

In light of the connection between Direct Product Lemmas and error-correcting codes explained earlier in the introduction, our iterated Direct Product construction also yields a deterministic logspace (in fact, uniform $\mathsf{NC}^1$) encodable/decodable error-correcting code that corrects a constant fraction of errors. Spielman's $\mathsf{NC}^1$ encodable/decodable codes also correct a constant fraction of errors, but they have much better other parameters. In particular, Spielman's encoding/decoding is in *linear* time, and so the length of the encoded message is linear in the size of the original message. In contrast, our encoding time and the length of the encoding is only polynomial in the size of the original message. We believe, however, that our codes have a conceptually simpler construction, which closely follows the "Direct Product Lemma" approach.

**Remainder of the paper.** We give the necessary definitions in Section 2. In Section 3, we state and analyze our Direct Product Lemma. Applications of our Direct Product Lemma to linear-space hardness amplification and logspace encodable/decodable codes are given in Section 4. Section 5 proves a simpler version of the Direct Product Lemma, under the assumption that expanders with expansion better than degree/2 can be efficiently constructed.

## 2 Preliminaries

### 2.1 Worst-case and average-case hardness

Given a bound $b$ on a computational resource *resource* (*resource* can be, e.g., deterministic time, space, circuit size, or some combination of such resources), we say that a function $f : A \to B$ (for some sets $A$ and $B$) is *worst-case hard for b-bounded resource* if every algorithm using at most $b$ amount of *resource* disagrees with the function $f$ on at least one input $x \in A$.

For $0 \leqslant \delta \leqslant 1$ and a bound $b$ on *resource*, a function $f : A \to B$ is called *average-case $\delta$-hard (or, simply, $\delta$-hard) for b-bounded resource* if every algorithm using at most $b$ amount of *resource* disagrees with the function $f$ on at least a fraction $\delta$ of inputs from $A$. Observe that for $\delta = 1/|A|$, the notion of $\delta$-hardness coincides with that of worst-case hardness.

### 2.2 Expanders

Let $G = (V, E)$ be any $d$-regular undirected graph on $n$ vertices. Let $A = \{a_{i,j}\}_{i,j=1}^{n}$ be the normalized adjacency matrix of $G$, i.e., $a_{i,j} = \frac{1}{d}*$(the number of edges between $i$ and $j$). For a constant $\lambda < 1$, the graph $G = (V, E)$ is called a $\lambda$-expander if the second largest (in the absolute value) eigenvalue of the matrix $A$ is at most $\lambda$.

Another (essentially equivalent) definition of expanders is the following. A $d$-regular graph $G = (V, E)$ is an $(\alpha, \beta)$-expander if for every subset $W \subseteq V$ with $|W| \leqslant \alpha|V|$,

$$\Big|\{v \in V \mid \exists w \in W \text{ such that } (v, w) \in E\}\Big| \geqslant \beta|W| \ .$$

The well-known basic property of expander graphs is fast mixing. For any vertex $v$ of the graph, a random walk from $v$ of $t$ steps will end up at a vertex $w$ that is very close to being uniformly distributed among all vertices of $G$; the deviation from the uniform distribution can be bounded by $\lambda^t$. Another basic property of expanders, which we will use in the analysis of our Direct Product Lemma, is the following lemma; a variant of this lemma (for edge sets rather than vertex sets) is proved in [Din05, Lemma 5.4].

**Lemma 1.** *Let $G = (V, E)$ be any $d$-regular $\lambda$-expander, and let $S \subset V$ be any set. For any value $t$, let $W_i$, for $i \in [0..t]$, be the set of all $t$-step walks in $G$ that pass through a vertex from $S$ in step $i$. Then, for each $i \in [0..t]$, a random walk from the set $W_i$ is expected to contain at most $2t\frac{|S|}{|V|} + O(1)$ vertices from the set $S$.*

We will need an infinite family of $d$-regular $\lambda$-expanders $\{G_n = (V_n, E_n)\}_{n=1}^{\infty}$, where $G_n$ is a graph on $2^n$ vertices; we assume that the vertices of $G_n$ are identified with $n$-bit strings. We need that such a family of graphs be *efficiently constructible* in the sense that given the label of a vertex $v \in V_n$ and a number $i \in [d]$, the $i$'th neighbor of $v$ in $G_n$ can be computed efficiently by a deterministic polynomial-time and linear-space algorithm. We will spell out the exact constructibility requirement in Section 3.1.

### 2.3 Space complexity

We review definitions concerning space complexity, since for our main Direct Product Lemma, it will be important to measure the space complexity of the algorithms quite carefully.

**Definition 2 (Standard Space Complexity).** An algorithm is said to compute a function $f$ in space $S$ if given as input $x$ on a *read-only* input tape, it uses a work tape of $S$ cells and halts with $f(x)$ on the work tape. Such an algorithm is said to have *space complexity $S$*.

**Definition 3 (Total Space Complexity).** An algorithm $A$ is said to compute a function $f$ with domain $\{0,1\}^n$ in *total space* $S$ if on an $n$-bit input $x$,

1. $A$ has read/write access to the input tape,
2. in addition to the $n$ input tape cells, $A$ is allowed another $S-n$ tape cells, and
3. at the end of its computation, the tape contains $f(x)$.

Such an algorithm is then said to have *total space complexity* $S$.

**Definition 4 (Input-Preserving Space Complexity).** An algorithm $A$ is said to compute a function $f$ with domain $\{0,1\}^n$ in *input-preserving space* $S$ if on an $n$-bit input $x$,

1. $A$ has read/write access to the input tape,
2. in addition to the $n$ input tape cells, $A$ is allowed another $S-n$ tape cells, and
3. at the end of its computation, the tape contains $x; f(x)$.

That is, we allow the algorithm to write on the input portion of the tape, provided it is restored to its original content at the end of the computation. Such an algorithm is then said to have *input-preserving space complexity* $S$. (Note that the input-preserving space complexity of a function $f(x)$ is the same as the total space complexity of the function $f'(x) \overset{\text{def}}{=} x; f(x)$.)

The following simple observation lets us pass between these models of space complexity with a linear additive difference.

**Fact 5.** *If there is an algorithm $A$ with space complexity $S$ to compute a function with domain $\{0,1\}^n$, then there is an algorithm $A'$ with input-preserving (total) space complexity $S+n$ to compute $f$. Conversely, if there is an algorithm $B'$ with input-preserving (total) space complexity $S'$ to compute $f$, then there is an algorithm $B$ with space complexity $S'$ to compute $f$.*

We will use the input-preserving space complexity to analyze the efficacy of our Direct Product Lemma and its iterative application to amplify hardness. However, by Fact 5 above, our end result can be stated in terms of the standard space complexity of Definition 2.

## 3 A New Direct Product Lemma

### 3.1 Construction

We need the following two ingredients:

- Let $G=(V,E)$ be any efficiently constructible $d$-regular expander on $|V|=2^n$ vertices which are identified with $n$-bit strings (here $d$ is an absolute constant $d$, and so we will typically hide factors depending on $d$ in the $O$-notation). By efficient constructibility, we mean the following. There is an algorithm running in time $T_{expander}=\mathsf{poly}(n)$ and total space $S_{expander}=O(n)$, that given as input an $n$-bit string $x$ and an index $i\in[d]$, outputs the pair $N_G(x,i)\overset{def}{=}(y,j)$, where $y\in\{0,1\}^n$ is the $i$'th neighbor in $G$ of $x$, and $j\in[d]$ is such that $x$ is the $j$'th neighbor of $y$. We can obtain such expanders from [RVW02].[1]

[1]Normally the space complexity of expander constructions is measured in the sense of Definition 2. However, by fact 5, for $O(n)$ space, we can pass freely to the total space complexity model of Definition 3.

- Let $\mathcal{C}$ be any polynomial-time and linear-space encodable (via *Enc*) and decodable (via *Dec*) linear binary error-correcting code with constant rate $1/c$ and constant relative distance $\rho$.

Our construction proceeds in two steps.

**Step 1:** Let $f : \{0,1\}^n \to \{0,1\}$ be any given Boolean function. For any $t \in \mathbb{N}$, define a new, non-Boolean function $g : \{0,1\}^n \times [d]^t \to \{0,1\}^{t+1}$ as follows:

$$g(v, i_1, \dots, i_t) = f(v)f(v_1)\dots f(v_t),$$

where for each $1 \leqslant j \leqslant t$, $v_j$ is the $i_j$th neighbor of vertex $v_{j-1}$ in the expander graph $G$ (we identify $v$ with $v_0$); recall that the vertices of $G$ are labeled by $n$-bit strings.

**Step 2:** Finally, define a Boolean function $h : \{0,1\}^n \times [d]^t \times [c(t+1)] \to \{0,1\}$ as

$$h(v, i_1, \dots, i_t, j) = \mathit{Enc}(g(v, i_1, \dots, i_t))_j,$$

where $\mathit{Enc}(y)_j$ denotes the the $j$th bit in the encoding of the string $y$ using the binary error-correcting code $\mathcal{C}$.

**Complexity of the encoding:** Suppose that the $n$-variable Boolean function $f$ is computable in deterministic time $T$ and input-preserving space $S$. Then the non-Boolean function $g$ obtained from $f$ in Step 1 of the construction above will be computable in deterministic time $T_g = O(t(T + T_{expander})) = O(t(T + \mathsf{poly}(n)))$ and input-preserving space at most $S_g = \max\{S, S_{expander}\} + O(t)$. The claim about time complexity is obvious. For the space complexity, to compute $g(v, i_1, \dots, i_t)$, we first compute $f(v)$ using input-preserving space $S$. We then re-use this space to compute $N_G(v, i_1) = (v_1, j_1)$ in total space $S_{expander}$. We remember $i_1, j_1$ (these take only $O(1)$ space) separately, but replace $v$ by $v_1$, and compute $f(v_1)$ in input-preserving space $S$. We next likewise compute $N_G(v_1, i_2) = (v_2, j_2)$, and replace $v_1$ by $v_2$, compute $f(v_2)$, and so on. In the end, we would have computed $f(v)f(v_1)\dots f(v_t)$ in total space $\max\{S, S_{expander}\} + O(t)$. However, we need to restore the original input $v, i_1, i_2, \dots, i_t$. For this, we use the stored "back-indices" $j_t, j_{t-1}, \dots, j_1$ to walk back from $v_t$ to $v$ in a manner identical to the forward walk.

The Boolean function $h$ obtained from $g$ in Step 2 will be computable in time $T_g + \mathsf{poly}(t)$ and input-preserving space $S_g + O(t)$. Note that, assuming $S \geqslant S_{expander}$, the input-preserving space complexity of $h$ is at most an additive constant term $O(t)$ bigger than that of $f$.

## 3.2 Analysis

We will show that the "direct product construction" described above increases the hardness of a Boolean function $f$ by a multiplicative factor $\Omega(t)$.

**Lemma 6 (Direct Product Lemma).** *Suppose an $n$-variable Boolean function $f$ has hardness $\delta \leqslant 1/t$ for deterministic time $T$ and input-preserving space $S \geqslant S_{expander} + \Omega(t)$. Let $h$ be the Boolean function obtained from $f$ using the direct product construction described above. Then $h$ has hardness $\Omega(t\delta)$ for deterministic time $T' = \frac{T}{O(t^2 d^t)} - \mathsf{poly}(n)$ and input-preserving space $S' = S - O(t)$.*

The proof of the Direct Product Lemma above will consist of two parts. First we argue that the non-Boolean function $g$ (obtained from $f$ by evaluating $f$ along $t$-step walks in the expander $G$) will have hardness $\Omega(t)$-factor larger than the hardness of $f$. Then we argue that turning the function $g$ into the Boolean function $h$ via encoding the outputs of $g$ by a "good" error-correcting code will reduce its hardness by only a constant factor independent of $t$ (but dependent on the relative distance $\rho$ of the code).

**Lemma 7.** *Suppose an $n$-variable Boolean function $f$ has hardness $\delta \leqslant 1/t$ for deterministic time $T$ and input-preserving space $S \geqslant S_{expander} + \Omega(t)$. Let $g$ be the non-Boolean function obtained from $f$ using the first step of the direct product construction described above. Then $g$ has hardness $\delta' = \Omega(t\delta)$ for deterministic time $T' = \frac{T}{O(td^t)} - t\mathsf{poly}(n)$ and input-preserving space $S' = S - O(t)$.*

*Proof.* Let $C'$ be a deterministic algorithm using time $T'$ and input-preserving space $S'$ that computes $g$ correctly on $1 - \delta'$ fraction of inputs, for the least possible $\delta'$ that can be achieved by algorithms with these time/space bounds. We will define a new deterministic algorithm $C$ using time at most $T$ and input-preserving space $S$, and argue that $\delta'$ (the fraction of inputs computed incorrectly by $C'$) is at least $\Omega(t)$ times larger than the fraction of inputs computed incorrectly by $C$. Since the latter fraction must be at least $\delta$ (as $f$ is assumed $\delta$-hard for time $T$ and input-preserving space $S$), we conclude that $\delta' \geqslant \Omega(t\delta)$.

We will compute $f$ by an algorithm $C$ defined as follows. On input $x \in \{0,1\}^n$, for each $i \in [0..t]$, record the majority value $b_i$ taken over all values $C'(w)_i$, where $w$ is a $t$-step walk in the graph $G$ that passes through $x$ at step $i$ and $C'(w)_i$ is the $i$th bit in the $(t+1)$-tuple output by the circuit $C'$ on input $w$ (here we assumed that the $t+1$ values output by $C$ are indexed by $0, 1, \ldots, t$). Output the majority over all the values $b_i$, for $0 \leqslant i \leqslant t$. A more formal description of the algorithm is given in the table Algorithm 1.

```
INPUT: $x \in \{0,1\}^n$.
GOAL: Compute $f(x)$.

$count_1 = 0$
for each $i = 0..t$
    $count_2 = 0$
    for each $t$-tuple $(k_1, k_2, \ldots, k_t) \in [d]^t$
        Compute the vertex $y$ reached from $x$ in $i$ steps by taking edges labeled $k_1, k_2, \ldots, k_i$,
            together with the "back-labels" $\ell_1, \ell_2, \ldots, \ell_i$ needed to get back from $y$ to $x$.
        $count_2 = count_2 + C'(y, \ell_1, \ell_2, \ldots, \ell_i, k_{i+1}, \ldots, k_t)_i$
        Restore $x$ by walking from $y$ for $i$ steps using edge-labels $\ell_1, \ell_2, \ldots, \ell_i$.
    end for
    if $count_2 \geqslant d^t/2$ then $count_1 = count_1 + 1$ end if
end for
if $count_1 \geqslant t/2$ then RETURN 1 else RETURN 0
end Algorithm
```

**Algorithm 1:** Algorithm $C$

It is straightforward to verify that the algorithm $C$ can be implemented in deterministic time $O(td^t(T' + t\mathsf{poly}(n)))$. By choosing $T'$ as in the statement of the lemma, we can ensure that the running time of $C$ is at most $T$. It is also easy to argue that the input-preserving space complexity $S$ of algorithm $C$ is at most $\max\{S_{expander}, S'\} + O(t)$ (the argument goes along the lines of the one we used to argue about the complexity of the encoding at the end of Section 3.1). Hence by choosing $S' = S - O(t) \geqslant S_{expander}$ we get the input-preserving space complexity of $C$ at most $S$.

We now analyze how many mistakes the algorithm $C$ makes in computing $f$. Define the set $Bad = \{x \in \{0,1\}^n \mid C(x) \neq f(x)\}$. Pick a subset $B \subseteq Bad$ such that $|B|/|V| = \min\{|Bad|/|V|, 1/t\}$. By definition, if $x \in Bad$, then for each of at least $1/2$ values of $i \in [0..t]$, the algorithm $C'$ is wrong on at least half of all $t$-step walks that pass through $x$ in step $i$. Define a 0-1 matrix $M$ with $|B|$ rows and $t+1$ columns such that, for $x \in B$ and $i \in [0..t]$, $M(x,i) = 0$ iff $C'$ is wrong on at least half of all $t$-step walks that pass through $x$ in step $i$. Then the fraction of 0s in the matrix $M$ is at least $1/2$. By averaging, we conclude that there exists a subset $I \subseteq [0..t]$ of size at least $t/4$ such that, for each $i \in I$, the $i$th column of $M$ contains at least $1/4$ fraction of 0s. This means that for each $i \in I$, the algorithm $C'$ is wrong on at least $\frac{|B|}{4}\frac{d^t}{2} = \frac{1}{8}|B|d^t$ of all $|B|d^t$ walks of length $t$ that pass through the set $B$ at step $i$.

For $x \in B$ and $i \in [0..t]$, let us denote by $W_{i,x}$ the set of all $t$-step walks that pass through $x$ in step $i$; observe that $|W_{i,x}| = d^t$. We define $W_i = \cup_{x \in B} W_{i,x}$. Since $W_{i,x}$ and $W_{i,y}$ are disjoint for $x \neq y$, we get $|W_i| = |B|d^t$. Also, for $x \in B$ and $i \in [0..t]$, denote by $W^*_{i,x}$ the set of all $t$-step walks $w \in W_{i,x}$ such that $C'(w) \neq g(w)$. Define $W^*_i = \cup_{x \in B} W^*_{i,x}$, and $W^* = \cup_{i=0}^{t} W^*_i$. Using this notation, we have that $|W^*_i| \geqslant \frac{1}{8}|W_i|$ for each $i \in I$.

For each $i \in [0..t]$, let $H_i \subseteq W_i$ be the set of all walks $w \in W_i$ that contain more than $m$ elements from $B$. Using the properties of the expander $G$, we can choose $m$ to be a sufficiently large constant (independent of $t$) such that, for all $i$, $|H_i| < \frac{1}{16}|W_i|$.

Indeed, by Lemma 1 above, for every $i$, a random walk $w \in W_i$ is expected to contain at most $2t|B|/|V| + O(1)$ vertices from $B$. Since, by our choice of parameters, $|B|/|V| \leqslant 1/t$, a random $w \in W_i$ contains on average at most $b = O(1)$ vertices from $B$. By Markov's inequality, the probability that a random $w \in W_i$ contains more than $m = 16b$ vertices from $B$ is at most $1/16$.

Thus we have

$$\sum_{i \in I} |W^*_i \setminus H_i| = \sum_{i \in I} (|W^*_i| - |H_i|) \geqslant |I|(\frac{1}{8} - \frac{1}{16})|W_i| \geqslant \frac{t}{64}|B|d^t. \tag{1}$$

On the other hand, we have

$$\sum_{i \in I} |W^*_i \setminus H_i| \leqslant m|W^* \setminus (\cup_{i=0}^{t} H_i)| \leqslant m|W^*|. \tag{2}$$

Combining Eqs. (1) and (2), we get

$$|W^*| \geqslant \frac{t}{64m}|B|d^t.$$

Dividing both sides by the number $|V|d^t$ of all possible $t$-step walks in $G$ (which is the number of all possible inputs to the algorithm $C'$), we get that that $C'$ makes mistakes on at least $\frac{t}{64m}|B|/|V|$ fraction of inputs. Note that $|B|/|V| \geqslant \delta$ since $f$ is assumed to be $\delta$-hard for time $T$ and input-preserving space $S$. It follows that the function $g$ is $\Omega(t\delta)$-hard for time $T'$ and input-preserving space $S'$. ☐

The analysis of the second step of our Direct Product construction uses the standard approach of "code concatenation". We include a proof for the sake of completeness.

**Lemma 8.** *Let $A = \{0,1\}^n \times [d]^t$. Suppose that a function $g : A \to \{0,1\}^{t+1}$ is $\delta$-hard for deterministic time $T$ and input-preserving space $S$. Let $h : A \times [c \cdot (t+1)] \to \{0,1\}$ be the Boolean function obtained from $g$ as described in Step 2 of the Direct Product construction above (using the error-correcting code with relative distance $\rho$ and rate $1/c$). Then the function $h$ is $\delta \cdot \rho/2$-hard for deterministic time $T' = (T - \mathsf{poly}(t))/O(t)$ and input-preserving space $S' = S - O(t)$.*

*Proof.* Let $C'$ be an algorithm running in deterministic time $T'$ and input-preserving space $S'$ that computes $h$ on $1 - \delta'$ fraction of inputs, for the smallest possible $\delta'$ achievable by deterministic algorithms with such time/space bounds. Define an algorithm $C$ computing $g$ as follows: On input

$a \in A$, compute $C'(a,i)$ for all $i \in [c\cdot(t+1)]$, apply the decoder function $Dec$ of our error-correcting code to the obtained $c\cdot(t+1)$-bit string, and output the resulting $(t+1)$-bit string. Clearly, the running time of $C$ is at most $c(t+1)T' + \mathsf{poly}(t)$, where the $\mathsf{poly}(t)$ term accounts for the complexity of the decoding function $Dec$. The input-preserving space complexity of $C$ is at most $S' + O(t)$.

Consider the set $Bad = \{a \in A \mid C(a) \neq g(a)\}$. For each $a \in Bad$, the string $C'(a,1)\dots C'(a,c\cdot(t+1))$ must be at least $\rho/2$ far in relative Hamming distance from the correct encoding $Enc(g(a))$ of $g(a)$. Thus the number of inputs computed incorrectly by $C'$ is at least $|Bad|\frac{\rho}{2}c\cdot(t+1)$. Dividing this number by the total number $|A|c\cdot(t+1)$ of inputs to $C'$, we get that $C'$ is incorrect on $\delta' \geqslant \frac{\rho}{2}\frac{|Bad|}{|A|}$ fraction of inputs. Since $g$ is assumed $\delta$-hard for time $T$ and input-preserving space $S$, we get that $|Bad|/|A| \geqslant \delta$. It follows that $\delta' \geqslant \frac{\rho}{2}\delta = \Omega(\delta)$. ☐

*Proof of Lemma 6.* The proof follows by combining Lemmas 7 and 8. ☐

## 3.3 Iteration

Our Direct Product Lemma (Lemma 6) can be applied repeatedly to increase the hardness of a given Boolean function at an exponential rate, as long as the current hardness is less than some universal constant. In particular, as shown in the corollary below, we can turn a $\delta$-hard Boolean function into a $\Omega(1)$-hard Boolean function. Note that we state this result in terms of the usual space complexity, and not the input-preserving space complexity that we used to analyze a single Direct Product.

**Corollary 9.** *Let $f$ be an $n$-variable Boolean function that is $\delta$-hard for deterministic time $T$ and space $S \geqslant \Omega(n+\log\frac{1}{\delta})$. Then there is a Boolean function $f'$ on $n+O(\log\frac{1}{\delta})$ variables such that $f'$ is $\Omega(1)$-hard for deterministic time $T' = T\mathsf{poly}(\delta) - \mathsf{poly}(n)$ and space $S' = S - n - O(\log\frac{1}{\delta})$. Moreover, if $f$ is computable in time $\tilde{T}$ and space $\tilde{S}$, then $f'$ is computable in time $(\tilde{T} + \mathsf{poly}(n))/\mathsf{poly}(\delta)$ and space $\tilde{S} + O(n+\log\frac{1}{\delta})$.*

*Proof.* Pick a constant $t$ large enough so that the $\Omega(t)$ factor in the statement of Lemma 6 is at least 2. With this choice of $t$, each application of our Direct Product construction will double the hardness of the initial Boolean function.

By Fact 5, such an $f$ is $\delta$-hard for deterministic time $T$ and input-preserving space $S$. Let $f'$ be a Boolean function obtained from $f$ by repeated application of the Direct Product construction for $\log\frac{1}{\delta}$ steps (using an expander with $S_{expander} = O(n)$). Then it is straightforward to check that $f'$ is a $n + O(t\log\frac{1}{\delta})$-variable Boolean function of $\Omega(1)$-hardness for deterministic time $T' = T\delta^{O(t)} - \mathsf{poly}(n)$ and input-preserving space $S'' = S - O(\log\frac{1}{\delta})$. Referring to Fact 5 again, $f'$ is $\Omega(1)$-hard for deterministic time $T$ and space $S' = S'' - n = S - n - O(\log\frac{1}{\delta})$.

The time and space upper bounds for $f'$ follow easily from the complexity analysis of the Direct Product construction (and using Fact 5 to convert from space to input-preserving space and back). ☐

**Remark 10.** The constant average-case hardness in Corollary 9 above can be boosted to any constant less than 1/4 by one additional amplification with a suitable expander, as in [GI01, Tre03] (specifically, see Theorem 7 in [Tre03]).

**Remark 11.** We want to point out that Spielman's logspace encodable/decodable error-correcting codes [Spi96] can also be used for "worst-case to constant average-case" hardness amplification via deterministic linear-space reductions. So Corollary 9 is implicit in [Spi96].

## 4 Applications

### 4.1 Hardness amplification via deterministic space-efficient reductions

The iterated Direct Product construction of Corollary 9 gives us a way to convert worst-case hard Boolean functions into constant-average-case hard ones, with space-efficient deterministic reductions. The following theorems are immediate consequences of Corollary 9 and Remark 10. Below we use standard notation for the complexity classes $\mathsf{E} = \mathsf{DTIME}(2^{O(n)})$ and $\mathsf{LINSPACE} = \mathsf{SPACE}(O(n))$.

**Theorem 12.** *Let $\alpha < 1/4$ be an arbitrary constant. If there is a language $L \in \mathsf{E} \setminus \mathsf{LINSPACE}$, then there is a language $L' \in \mathsf{E}$ that is $\alpha$-hard for $\mathsf{LINSPACE}$.*

**Theorem 13.** *Let $\alpha < 1/4$ be an arbitrary constant. For every $c > 0$, there is a $c' > 0$ such that the following holds. If there is a language $L \in \mathsf{LINSPACE}$ that cannot be computed by any deterministic algorithm running in linear space and, simultaneously, time $2^{c'n}$, then there is a language $L' \in \mathsf{LINSPACE}$ that is $\alpha$-hard for any deterministic algorithm running in linear space and, simultaneously, time $2^{cn}$.*

### 4.2 Logspace encodable/decodable error-correcting codes

As mentioned in the introduction, every Direct Product Lemma gives rise to error-correcting codes with encoding/decoding complexity determined by the complexity of the reductions used in the proof of the Direct Product Lemma. In our case, we get error-correcting codes with polynomial rate that have deterministic logspace encoding/decoding complexity, and can correct up to a constant fraction of errors. Thus we get an alternative construction (with much weaker parameters) to Spielman's logspace encodable/decodable codes [Spi96].

**Theorem 14.** *There is an explicit code $\mathcal{C}$ mapping $n$-bit messages to $\mathsf{poly}(n)$-bit codewords such that*

1. *$\mathcal{C}$ can correct a constant fraction of errors,*
2. *both encoding and decoding can be implemented in deterministic logspace (in fact, uniform $\mathsf{NC}^1$).*

**Remark 15.** We are not aware of any logspace encodable/decodable asymptotically good codes other than Spielman's construction [Spi96], and the recent improvements to its error-correction performance [GI01, GI02]. Allowing $\mathsf{NC}^2$ complexity seems to give several other choices of error-correcting codes.

## 5 A simple graph based amplification

Here we observe that the existence of efficiently constructible $d$-regular expanders with expansion factor better than $d/2$ would give us another deterministic linear-space hardness amplification.

Recall Trevisan's derandomized Direct Product construction.

**Definition 16.** Given a $d$-regular graph $G$ on $2^n$ vertices, where each vertex is identified with an $n$-bit string, and a Boolean function $f : \{0,1\}^n \to \{0,1\}$, we define a function $g = G(f) : \{0,1\}^n \to \{0,1\}^d$ as follows. For $x \in \{0,1\}^n$, let $N_1(x), N_2(x), \ldots, N_d(x)$ denote the $d$ neighbors of $x$ in $G$ (as per some fixed ordering). Then $g(x) \overset{def}{=} f(N_1(x))f(N_2(x))\ldots f(N_d(x))$.

We note that a similar definition has been used in the construction of error-correcting codes in several works beginning with [ABN$^+$92] and more recently in [GI01, GI03].

**Lemma 17.** *Let $G = (\{0,1\}^n, E)$ be an efficiently (in total space $S_{expander}$) constructible $d$-regular $(\delta, d/2 + \gamma_d)$-expander for some $\gamma_d > 0$. Let $f : \{0,1\}^n \to \{0,1\}$ be $\delta$-hard for deterministic time $T$ and input-preserving space $S \geqslant S_{expander} + \Omega(d)$. Then the function $g = G(f)$ from Definition 16 is $\gamma_d\delta$-hard for deterministic time $T' = \frac{T}{d} - \mathsf{poly}(n)$ and input-preserving space $S - O(d)$.*

*Proof.* Let $C'$ be a deterministic algorithm running in time at most $T'$ and input-preserving space $S'$ that computes $g$ correctly on a fraction $1 - \delta'$ of the inputs, for the least possible $\delta'$ that can be achieved by algorithms within these time/space bounds. Using $C'$, we will define a deterministic algorithm $C$ running in time at most $T$ and input-preserving space $S$, and argue that the fraction of inputs $x$ where $C(x) \neq f(x)$ is at most $\delta'/\gamma_d$. Since $f$ is assumed to be $\delta$-hard, the algorithm $C$ must err on at least a fraction $\delta$ of inputs. Hence, we get that $\delta' \geqslant \gamma_d\delta$.

The algorithm $C$ to compute $f$ works as follows. On input $x \in \{0,1\}^n$, it will simulate $C'$ on all neighbors of $x$, record the value they "suggest" for $f(x)$, and finally take a majority vote. More formally, on input $x \in \{0,1\}^n$, for each $1 \leqslant i \leqslant d$, let $y_i$ be the $i$'th neighbor of $x$. Compute the bit $b_i = C'(y_i)_{j_i}$, the $j_i$'th component of the $d$-tuple $C'(y_i)$, where $j_i \in \{1, 2, \ldots, d\}$ is such that $x$ is the $j_i$'th neighbor of $y_i$. Finally, output the majority over all the bits $b_i$, $1 \leqslant i \leqslant d$.

It is easily seen that the running time of $C$ is at most $d \cdot (T' + \mathsf{poly}(n))$. The input-preserving space complexity of $C$ can be bounded by $\max\{S_{expander}, S'\} + O(d)$, as in the proof of Lemma 7.

Define the set $Bad = \{x \in \{0,1\}^n \mid C(x) \neq f(x)\}$. Since $f$ is $\delta$-hard for time $T$ and space $S$, and $C$ runs in time $T$ and space $S$, we have $|Bad| \geqslant \delta 2^n$. Let $B$ be an arbitrary subset of $Bad$ of size $\delta 2^n$. By the expansion property of $G$, we have that the set

$$N_G(B) \stackrel{def}{=} \{y \in \{0,1\}^n \mid \exists x \in B \text{ such that } (x, y) \in E(G)\}$$

satisfies

$$|N_G(B)| \geqslant (d/2 + \gamma_d)|B| \tag{3}$$

Since $C$ bases its value for $x$ on a majority vote among neighbors of $x$, the following holds: For each $x \in B$, we must have that at least half of $x$'s neighbors in $G$ must fall in the set

$$W \stackrel{def}{=} \{y \in \{0,1\}^n \mid C'(y) \neq g(y)\}$$

of values that $C'$ gets wrong. Note that $|W| = \delta' 2^n$. In other words, for each $x \in B$, at most $d/2$ neighbors of $x$ fall outside $W$. Hence

$$|N_G(B)| \leqslant |W| + (d/2) \cdot |B| \tag{4}$$

By (3) and (4), we have $|W| \geqslant \gamma_d|B|$, or equivalently $\delta' \geqslant \gamma_d\delta$, as desired. □

Thus, provided explicit expanders with expansion better than $d/2$ are known, we can apply the above amplification repeatedly to get a deterministic linear-space "worst-case to constant average-case" hardness amplification. Unfortunately, we do not know explicit expanders with expansion factor better than $d/2$. The recent work of Capalbo *et al.* [CRVW02] applies only to bipartite graphs (and guarantees expansion of only one of the two sides in the bipartition). Beating the $d/2$ barrier for general graphs remains a challenging open question.

## References

[ABN$^+$92] N. Alon, J. Bruck, J. Naor, M. Naor, and R. Roth. Construction of asymptotically good low-rate error-correcting codes through pseudo-random graphs. *IEEE Transactions on Information Theory*, 38:509–516, 1992.

[BFNW93] L. Babai, L. Fortnow, N. Nisan, and A. Wigderson. BPP has subexponential time simulations unless EXPTIME has publishable proofs. *Computational Complexity*, 3:307–318, 1993.

[BM84] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13:850–864, 1984.

[CRVW02] M.R. Capalbo, O. Reingold, S. Vadhan, and A. Wigderson. Randomness conductors and constant-degree lossless expanders. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, pages 659–668, 2002.

[Din05] I. Dinur. The PCP theorem by gap amplification. *Electronic Colloquium on Computational Complexity*, TR05-046, 2005.

[GI01] V. Guruswami and P. Indyk. Expander-based constructions of efficiently decodable codes. In *Proceedings of the Forty-Second Annual IEEE Symposium on Foundations of Computer Science*, pages 658–667, 2001.

[GI02] V. Guruswami and P. Indyk. Near-optimal linear-time codes for unique decoding and new list-decodable codes over smaller alphabets. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, pages 812–821, 2002.

[GI03] V. Guruswami and P. Indyk. Linear-time encodable and list decodable codes. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, pages 126–135, 2003.

[GNW95] O. Goldreich, N. Nisan, and A. Wigderson. On Yao's XOR-Lemma. *Electronic Colloquium on Computational Complexity*, TR95-050, 1995.

[Imp95] R. Impagliazzo. Hard-core distributions for somewhat hard problems. In *Proceedings of the Thirty-Sixth Annual IEEE Symposium on Foundations of Computer Science*, pages 538–545, 1995.

[IW97] R. Impagliazzo and A. Wigderson. P=BPP if E requires exponential circuits: Derandomizing the XOR Lemma. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 220–229, 1997.

[Lev87] L.A. Levin. One-way functions and pseudorandom generators. *Combinatorica*, 7(4):357–363, 1987.

[NW94] N. Nisan and A. Wigderson. Hardness vs. randomness. *Journal of Computer and System Sciences*, 49:149–167, 1994.

[RVW02] O. Reingold, S. Vadhan, and A. Wigderson. Entropy waves, the zig-zag graph product, and new constant-degree expanders. *Annals of Mathematics*, 155(1):157–187, 2002.

[Spi96] D.A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory*, 42(6):1723–1732, 1996.

[Tre03] L. Trevisan. List-decoding using the XOR lemma. In *Proceedings of the Forty-Fourth Annual IEEE Symposium on Foundations of Computer Science*, pages 126–135, 2003.

[Yao82] A.C. Yao. Theory and applications of trapdoor functions. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982.