

# HARDNESS AMPLIFICATION VIA SPACE-EFFICIENT DIRECT PRODUCTS

VENKATESAN GURUSWAMI AND VALENTINE KABANETS

**Abstract.** We prove a version of the derandomized Direct Product lemma for deterministic space-bounded algorithms. Suppose a Boolean function  $g : \{0, 1\}^n \rightarrow \{0, 1\}$  cannot be computed on more than a fraction  $1 - \delta$  of inputs by any deterministic time  $T$  and space  $S$  algorithm, where  $\delta \leq 1/t$  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  $g'(w) \stackrel{\text{def}}{=} (g(v_1), \dots, g(v_t))$  cannot be computed on more than a fraction  $1 - \Omega(t\delta)$  of inputs by any deterministic time  $\approx T/d^t - \text{poly}(n)$  and space  $\approx S - O(t)$  algorithm. 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 (1996). 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 proof of the PCP theorem by gap amplification using expanders (Dinur 2006).

**Keywords.** Direct products, hardness amplification, error-correcting codes, expanders, random walks.

**Subject classification.** 68Q17, 68Q25, 68P30, 94B05, 94B35

## 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 a 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 “pseudorandom” 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 made by Blum & Micali (1984) and Yao (1982) was that certain average-case hard problems (one-way functions) can be used to build pseudorandom generators. Later Nisan & Wigderson (1994) showed that Boolean functions of sufficiently high average-case circuit complexity can be used to derandomize (i.e., simulate efficiently and deterministically) any probabilistic polynomial-time algorithm.

The construction of Nisan & Wigderson (1994) requires an exponential-time computable Boolean function family  $\{f_n : \{0, 1\}^n \rightarrow \{0, 1\}\}_{n>0}$  such that no Boolean circuit of size  $s(n)$  can agree with  $f_n$  on more than a fraction  $1/2 + 1/s(n)$  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 (Babai *et al.* 1993). They use 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  $g : \{0, 1\}^n \rightarrow \{0, 1\}$ , define  $g^k : (\{0, 1\}^n)^k \rightarrow \{0, 1\}^k$  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. We say that a function  $g$  is  $\delta$ -hard for circuit size  $s$  if  $g$  cannot be computed by circuits of size  $s$  on more than a fraction  $1 - \delta$  of inputs. In this notation, if  $g$  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 a fraction  $(1 - \delta)^k$  of inputs. The result establishing the correctness of this intuition is known as Yao’s Direct Product lemma (Yao 1982), and has a number of different proofs (Goldreich *et al.* 1995; Impagliazzo 1995; Impagliazzo & Wigderson 1997; Levin 1987).

## 1.2. Derandomized Direct Products and Error-Correcting Codes.

Impagliazzo (1995) and Impagliazzo & Wigderson (1997) consider a “deran-

domized” 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  $F : \{0, 1\}^r \rightarrow (\{0, 1\}^n)^k$  such that the input size  $r$  of  $F$  is much smaller than the output size  $kn$ . They give several examples of the function  $F$  so that the average-case hardness of the derandomized Direct Product function is about the same as that of the standard, non-derandomized Direct Product function. In particular, Impagliazzo (Impagliazzo 1995) shows that if  $g$  is  $\delta$ -hard (for certain size circuits) for  $\delta < 1/O(n)$ , then for a pairwise independent  $F : \{0, 1\}^{2n} \rightarrow (\{0, 1\}^n)^n$ , the function  $g'(y) = (g(F(y)_1), \dots, g(F(y)_n))$  is  $\Omega(\delta n)$ -hard (for slightly smaller circuits).

Trevisan (2003) 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. Let us first consider the case of the standard, non-derandomized Direct Product lemma. 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 for circuit size  $s$ , then  $g^k$  is  $\varepsilon \approx 1 - (1 - \delta)^k$ -hard for circuit size  $s'$ , for some  $s' < s$ . Usually the proof is constructive in the sense that it gives an explicit algorithm  $\mathcal{A}$  with the property: Given access to some circuit computing  $g^k$  on all but at most an  $\varepsilon$  fraction of inputs, the algorithm  $\mathcal{A}$  produces a slightly larger circuit that computes  $g$  on all but at most a  $\delta$  fraction of inputs (or a list of circuits such that at least one of them computes  $g$  on all but at most a  $\delta$ -fraction of inputs).

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  $\varepsilon$  fraction of positions, we can construct (a list of  $N$ -bit strings containing) a string  $Msg'$  such that  $Msg$  and  $Msg'$  disagree in less than a  $\delta$  fraction of positions. Here the algorithm  $\mathcal{A}$  from the proof of the Direct Product lemma is used as a decoding algorithm for the corresponding code.

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  $F : \{0, 1\}^r \rightarrow (\{0, 1\}^n)^k$  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 algorithm  $\mathcal{A}$  from the proof of 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 list size at most  $2^m$ . If one wants to get codes with  $\varepsilon$  being asymptotically close to 1, then list-decoding is indeed necessary. However, for a constant  $\varepsilon$ , unique decoding is possible, and so one can hope for 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 in (Impagliazzo 1995; Impagliazzo & Wigderson 1997) are proved using nonuniform reductions. Using the graph-based construction of error-correcting codes of Guruswami & Indyk (2001), Trevisan (2003) 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 (2003) defines the function  $F : \{0, 1\}^n \rightarrow (\{0, 1\}^n)^k$  as  $F(y) = (y_1, \dots, y_k)$ , where the  $y_i$ 's are the neighbors of the vertex  $y$  in the graph  $G_n$ . He then argues that, for a Boolean function  $g : \{0, 1\}^n \rightarrow \{0, 1\}$ , 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  $O(T(n) \cdot \text{poly}(n, k))$  that solves  $g$  on a fraction  $1 - \delta$  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  $g' : \{0, 1\}^n \rightarrow \{0, 1\}^k$  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  $g' : \{0, 1\}^n \times [d]^t \rightarrow \{0, 1\}^{t+1}$  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  $1 \leq j \leq t$ ) 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  $\approx T/d^t - \text{poly}(n)$  and space  $\approx S - O(t)$ .

Note that if  $g$  is  $\delta$ -hard, then we expect  $g^t(x_1, \dots, x_t) = (g(x_1), \dots, g(x_t))$  (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. Our result thus provides yet another example where samples obtained by taking a random walk on an expander graph can be used instead of truly independent random samples without a significant deterioration of parameters of interest (in our case, the hardness  $\delta'$ ).

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  $\Omega(t\delta)$ . Fix  $t$  to be a large enough constant so that this  $\Omega(t\delta)$  is at least  $2\delta$ . 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 obtain a new Boolean function  $g''$  on  $n + O(\log(1/\delta))$  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  $\approx T \cdot \text{poly}(\delta)$  and space  $\approx S - O(\log(1/\delta))$ .

As a corollary, we can convert a worst-case hard function for linear space into a constant average-case hard function for linear space. Using the previously mentioned connection to codes, our methods also imply a family of error-correcting codes that can be encoded and decoded from a constant fraction of errors in logarithmic space. These consequences are not new and also follow from Spielman's work (Spielman 1996) on linear time codes (his codes are also logspace encodable and decodable). This is discussed in more detail in the next section.

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 deter-

ministically encodable/decodable code would use space  $\Omega(k) = \Omega(1/\delta)$ .

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

## 2. Related work

Impagliazzo & Wigderson (1997) use expander walks in combination with the Nisan-Wigderson generator (Nisan & Wigderson 1994) to prove a different derandomized Direct Product lemma. They start with a Boolean function of constant average-case hardness (against circuits) and construct a new Boolean function of average-case hardness exponentially close to  $1/2$ , using probabilistic reductions that take some advice. In contrast, (i) we analyze the hardness of a direct product using vertices of an expander walk only, (ii) our Direct Product lemma works for a different range of parameters (amplifying worst-case hardness to constant average-case hardness), and (iii) our reductions are completely uniform efficient *deterministic* algorithms.

A derandomized Direct Product lemma of Impagliazzo (1995) can be used to amplify inverse polynomial average-case hardness to constant average-case hardness (against circuits). Instead of vertices of an expander walk, the construction of (Impagliazzo 1995) uses pairwise independent probability distributions. Given an  $n$ -variable Boolean function of average-case hardness  $n^{-c}$ , the construction there produces a constant average-case hard Boolean function on about  $(2^c)n$  variables. Our construction is much more efficient in its use of additional variables: it produces a constant average-case hard Boolean function on about  $n + c \log n$  variables.

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 expander-based error-correcting codes of Spielman (1996). 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. Also, using Spielman’s space-efficient code as the inner error-correcting code inside Trevisan’s construction mentioned above, one could obtain a space-efficient version of the Direct Product lemma with parameters matching ours. However, the proof of this Direct Product lemma would rely on a highly non-trivial construction and analysis of codes from (Spielman 1996); as just remarked, Spielman’s codes by themselves already give “worst-case to constant average-case hardness” reduction for linear space. In contrast, our Direct Product lemma does not rely on any sophisti-

cated explicit codes. We only need an error-correcting code for constant-size messages, and such a code can be obtained by a brute-force search.

In view of the connection between Direct Product lemmas and codes, our iterated Direct Product construction also yields a deterministic logspace (in fact, uniform  $\text{NC}^1$ ) encodable/decodable error-correcting code that corrects a constant fraction of errors. Spielman’s  $\text{NC}^1$  encodable/decodable codes also correct a constant fraction of errors, but their other parameters are much better. 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 are 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.

Finally, our proof method is inspired by Dinur’s recent proof of the PCP Theorem (Dinur 2006). She describes a procedure for increasing the unsatisfiability gap of a given unsatisfiable Boolean formula by a constant factor, at the cost of a constant-factor increase in the size of the new formula. Iterating this gap amplification for  $O(\log n)$  steps, she converts any unsatisfiable formula with  $n$  clauses to a polynomially larger formula  $\phi$  such that no assignment can satisfy more than a constant fraction of clauses in  $\phi$ . A single step of gap amplification uses expanders to define a new, harder formula; intuitively, a new formula corresponds to a certain derandomized “direct product” of the old formula, where derandomization is done using constant-length expander walks. In the present paper, we also use constant-size expander walks to derandomize direct products, achieving a constant-factor hardness amplification at the cost of constant additive increase in the space complexity of the new function. Iterating this step  $O(n)$  times, allows us to convert a Boolean function that is worst-case hard for linear space into one that is constant average-case hard for linear space.

**Remainder of the paper.** We give the necessary definitions in Section 3. In Section 4, 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 5. Section 6 proves a simpler version of the Direct Product lemma, under the assumption that degree  $d$  expanders with expansion better than  $d/2$  can be efficiently constructed. We finish with some concluding remarks in Section 7.

### 3. Preliminaries

**3.1. Notation.** For integers  $l \leq r$ , we will denote by  $[l..r]$  the set of all integers  $i$  where  $l \leq i \leq r$ .

A length- $k$  walk on a graph is a sequence of vertices  $v_0, v_1, \dots, v_k$  where  $v_i$  is connected to  $v_{i+1}$  for every  $0 \leq i < k$ . When we say that such a walk passes through a vertex  $v$  in step  $j$ , for some  $0 \leq j \leq k$ , we mean that  $v_j = v$ ; i.e., the walk ends its  $j$ th step at the vertex  $v$ . For a subset  $S$  of vertices, we say that a walk passes through  $S$  in step  $j$ , if the  $j$ th step of the walk is a vertex from  $S$ ; i.e.,  $v_j \in S$ .

**3.2. 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  $f : A \rightarrow B$  (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  $0 \leq \delta \leq 1$  and a bound  $b$  on *Resource*, a function  $f : A \rightarrow B$  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.

Finally, when we talk about a combination of resources (e.g., time  $T$  and space  $S$ ), we mean hardness for algorithms that satisfy all the resource bounds *simultaneously* (e.g., running in time  $T$  and simultaneously in space  $S$ ).

**3.3. Expanders.** Let  $G = (V, E)$  be a  $d$ -regular undirected graph on  $n$  vertices. Let  $A = (a_{ij})$  be the normalized adjacency matrix of  $G$ ; i.e.,  $a_{ij} = E_{ij}/d$  where  $E_{ij}$  is the number of edges between  $i$  and  $j$ . Let  $\lambda_1, \dots, \lambda_n$  be all eigenvalues of  $A$ , ordered in non-increasing order of their absolute values; i.e.,  $|\lambda_1| \geq |\lambda_2| \geq \dots \geq |\lambda_n|$ . It is easy well known (and easy to show) that  $|\lambda_1| = 1$ . For a constant  $\lambda < 1$ , the graph  $G = (V, E)$  is called a  $\lambda$ -*expander* if  $|\lambda_2| \leq \lambda$ .

It is essentially equivalent to define expanders in terms of the following *expansion* property. A  $d$ -regular graph  $G = (V, E)$  is an  $(\alpha, \beta)$ -expander if for every subset  $W \subseteq V$  with  $|W| \leq \alpha|V|$ ,

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

The well-known basic property of expander graphs is fast mixing. Namely, a random  $t$ -step walk from any fixed vertex  $v$  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. It essentially says that the expected number of steps that a random walk lands in some set  $S \subseteq V$  of vertices of an expander graph is at most the density of this set  $S$ , as would trivially be the case for a walk on a complete graph. This remains true even if we condition on the random walk starting at a random vertex of  $S$ , or landing at some vertex of  $S$  in step  $i$  for some fixed  $i$ . A variant of this lemma (for edge sets rather than vertex sets) is proved in (Dinur 2006, Lemma 5.4); the vertex-case is in fact simpler to argue. For completeness, we give the proof in Appendix A.

**LEMMA 3.1.** *Let  $G = (V, E)$  be any  $d$ -regular  $\lambda$ -expander for some constant  $\lambda < 1$ , and let  $S \subseteq V$  be any set. For any integer  $t$ , let  $W_i$ , for  $0 \leq i \leq t$ , be the set of all  $t$ -step walks in  $G$  that land at 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  $t(|S|/|V|) + 2/(1 - \lambda) = t(|S|/|V|) + O(1)$  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 4.1.

**3.4. Space complexity.** We review definitions concerning space complexity, since for our main Direct Product lemma, we need to measure the space complexity of the algorithms very carefully.

**DEFINITION 3.2 (Standard Space Complexity).** *An algorithm computes 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 3.3 (Total Space Complexity).** *An algorithm  $A$  computes a function  $f$  with domain  $\{0, 1\}^n$  in total space  $S$  if on an  $n$ -bit input  $x$ :*

- (i)  $A$  has read/write access to the input tape;

- (ii) in addition to the  $n$  input tape cells,  $A$  is allowed another  $S - n$  tape cells; and,
- (iii) at the end of its computation, the tape contains  $f(x)$ .

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

**DEFINITION 3.4** (Input-Preserving Space Complexity). *An algorithm  $A$  computes a function  $f$  with domain  $\{0, 1\}^n$  in input-preserving space  $S$  if on an  $n$ -bit input  $x$ :*

- (i)  $A$  has read/write access to the input tape;
- (ii) in addition to the  $n$  input tape cells,  $A$  is allowed another  $S - n$  tape cells; and,
- (iii) at the end of its computation,  $A$  has on its tape  $x; f(x)$  (i.e., both the input and the value of the function  $f$  on that input, separated by the special symbol “;”).

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 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  $f'(x) \stackrel{\text{def}}{=} x; f(x)$ .)

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

**FACT 3.5.** *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 (respectively, total) space complexity  $S + n$  to compute  $f$ . Conversely, if there is an algorithm  $B'$  with input-preserving (respectively, 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 3.5, our end result can be stated in terms of the standard space complexity of Definition 3.2.

## 4. A New Direct Product Lemma

**4.1. Construction.** We need the following two ingredients:

(i) [**expander graphs**] Let  $G = (V, E)$  be any efficiently constructible  $d$ -regular  $\lambda$ -expander on  $|V| = 2^n$  vertices which are identified with  $n$ -bit strings (here  $d$  and  $\lambda < 1$  are absolute constants, and 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  $T_{\text{expander}} = \text{poly}(n)$  and total space  $S_{\text{expander}} = O(n)$ , which given as input an  $n$ -bit string  $x$  and an index  $i \in [d]$ , outputs the pair  $N_G(x, i) \stackrel{\text{def}}{=} (y, j)$ , 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  $y$ .

REMARK 4.1. We can obtain such expander graphs from (Gabber & Galil 1981; Lubotzky et al. 1988; Reingold et al. 2002). For instance, Gabber & Galil (1981) show how to construct constant-degree expanders of size  $m^2$ , for any natural number  $m$ . It is easy to verify that their family of expanders is efficiently constructible. Note that normally the space complexity of expander constructions is measured in the sense of Definition 3.2; however, by Fact 3.5, for  $O(n)$  space, we can pass freely to the total space complexity model of Definition 3.3. One small technical problem is that we need graphs of size  $2^n$  for every  $n$ , whereas the Gabber-Galil graphs are of size  $m^2$  for every  $m$ . We can solve this problem easily as follows. For even  $n = 2k$ , we can just use the Gabber-Galil graph with  $m = 2^k$ . For odd  $n = 2k - 1$ , we first construct the Gabber-Galil graph for  $m = 2^k$  of size  $m^2 = 2^{n+1}$ , and then merge pairs of vertices  $(2i, 2i + 1)$  for all  $0 \leq i < 2^n$ . This yields a new graph on exactly  $2^n$  vertices, with constant degree (which is twice the degree of the original graph). It is also easy to see that the expansion properties of the new graph get worse by at most a constant factor, so it is still a good expander. To have the graphs of the same degree for both even and odd  $n$ , we can arbitrarily add enough new edges to the graph obtained for the case of even  $n$ , to double its degree.

(ii) [**error-correcting codes**] For an integer parameter  $t$ , let  $\mathcal{C}$  be a binary linear error-correcting code of dimension  $t + 1$ , block length  $c(t + 1)$  for an integer  $c$ , and which has relative distance  $\rho > 0$ . Assume that  $\mathcal{C}$  can be encoded as well as decoded up to a fraction  $\rho/2$  of errors in  $\text{poly}(t)$  time and  $O(t)$  space.

REMARK 4.2. We can get such explicit codes from, e.g., (Justesen 1972). We would like to point out that we will need to encode and decode messages of

constant length  $t + 1$  only. We can exhaustively search for a linear error-correcting code with constant rate and constant relative distance (independent of  $t$ ); such a code can be shown to exist by a counting argument. Such a code can be found, encoded and decoded in time and space that is exponential in  $t$  (the encoding can in fact be done in polynomial time since the code is linear). Since  $t$  is a constant, all these costs are just constant. As will become clear from the analysis of our Direct Product construction, we can tolerate such constant costs associated with the code  $\mathcal{C}$ . So for our purposes, even an exhaustively constructed linear code would do.

Our construction proceeds in two steps.

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

$$g(v, i_1, \dots, i_t) = (f(v), f(v_{i_1}), \dots, f(v_{i_t})),$$

where for each  $1 \leq j \leq t$ ,  $v_j$  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:** Define a Boolean function  $h : \{0, 1\}^n \times [d]^t \times [c(t+1)] \rightarrow \{0, 1\}$  as

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

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

**Complexity of the construction:** 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  $T_g = O(t \cdot (T + T_{\text{expander}})) = O(t \cdot (T + \text{poly}(n)))$  and input-preserving space at most  $S_g = \max\{S, S_{\text{expander}}\} + O(t)$ .

Indeed, to compute  $g(v, i_1, \dots, i_t)$ , we first compute  $f(v)$  using time  $T$  and input-preserving space  $S$ . We then re-use this space  $S$  to compute  $N_G(v, i_1) = (v_1, j_1)$  in time  $T_{\text{expander}}$  and total space  $S_{\text{expander}}$ . 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 time  $T$  and 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 time  $O(t \cdot (T + n + T_{\text{expander}})) = O(t \cdot (T + \text{poly}(n)))$  and total space  $\max\{S, S_{\text{expander}}\} + O(t)$ . 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. This can be done with no extra space (by re-using space) and in time  $O(t \cdot (n + T_{\text{expander}})) = O(t \cdot \text{poly}(n))$ .

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

**4.2. Analysis.** We will show that the “Direct Product construction” described above increases the hardness of a Boolean function  $f$  by a multiplicative factor of  $\Omega(t)$ .

**LEMMA 4.3 (Direct Product Lemma).** *Suppose an  $n$ -variable Boolean function  $f$  has hardness  $\delta \leq 1/t$  for deterministic time  $T$  and input-preserving space  $S \geq S_{\text{expander}} + O(t)$ . 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  $T' = T/O(t^2d^t) - \text{poly}(n, t)$  and input-preserving space  $S' = S - O(t)$ .*

The proof of the Direct Product lemma above will consist of two parts, given by Lemma 4.4 and Lemma 4.7 below. First we argue that the non-Boolean function  $g$  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  $t$ .

**LEMMA 4.4.** *Suppose an  $n$ -variable Boolean function  $f$  has hardness  $\delta \leq 1/t$  for deterministic time  $T$  and input-preserving space  $S \geq S_{\text{expander}} + O(t)$ . 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  $\Omega(t\delta)$  for deterministic time  $T' = T/O(td^t) - t \cdot \text{poly}(n)$  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 a fraction  $1 - \delta'$  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  $\delta'$  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  $\delta' \geq \Omega(t\delta)$ .

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 ends its  $i$ th step at  $x$  and  $C'(w)_i$  is the  $i$ th bit in the  $(t + 1)$ -tuple output by the circuit  $C'$  on input  $w$ . Output the majority over all the values  $b_i$ , for  $0 \leq i \leq t$ . A more formal description of the algorithm is given in Algorithm 1, below.

```

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

```

**Algorithm 1:**  $C$

It is easy to argue that the input-preserving space complexity  $S$  of algorithm  $C$  is at most  $\max\{S_{\text{expander}}, S'\} + O(t)$ ; the argument goes along the lines of the one we used to argue about the complexity of the encoding at the end of Section 4.1. Hence by choosing  $S' = S - O(t) \geq S_{\text{expander}}$  we get the input-preserving space complexity of  $C$  at most  $S$ . It is also easy to verify that algorithm  $C$  can be implemented in deterministic time  $O(td^t(T' + t \cdot \text{poly}(n)))$ . By choosing  $T'$  as in the statement of the lemma, we can ensure that the running time of  $C$  is at most  $T$ .

We now analyze how many mistakes the algorithm  $C$  makes in computing  $f$ . Define the set  $Bad = \{x \in \{0, 1\}^n \mid C(x) \neq f(x)\}$ . Observe that by the  $\delta$ -hardness assumption about  $f$ , we have  $|Bad|/|V| \geq \delta$ . Choose a maximal-size subset  $B \subseteq Bad$  so that  $|B|/|V| \leq 1/t$ . That is, if  $|Bad|/|V| \leq 1/t$ , set

$B = Bad$ ; otherwise, form  $B$  by removing some elements from  $Bad$  so that  $|B|/|V| = 1/t$ . In the first case, we obviously have  $|B|/|V| \geq \delta$ . Since we assumed that  $1/t \geq \delta$ , the set  $B$  obtained in the second case is still such that  $|B|/|V| \geq \delta$ .

By definition, if  $x \in Bad$ , then for each of at least  $1/2$  of the values of  $i \in [0..t]$ , the algorithm  $C'$  is wrong on at least half of all  $t$ -step walks that end their  $i$ th step at  $x$ . 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 0's 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 0's. This means that for each  $i \in I$ , there is at least  $B/4$  inputs  $x$  such that the algorithm  $C'$  is wrong on at least  $1/2$  of all  $t$ -step walks that end their  $i$ th step at  $x$ . That is,  $C'$  is wrong on at least  $(|B|/4)(d^t/2) = |B|d^t/8$  of walks, which is  $1/8$  of all  $|B|d^t$  length- $t$  walks that end their  $i$ th step at a vertex in the set  $B$ .

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  $|W_{i,x}| = d^t$ . Taking the union over all  $x \in B$ , we get the set  $W_i = \cup_{x \in B} W_{i,x}$  of all  $t$ -step walks passing through some vertex in  $B$  at step  $i$ . 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  $W_i^* = \cup_{x \in B} W_{i,x}^*$ . Note that for each  $i \in I$ ,  $|W_i^*| \geq |W_i|/8$ . Finally, define  $W^* = \cup_{i=0}^t W_i^*$ ; by construction, for every  $w \in W^*$ ,  $C'(w) \neq g(w)$ , so it suffices to give a lower bound on  $|W^*|$  to argue that  $C'$  makes many mistakes.

If the sets  $W_i^*$ s were disjoint, we could lowerbound  $|W^*|$  by  $\sum_{i \in I} |W_i^*| \geq (t/4)(|W_i|/8) = t|B|d^t/32$ , which would mean that  $C'$  errs on at least a fraction  $(t|B|d^t/32)/(|V|d^t) = (t/32)|B|/|V| \geq (t/32)\delta$  of inputs. The problem is that the  $W_i^*$ s are not necessarily disjoint, and so  $\sum_{i \in I} |W_i^*|$  is bigger than  $|\cup_{i \in I} W_i^*|$ . However, as we argue below using the properties of expander walks, the difference between the two expressions is small.

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  $t$ ) so that for all  $i$ ,  $|H_i| \leq |W_i|/16$ . Indeed, by Lemma 3.1, for every  $i$  a random walk  $w \in W_i$  is expected to contain at most  $t(|B|/|V|) + O(1)$  vertices from  $B$ . Since  $(|B|/|V|) \leq 1/t$ , a random  $w \in W_i$  contains on average at most  $c = O(1)$  vertices from  $B$ . By Markov's inequality, the probability that a random  $w \in W_i$

contains more than  $m = 16c$  vertices from  $B$  is at most  $1/16$ . Thus we have

$$(4.5) \quad \sum_{i \in I} |W_i^* \setminus H_i| = \sum_{i \in I} (|W_i^*| - |H_i|) \geq |I| \left( \frac{1}{8} - \frac{1}{16} \right) |W_i| \geq \frac{t}{64} |B| d^t.$$

On the other hand, since every walk from  $W_i^* \setminus H_i$  can appear in at most  $m$  different  $W_i^*$ s, we get

$$(4.6) \quad \sum_{i \in I} |W_i^* \setminus H_i| \leq m |\cup_{i \in I} (W_i^* \setminus H_i)| \leq m |W^* \setminus (\cup_{i=0}^t H_i)| \leq m |W^*|.$$

Combining (4.5) and (4.6), we get  $|W^*| \geq \frac{t}{64m} |B| d^t$ . 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  $C'$  makes mistakes on at least a fraction  $\frac{t|B|}{64m|V|}$  of inputs. As observed earlier,  $\frac{|B|}{|V|} \geq \delta$ . Hence the function  $g$  is  $\Omega(t\delta)$ -hard for time  $T'$  and input-preserving space  $S'$ .  $\square$

The second step, Lemma 4.7, of our Direct Product construction uses the standard approach of “code concatenation”.

**LEMMA 4.7.** *Let  $A = \{0, 1\}^n \times [d]^t$ . Suppose that a function  $g : A \rightarrow \{0, 1\}^{t+1}$  is  $\delta$ -hard for deterministic time  $T$  and input-preserving space  $S$ . Let  $h : A \times [c \cdot (t + 1)] \rightarrow \{0, 1\}$  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  $T' = (T - \text{poly}(t))/O(t)$  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 a fraction  $1 - \delta'$  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  $c(t + 1)T' + \text{poly}(t)$ , where the  $\text{poly}(t)$  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  $(\rho/2)$ -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  $|Bad|(\rho/2)c \cdot (t+1)$ . Dividing this number by the total number  $|A|c \cdot (t+1)$  of inputs to  $C'$ , we get that  $C'$  is incorrect on a fraction  $\delta' \geq (\rho/2)(|Bad|/|A|)$  of inputs. Since  $g$  is assumed  $\delta$ -hard for time  $T$  and input-preserving space  $S$ , we get that  $|Bad|/|A| \geq \delta$ . It follows that  $\delta' \geq (\rho/2)\delta = \Omega(\delta)$ .  $\square$

**4.3. Iteration.** Our Direct Product lemma, Lemma 4.3, 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 function that is  $\delta$ -hard with respect to LINSPEACE into a function that is  $\Omega(1)$ -hard with respect to LINSPEACE. 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. The following result is implicit in (Spielman 1996), using his logspace encodable/decodable error-correcting codes that can correct a constant fraction of errors.

**COROLLARY 4.8.** *Let  $f$  be an  $n$ -variable Boolean function that is  $\delta$ -hard for deterministic time  $T$  and space  $S \geq O(n)$ . Then there is a Boolean function  $f'$  on  $n + O(\log(1/\delta))$  variables such that  $f'$  is  $\Omega(1)$ -hard for deterministic time  $T' = T \cdot \text{poly}(\delta) - \text{poly}(n)$  and space  $S' = S - n - O(\log(1/\delta))$ . Moreover, if  $f$  is computable in time  $\tilde{T}$  and space  $\tilde{S}$ , then  $f'$  is computable in time  $\tilde{T} \text{poly}(1/\delta) + \text{poly}(n/\delta)$  and space  $\tilde{S} + O(n)$ .*

**PROOF.** Pick a constant  $t$  large enough so that the  $\Omega(t)$  factor in the statement of Lemma 4.3 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 3.5, such an  $f$  is  $\delta$ -hard for deterministic time  $T$  and input-preserving space  $S$ . Let  $f'$  be the Boolean function obtained from  $f$  by repeated application of the Direct Product construction for  $\log \frac{1}{\delta}$  steps (using an expander with  $S_{\text{expander}} = O(n)$ ). Then it is straightforward to check that  $f'$  is a  $n + O(t \cdot \log \frac{1}{\delta})$ -variable Boolean function of  $\Omega(1)$ -hardness for deterministic time  $T' = T\delta^{O(t)} - \text{poly}(n)$  and input-preserving space  $S'' = S - O(t \cdot \log(1/\delta))$ . Referring to Fact 3.5 again,  $f'$  is  $\Omega(1)$ -hard for deterministic time  $T'$  and space  $S' = S'' - n = S - n - O(t \cdot \log(1/\delta))$ .

The time and space upper bounds for  $f'$  follow easily from the complexity analysis of the Direct Product construction (and using Fact 3.5 to convert from space to input-preserving space and back).  $\square$

REMARK 4.9. *The constant average-case hardness in Corollary 4.8 above, say  $\gamma$ -hardness for some absolute constant  $\gamma > 0$ , can be boosted to any constant less than  $1/4$ , say  $1/4 - \varepsilon$  for some small  $\varepsilon > 0$ . This can be achieved by one additional amplification with a suitable expander, followed by concatenation with a constant-sized binary code of relative distance close to  $1/2$ . The time and space bounds for which we get  $(1/4 - \varepsilon)$ -hardness deteriorate by just constant factors depending on  $\gamma, \varepsilon$ . For details, see (Guruswami & Indyk 2001) where this approach is used to construct binary codes of positive rate that can be encoded and decoded up to a fraction  $(1/4 - \varepsilon)$  of errors in linear time, starting with a linear time code that could correct a fraction  $\gamma > 0$  of errors.<sup>1</sup> The relevance of this method for deterministic uniform hardness amplification was realized by Trevisan (2003) (specifically, see Theorem 7 in (Trevisan 2003)).*

## 5. Applications

**5.1. Hardness amplification via deterministic space-efficient reductions.** The iterated Direct Product construction of Corollary 4.8 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 4.8 and Remark 4.9. Below we use standard notation for the complexity classes  $\mathbf{E} = \mathbf{DTIME}(2^{O(n)})$  and  $\mathbf{LSPACE} = \mathbf{SPACE}(O(n))$ . When we say that a language  $L'$  is infinitely often  $\alpha$ -hard for complexity class  $C$ , we mean that there is no  $C$ -type algorithm which, for all but finitely many input lengths  $n$ , decides  $L'$  on more than a fraction  $1 - \alpha$  of  $n$ -bit inputs. In other words, for every  $C$ -type algorithm  $A$ , there exist infinitely many input lengths  $n$  such that  $A$  incorrectly decides  $L'$  on at least a fraction  $\alpha$  of  $n$ -bit inputs.

THEOREM 5.1. *Let  $\alpha < 1/4$  be an arbitrary constant. If there is a language  $L \in \mathbf{E} \setminus \mathbf{LSPACE}$ , then there is a language  $L' \in \mathbf{E}$  that is infinitely often  $\alpha$ -hard for  $\mathbf{LSPACE}$ .*

PROOF. View a given language  $L$  as a family of Boolean functions  $\{f_n\}_{n \geq 0}$ , where  $f_n$  is the characteristic function of the  $n$ th slice of  $L$ , i.e., the set  $L \cap \{0, 1\}^n$ . For each  $n$ , define the function  $f'$  to be the encoding of  $f$  given in Corollary 4.8 for  $\delta = 2^{-n}$ . The function  $f'$  will have  $c \cdot n$  inputs, for some constant  $c$ . Define the language  $L'$  so that the functions  $f'$  are its characteristic

<sup>1</sup>The underlying expander-based approach was first used in (Alon *et al.* 1992) to construct simple codes of relative distance  $(1 - \varepsilon)$  over an alphabet of size  $2^{O(1/\varepsilon)}$ .

functions for input length  $c \cdot n$ , and define  $L'$  arbitrarily on all other lengths. Then  $L'$  is constant average-case hard for LINSPLACE.

Indeed, suppose there is a LINSPLACE algorithm that decides  $L'$  well on average for all sufficiently large input lengths. In particular, this algorithm will succeed on input lengths  $c \cdot n$  for all sufficiently large  $n$ . Then an algorithm that on input  $x \in \{0, 1\}^n$  runs the decoding procedure from the proof of Corollary 4.8 on the  $(c \cdot n)$ th slice of  $L'$  will correctly decide  $L$  for all sufficiently large input lengths  $n$ .  $\square$

Similarly, we can also prove the following.

**THEOREM 5.2.** *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  $L \in \text{LINSPLACE}$  that cannot be computed by any deterministic algorithm running in linear space and, simultaneously, time  $2^{c'n}$ , then there is a language  $L' \in \text{LINSPLACE}$  that is infinitely often  $\alpha$ -hard for any deterministic algorithm running in linear space and, simultaneously, time  $2^{c'n}$ .*

Theorem 5.1 and Theorem 5.2 are stated in the “infinitely often” setting. Both theorems can also be stated in the “almost everywhere” setting, where the assumption and conclusion will be about hardness almost everywhere (i.e., hardness for all sufficiently large input lengths).

For instance, the “almost-everywhere” version of Theorem 5.1 will be as follows. Below, when we say that a language  $L \notin \text{ioLINSPLACE}$ , we mean that every LINSPLACE algorithm will incorrectly decide  $L$  for all sufficiently large input lengths  $n$ .

**THEOREM 5.3.** *Let  $\alpha < 1/4$  be an arbitrary constant. If there is a language  $L \in \mathbf{E} \setminus \text{ioLINSPLACE}$ , then there is a language  $L' \in \mathbf{E}$  that is almost everywhere  $\alpha$ -hard for LINSPLACE.*

**PROOF.** As before, a language  $L \notin \text{ioLINSPLACE}$  yields a family of  $n$ -variable Boolean functions hard for ioLINSPLACE. Each such function will be encoded by our iterated Direct Product construction as a  $c \cdot n$ -variable function  $f'$ . We define  $L'$  so that for every  $x$  of length  $c \cdot n + d$ , for  $0 \leq d < c$ ,  $x \in L'$  iff  $f'(x') = 1$  for the  $(c \cdot n)$ -length prefix  $x'$  of  $x$ .

Suppose there is a LINSPLACE algorithm  $A$  that decides  $L'$  well on average for infinitely many input lengths. By averaging, there exists a value  $0 \leq d_0 < c$  such that  $A$  decides  $L'$  well on average for infinitely many input lengths of the form  $c \cdot n + d_0$ . By another averaging argument, there exists a binary string  $y_0 \in \{0, 1\}^{d_0}$  such that  $A$  decides  $L'$  well on average for infinitely many input

lengths of the form  $c \cdot n + d_0$  where the  $d_0$ -length suffix of the input is fixed to the string  $y_0$ . Given these  $d_0$  and  $y_0$ , we can use  $A$  to compute the function  $f'$  well on average for infinitely many input lengths. Hence, by hardwiring these values  $d_0$  and  $y_0$ , we get a LINSPACE algorithm computing the function  $f$  for infinitely many input lengths.  $\square$

**5.2. Logspace encodable/decodable error-correcting codes.** As mentioned in the introduction (Section 1.2), 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 polynomial rather than linear rate) to Spielman's logspace encodable/decodable codes (Spielman 1996).

**THEOREM 5.4.** *There is an explicit code  $\mathcal{C}$  mapping  $n$ -bit messages to  $\text{poly}(n)$ -bit codewords such that*

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

**REMARK 5.5.** *We are not aware of any logspace encodable/decodable asymptotically good codes other than Spielman's construction (Spielman 1996), along with improvements to its error-correction performance (Guruswami & Indyk 2001, 2002). Allowing  $\text{NC}^2$  complexity gives several other choices of error-correcting codes.*

## 6. A simple graph based amplification

Here we observe that the existence of efficiently constructible  $d$ -regular expanders with vertex expansion factor better than  $d/2$  would give us another deterministic linear-space hardness amplification. We recall Trevisan's derandomized Direct Product construction below. We note that a similar definition has been used in the construction of codes in several works beginning with (Alon *et al.* 1992) and more recently in (Guruswami & Indyk 2001, 2003).

DEFINITION 6.1. Given a  $d$ -regular graph  $G$  on  $2^n$  vertices, where each vertex is identified with an  $n$ -bit string, and a Boolean function  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ , we define a function  $g = G(f) : \{0, 1\}^n \rightarrow \{0, 1\}^d$  as follows. For  $x \in \{0, 1\}^n$ , let  $N_1(x), N_2(x), \dots, N_d(x)$  denote the  $d$  neighbors of  $x$  in  $G$  (as per some fixed ordering). Then  $g(x) \stackrel{\text{def}}{=} (f(N_1(x)), f(N_2(x)), \dots, f(N_d(x)))$ .

LEMMA 6.2. Let  $G = (\{0, 1\}^n, E)$  be an efficiently (in total space  $S_{\text{expander}}$  and  $\text{poly}(n)$ -time) constructible  $d$ -regular  $(\delta, d/2 + \gamma_d)$ -expander for some  $\gamma_d > 1$ . Let  $f : \{0, 1\}^n \rightarrow \{0, 1\}$  be  $\delta$ -hard for deterministic time  $T$  and input-preserving space  $S \geq S_{\text{expander}} + \Omega(d)$ . Then the function  $g = G(f)$  from Definition 6.1 is  $\gamma_d \delta$ -hard for deterministic time  $T' = \frac{T}{d} - \text{poly}(n)$  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  $\delta' \geq \gamma_d \delta$ .

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. It is easily seen that the running time of  $C$  is at most  $d \cdot (T' + \text{poly}(n))$ . The input-preserving space complexity of  $C$  can be bounded by  $\max\{S_{\text{expander}}, S'\} + O(d)$ , as in the proof of Lemma 4.4.

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  $|Bad| \geq \delta 2^n$ . Let  $B$  be an arbitrary subset of  $Bad$  of size  $\delta 2^n$ . By the expansion property of  $G$ , we have that the set

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

satisfies

$$(6.3) \quad |N_G(B)| \geq (d/2 + \gamma_d)|B|.$$

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

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

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

$$(6.4) \quad |N_G(B)| \leq |W| + (d/2) \cdot |B|$$

By (6.3) and (6.4), we have  $|W| \geq \gamma_d |B|$ , or equivalently  $\delta' \geq \gamma_d \delta$ , as desired.  $\square$

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 work of Capalbo *et al.* (2002) applies only to bipartite graphs. Beating the  $d/2$  barrier for general graphs remains a challenging open question.

## 7. Concluding remarks

We proved a version of a derandomized Direct Product lemma in the setting of uniform space-bounded deterministic algorithms. Previously such Direct Product lemmas were proved for the nonuniform setting of Boolean circuits (Impagliazzo 1995; Impagliazzo & Wigderson 1997), and for uniform time-bounded deterministic algorithms (Trevisan 2003). Our hardness amplification is iterative. Starting with a Boolean function that is  $\delta$ -hard against deterministic space  $S$  algorithms (for  $\delta$  smaller than some universal constant), we get after a single iteration a new Boolean function that is  $2\delta$ -hard against algorithms using space about  $S - O(1)$ . Using  $O(n)$  such iterations, we can convert an  $n$ -variable Boolean function which is worst-case hard against linear space (i.e.,  $\delta = 2^{-n}$ ) into a Boolean function that is  $\Omega(1)$  average-case hard against linear space.

We want to stress that our construction is another example of an iterative algorithm that gradually transforms a given input object (in our case, a Boolean function of worst-case hardness) into a new object with improved parameters of interest (in our case, a Boolean function of constant average-case hardness) — a survey by Goldreich (2005) explicitly discusses this recently popular and influential paradigm. Our construction is inspired by two recent breakthrough constructions of Reingold (2005) and Dinur (2006), which in turn rely on the ideas of the celebrated iterative construction of expander graphs by Reingold, Vadhan, and Wigderson (Reingold *et al.* 2002).

We anticipate that there will be further examples of such iterative, “gradual-improvement” algorithms in the future.

## Acknowledgements

We are grateful to the anonymous reviewers for detailed comments on the presentation. Venkatesan Guruswami is supported in part by NSF grant CCF-0343672, a Sloan Research Fellowship, and a David and Lucile Packard Foundation Fellowship. Valentine Kabanets is supported in part by an NSERC Discovery grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## References

- N. ALON, J. BRUCK, J. NAOR, M. NAOR & R. ROTH (1992). Construction of asymptotically good low-rate error-correcting codes through pseudo-random graphs. *IEEE Transactions on Information Theory* **38**, 509–516.
- L. BABAI, L. FORTNOW, N. NISAN & A. WIGDERSON (1993). BPP has subexponential time simulations unless EXPTIME has publishable proofs. *Computational Complexity* **3**, 307–318.
- M. BLUM & S. MICALI (1984). How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing* **13**, 850–864.
- M.R. CAPALBO, O. REINGOLD, S. VADHAN & A. WIGDERSON (2002). Randomness conductors and constant-degree lossless expanders. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, 659–668.
- I. DINUR (2006). The PCP theorem by gap amplification. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, 241–250.
- O. GABBER & Z. GALIL (1981). Explicit construction of linear sized superconcentrators. *Journal of Computer and System Sciences* **22**, 407–420.
- O. GOLDREICH (2005). Bravely, Moderately: A Common Theme in Four Recent Results. *Electronic Colloquium on Computational Complexity* **TR05-098**.
- O. GOLDREICH, N. NISAN & A. WIGDERSON (1995). On Yao’s XOR-Lemma. *Electronic Colloquium on Computational Complexity* **TR95-050**.

- V. GURUSWAMI & P. INDYK (2001). Expander-based constructions of efficiently decodable codes. In *Proceedings of the Forty-Second Annual IEEE Symposium on Foundations of Computer Science*, 658–667.
- V. GURUSWAMI & P. INDYK (2002). 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*, 812–821.
- V. GURUSWAMI & P. INDYK (2003). Linear-time encodable and list decodable codes. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, 126–135.
- R. IMPAGLIAZZO (1995). Hard-core distributions for somewhat hard problems. In *Proceedings of the Thirty-Sixth Annual IEEE Symposium on Foundations of Computer Science*, 538–545.
- R. IMPAGLIAZZO & A. WIGDERSON (1997).  $P=BPP$  if  $E$  requires exponential circuits: Derandomizing the XOR Lemma. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, 220–229.
- J. JUSTESEN (1972). A class of constructive asymptotically good algebraic codes. *IEEE Transactions on Information Theory* **18**, 652–656.
- L.A. LEVIN (1987). One-way functions and pseudorandom generators. *Combinatorica* **7**(4), 357–363.
- A. LUBOTZKY, R. PHILLIPS & P. SARNAK (1988). Ramanujan graphs. *Combinatorica* **8**(3), 261–277.
- N. NISAN & A. WIGDERSON (1994). Hardness vs. Randomness. *Journal of Computer and System Sciences* **49**, 149–167.
- O. REINGOLD (2005). Undirected ST-Connectivity in Log-Space. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, 376–385.
- O. REINGOLD, S. VADHAN & A. WIGDERSON (2002). Entropy waves, the zig-zag graph product, and new constant-degree expanders. *Annals of Mathematics* **155**(1), 157–187.
- D.A. SPIELMAN (1996). Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory* **42**(6), 1723–1732.
- L. TREVISAN (2003). List-decoding using the XOR lemma. In *Proceedings of the Forty-Fourth Annual IEEE Symposium on Foundations of Computer Science*, 126–135.

A.C. YAO (1982). Theory and applications of trapdoor functions. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Foundations of Computer Science*, 80–91.

## A. Proof of Lemma 3.1

We will first prove the following lemma.

LEMMA A.1. *Let  $G = (V, E)$  be any  $d$ -regular  $\lambda$ -expander, and let  $S \subset V$  be any set. Let  $i \geq 0$  be an integer. The probability that a random walk starting from a random vertex in  $S$  visits a vertex in  $S$  after exactly  $i$  steps is at most  $|S|/|V| + \lambda^i$ .*

PROOF. Let  $x \in \mathbb{R}^V$  be the vector corresponding to the distribution of the start vertex of the walk. That is,  $x_v = 1/|S|$  for  $v \in S$ , and  $x_v = 0$  for  $v \in V \setminus S$ . The distribution of the vertex where the walk lands after taking  $i$  steps is given by  $A^i x$ . Thus, the probability  $p$  that the walk visits  $S$  after  $i$  steps is equal to

$$(A.2) \quad p = \sum_{v \in S} (A^i x)_v = |S| \sum_{v \in V} x_v (A^i x)_v = |S| \langle x, A^i x \rangle .$$

Let  $x = x^{\parallel} + x^{\perp}$  where  $x^{\parallel}$  corresponds to the uniform distribution, i.e.,  $x^{\parallel} = \mathbf{1}/|V|$ , where  $\mathbf{1}$  is the all 1's vector. Since both  $x$  and  $x^{\parallel}$  correspond to distributions,  $\langle \mathbf{1}, x^{\perp} \rangle = 0$ , and thus  $x^{\perp}$  is orthogonal to the principal eigenvector of  $A$ . Hence  $\|A^i x^{\perp}\| \leq \lambda^i \|x^{\perp}\| \leq \lambda^i \|x\|$ , where  $\|y\| = \sqrt{\sum_v y_v^2}$  denotes the  $L_2$ -norm of a vector  $y$ . Now  $\langle x, A^i x \rangle = \langle x, A^i(x^{\parallel} + x^{\perp}) \rangle = \langle x, x^{\parallel} \rangle + \langle x, A^i x^{\perp} \rangle$ , and so

$$(A.3) \quad \langle x, A^i x \rangle \leq \langle x, x^{\parallel} \rangle + \|x\| \|A^i x^{\perp}\| \leq \langle x, x^{\parallel} \rangle + \lambda^i \|x\|^2 .$$

We have  $\|x\|^2 = 1/|S|$  and  $\langle x, x^{\parallel} \rangle = 1/|V|$ . Combining (A.2) and (A.3), we get  $p \leq \frac{|S|}{|V|} + \lambda^i$ , as claimed.  $\square$

Turning to Lemma 3.1, first consider the case  $i = 0$ , i.e., length  $t$  walks that begin at a vertex in  $S$ . By Lemma A.1 above, the expected number of vertices of the walk that lie in  $S$  (including the start vertex) is at most  $1 + \sum_{j=1}^t \left( \frac{|S|}{|V|} + \lambda^j \right) \leq t|S|/|V| + \frac{1}{1-\lambda} = t|S|/|V| + O(1)$ .

When  $1 \leq i \leq t$ , we do the above analysis separately for the part of the walk consisting of the last  $t - i$  steps and the reverse of the path consisting of

the first  $i$  steps, both of which begin at a vertex in  $S$ . Adding the expected number of vertices in  $S$  in these parts, we obtain a bound of

$$1 + \sum_{j=1}^i \left( \frac{|S|}{|V|} + \lambda^j \right) + \sum_{j=1}^{t-i} \left( \frac{|S|}{|V|} + \lambda^j \right) \leq \frac{t|S|}{|V|} + \frac{2}{1-\lambda} = \frac{t|S|}{|V|} + O(1)$$

for the overall expectation.

Manuscript received 8 February 2006

VENKATESAN GURUSWAMI  
Department of Computer Science and  
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