Uniform Hardness Amplification in NP
via Monotone Codes

Joshua Buresh-Oppenheim   Valentine Kabanets   Rahul Santhanam

School of Computing Science
Simon Fraser University
Vancouver, BC
Canada
{jburesho,kabanets,rsanthan}@cs.sfu.ca

December 4, 2006

Abstract

We consider the problem of amplifying uniform average-case hardness of languages in NP, where hardness is with respect to BPP algorithms. We introduce the notion of *monotone* error-correcting codes, and show that hardness amplification for NP is essentially equivalent to constructing efficiently *locally* encodable and *locally* list-decodable monotone codes. The previous hardness amplification results for NP [Tre03, Tre05] focused on giving a direct construction of some *locally* encodable/decodable monotone codes, running into the problem of large amounts of nonuniformity used by the decoding algorithm. In contrast, we propose the *indirect* approach to constructing locally encodable/decodable monotone codes, combining the uniform Direct Product Lemma of [IJK06] and arbitrary, *not necessarily locally encodable*, monotone codes. The latter codes have fewer restrictions, and so may be easier to construct.

We study what parameters are achievable by monotone codes in general, giving negative and positive results. We present two constructions of monotone codes. Our first code is a uniquely decodable code based on the Majority function, and has an efficient decoding algorithm. Our second code is combinatorially list-decodable, but we do not have an efficient decoding algorithm. In conjunction with an appropriate Direct Product Lemma, our first code yields uniform hardness amplification for NP from inverse polynomial to constant average-case hardness. Our second code, even with a brute-force decoding algorithm, yields further hardness amplification to $1/2 - \log^{-\Omega(1)} n$. Together, these give an alternative proof of Trevisan's result [Tre03, Tre05]. Getting any non-brute-force decoding algorithm for our second code would imply improved parameters for the problem of hardness amplification in NP.

1   Introduction

We consider the problem of hardness amplification of Boolean functions within NP. Given a Boolean function family (equivalently, a language) computable in NP which is "somewhat hard" to compute by any efficient randomized algorithm, we would like to define a new Boolean function family in NP which will be "even harder" to compute by the same class of efficient randomized algorithms.

The terms "somewhat hard" and "even harder" can be interpreted in several ways. First, one may want to take an NP function which is *worst-case* hard with respect to any randomized polynomial-time algorithm, and produce another NP function which will be *average-case* hard with respect to the same class of algorithms. While very desirable (especially for cryptography), such a

generic "worst-to-average-case" reduction for functions in NP is currently unknown, and moreover, there is some indication that such a reduction may be impossible [FF93, BT03, AGGM06].

Given this seeming impossibility of a "worst-to-average-case" reduction, one is interested in an "average-to-average-case" reduction that would yield a function of greater average-case hardness from the one of mild average-case hardness. Such reductions have been a focus of much attention in recent research in complexity theory, especially because of their role in derandomization; see, e.g., [Yao82, NW94, BFNW93, IW97, MV99, STV01]. In these derandomization papers, the focus was on amplifying average-case (or even worst-case) hardness of Boolean function families computable in *exponential* time, where hardness is with respect to *nonuniform* algorithms (Boolean circuits). For exponential-time Boolean function families, essentially optimal hardness-amplifying reductions are known with respect to both Boolean circuits [STV01] and BPP algorithms [TV02]. The hardness amplification problem for languages in NP with respect to Boolean circuits was first considered in [O'D04], and completely solved in [HVV04].

Trevisan [Tre03, Tre05] considers hardness amplification for NP languages where average-case hardness is with respect to *uniform* randomized polynomial-time algorithms. His main result shows that if NP contains a language L such that every BPP algorithm errs on at least a $1/n^c$ fraction of n-bit inputs for some constant c, then NP contains another language L' such that every BPP algorithm errs on at least a $1/2 - 1/\log^\alpha n$ fraction of n-bit inputs for some constant $0 < \alpha < 1$. This falls short of achieving the hardness $1/2 - 1/\text{poly}(n)$ for L' with respect to BPP algorithms, which would match the parameters known for the nonuniform setting of Boolean circuits [HVV04].

Trevisan's proof proceeds by carefully analyzing the use of advice in O'Donnell's proof of amplification [O'D04] in the non-uniform setting. A bottleneck here is the use of Impagliazzo's hard-core lemma [Imp95], the known proofs of which are highly non-uniform.

In the present paper, we suggest a different, coding-theoretic approach to improving Trevisan's result. Our main contribution is conceptual: we reduce the problem of hardness amplification in NP to that of constructing efficiently list-decodable *monotone* binary error-correcting codes (possibly of exponential rate). In terms of actual results, we only match Trevisan's result, but we show that any decoding procedure that is better than exhaustive decoding for our codes would improve on Trevisan's result. We give more details next.

1.1 Hardness amplification vs. error correction

Hardness amplification of Boolean functions is closely related to the problem of constructing certain error-correcting codes over the binary alphabet. In the hardness amplification problem, given a Boolean function $f : \{0,1\}^n \to \{0,1\}$ that is somewhat hard to compute, we want to construct a new Boolean function $g : \{0,1\}^m \to \{0,1\}$ that is even harder to compute. The task of producing g from f can be viewed as that of *encoding* the truth table of f (a binary string of length 2^n) into the truth table of g (of length 2^m). The fact that g must be harder than f means that if there is an efficient algorithm that computes g on a certain fraction of inputs, then there is another efficient algorithm that computes f on an even bigger fraction of inputs. Usually, hardness amplification has a constructive proof which shows how an algorithm for f can be obtained from an algorithm for g. This can be viewed as *decoding* the message f, given a somewhat corrupted codeword g.

For the error-correcting code to be useful in the setting of hardness amplification, it must have the following properties: (i) [encoding complexity] computing g at a given input $x \in \{0,1\}^m$ should be done within the same complexity class as computing f, and (ii) [decoding complexity] the algorithm for computing f that is obtained from the algorithm for g should be within the same complexity class as the algorithm for g.

For example, if f is a function from an NP function family, we want its "encoded version" g to be a function from an NP function family. Also, if the algorithm for g is a BPP algorithm, we want the "decoding" algorithm for f to be a BPP algorithm.

The first condition is satisfied if the function $g(y)$, for $y \in \{0,1\}^m$, is a polynomial-time computable *monotone* Boolean function of at most $\ell = \text{poly}(n)$ values $f(x_1), \ldots, f(x_\ell)$ for some $x_i \in \{0,1\}^n$ computed from y in deterministic polynomial time. That is, each bit at position y of the codeword g is a monotone function of a small number of bits in the message f (when g and f are viewed as binary strings of sizes 2^m and 2^n, respectively). We shall call such codes *locally encodable monotone* codes.

The second condition is satisfied if there is a BPP oracle algorithm that computes f, given oracle access to a function $\tilde{g}$ which is a somewhat corrupted version of the function g. That is, the error-correcting code has an efficient *local decoding* algorithm. The decoding is local in the sense that to compute the value of $f(x)$ at a given position $x \in \{0,1\}^n$, the algorithm needs to know the values of $\tilde{g}$ at only a few, usually at most $\text{poly}(n)$, positions.

Thus, for uniform hardness amplification in NP, it suffices to construct locally encodable/decodable monotone error-correcting codes. There is another complication. Unique decoding is possible only if the fraction of corrupted bits in a codeword is less than $1/4$. This means that using uniquely decodable codes will give hardness amplification to a constant $\gamma < 1/4$. If we want to construct a function g of hardness close to $1/2$ (ideally, $1/2 - 1/\text{poly}(n)$), we need to use *list-decodable* codes. The decoding algorithms for such codes can tolerate close to a $1/2$ fraction of errors, and will provide a list of candidate messages that contains the message f of interest to us. If this list is small (polynomial in n), we get a small number of candidate algorithms for f, one of which is correct.

Choosing the correct algorithm for f in case f is an NP function is relatively easy thanks to the "search-to-decision" reductions for problems in NP. Namely, it is well known that an efficient decision algorithm for SAT can be used to obtain an efficient algorithm for Search-SAT: given a SAT formula, find a satisfying assignment if one exists. For simplicity, imagine that f is actually SAT. Given a small number of candidate algorithms for f, we can obtain a small number of algorithms for Search-SAT, one of which is correct. Then we simply run all of these algorithms on a given input formula. If any of them finds a satisfying assignment, we accept; otherwise, we reject. It is easy to see that the obtained (single) algorithm correctly computes f.

Finally, we should point out that the decoding algorithm for locally encodable error-correcting codes cannot compute the original message f exactly. The local encodability restriction implies that any two messages f and f' that differ in few positions will have encodings g and g' that differ in few positions, i.e., the code does not have very good distance. So the best we can hope for is a decoding algorithm that computes f *approximately*, say for at least a $1 - 1/n^c$ fraction of inputs. For a list-decodable code, we get a list of algorithms, one of which approximately computes f. We call such codes *approximately list-decodable*. Converting a list of algorithms into a single algorithm approximately computing f is also possible in this case, thanks to the average-case version of a "search-to-decision" reduction for NP [BDCGL92].

To summarize, for uniform hardness amplification in NP, we need to construct approximately list-decodable monotone codes that are locally encodable/decodable, with small list size.

1.2 Previous results

In the language of codes defined above, the hardness amplification results of Trevisan [Tre03, Tre05] can be viewed as a construction of locally encodable and approximately locally list-decodable monotone codes with polynomial list size that can correct up to a $1/2 - 1/\log^\alpha n$ fraction of errors,

for some constant $0 < \alpha < 1$. The fraction of errors $1/2 - 1/\log^\alpha n$ is chosen so that the list size of the code is polynomial in n; to correct a $1/2 - \epsilon$ fraction of errors, the constructed code uses the list size $2^{\mathrm{poly}(1/\epsilon)}$.

1.3   Our results

Our starting point is the following observation. In the nonmonotone case, locally encodable/decodable codes can be obtained from Yao's XOR Lemma [Yao82]; see [Imp02, Tre03]. Yao's XOR Lemma says, roughly, that if a Boolean function $f : \{0,1\}^n \to \{0,1\}$ is δ-hard for circuits of size s (i.e., every circuit incorrectly computes f on at least a δ fraction of inputs), then the following function $f^{\oplus k}(x_1, \ldots, x_k) = \oplus_{i=1}^k f(x_i)$ is $(1/2 - \epsilon)$-hard for ϵ approximately equal to $2^{-\delta k}$ and somewhat smaller circuit size $s' = s \cdot \mathrm{poly}(\delta, \epsilon)$. In coding-theoretic terms, this can be viewed as encoding the 2^n-bit truth table of f with the 2^m-bit truth table of the function $g = f^{\oplus k}$, where $m = kn$.

All known proofs of Yao's XOR Lemma [Lev87, GNW95, Imp95, IW97, IJK06] give approximate local list-decoding algorithms for this XOR code, usually with a list size that is exponential in $1/\epsilon$; the correct list size should be polynomial in $1/\epsilon$ (see, e.g., [IJK06]). A recent proof [IJK06] of Yao's XOR Lemma achieves the correct list size $\mathrm{poly}(1/\epsilon)$ for the case of "large" $\epsilon = \Omega(\mathrm{poly}(1/n))$. Since in the case of hardness amplification in NP the goal is to construct a Boolean function of hardness $1/2 - \epsilon$ for $\epsilon = 1/\mathrm{poly}(n)$, this "largeness" requirement on ϵ in [IJK06] is not a limitation for us.

The construction in [IJK06] proceeds in two stages. It first gives an approximate local list-decoding algorithm with small list size for a direct-product code, where the truth table of $f : \{0,1\}^n \to \{0,1\}$ is encoded with the truth table of the following non-Boolean function $f^k(x_1, \ldots, x_k) = (f(x_1), \ldots, f(x_k))$. Note that this direct-product code maps binary messages into codewords of size 2^{kn} over the alphabet $\{0,1\}^k$. The new function f^k is computationally harder than f, but is not Boolean. To convert it to a Boolean function, the second stage of the construction uses the standard code concatenation: the direct-product code is concatenated with the Hadamard code, for which a list-decoding algorithm with optimal parameters is already known [GL89]. The concatenation yields a locally encodable and approximately locally list-decodable binary code with good parameters.

To obtain the same kind of codes in the monotone case, we follow a similar two-stage construction. First, we use the same direct-product codes of [IJK06]. This gives us codewords over a non-binary alphabet. To get binary codes, we need to concatenate these direct-product codes with some *monotone* code. Code concatenation means that every symbol (from the nonbinary alphabet $\{0,1\}^k$) of the direct-product code is encoded with a binary code. The size of the alphabet for the direct-product code is determined by the amount of hardness we want to achieve in the first stage. Choosing $k = \mathrm{poly}(n)$ guarantees sufficient hardness amplification for our purposes. Note that the binary code we use in concatenation needs to encode messages of size $k = \mathrm{poly}(n)$. Since the message size is so small, we can afford to use a binary code that is *not* locally encodable; the resulting concatenated code will still be locally encodable thanks to the local encodability of the direct-product code. Similarly for decodability, we do not need our binary code to be *locally* decodable; it is sufficient for us if the decoding algorithm for the code runs in time polynomial in the message size, which is $\mathrm{poly}(n)$ in our case. Also, as in the non-monotone case, we can allow the binary code to have *exponential rate*. That is, the code can map k-bit messages into $2^{\mathrm{poly}(k)}$-bit codewords, as long as the decoding algorithm, given oracle access to a corrupted version of the codeword, runs in time polynomial in the message size, i.e., $\mathrm{poly}(k)$.

The conclusion is that we need monotone approximately list-decodable codes, with possibly exponential rate, that are not necessarily *locally* encodable/decodable.

This leads us to the study of monotone codes, which are interesting in their own right. We first investigate what kinds of parameters are achievable with monotone codes, non-constructively. We prove that monotone codes in general have very poor distance and list-decodability, however these parameters improve greatly if we restrict our attention to balanced or almost balanced messages (i.e., messages with about an equal number of 0s and 1s). Getting monotone codes for almost balanced messages is good enough for our goal of uniform hardness amplification in NP, as will be explained later in the paper.

Our main technical contribution is an efficient unique decoding algorithm for a natural monotone code based on the Majority function. Plugging this code into our two-stage construction outlined above yields a uniform hardness amplification result for NP, taking a function $f \in$ NP of average-case hardness $1/n^c$ to a function $g \in$ NP of constant average-case hardness.

For the case of list-decodable monotone codes, we present a construction based on the iterated Majority function; argue information-theoretically that it has good parameters, using the noise stability estimates from [O'D04]; and show that even a brute-force approximate list-decoding algorithm for this code already yields hardness amplification in NP from constant average-case hardness to hardness $1/2 - \log^\alpha n$, for some $\alpha < 1$.

These two constructions give an alternative proof of the hardness amplification result due to Trevisan [Tre03, Tre05]. We also show that to get hardness amplification with better parameters, it suffices to present any nontrivial (i.e., non-brute-force) approximate list-decoding algorithm for some monotone code. Thus we reduce the problem of hardness amplification in NP to the question of constructing an efficiently approximately list-decodable monotone code.

The remainder of the paper. In section 2, we give a few preliminary definitions. In section 3, we define monotone codes, and show what we can and cannot hope to achieve with them. We also introduce a generic construction of a monotone code that we will use throughout. In section 4, we show how to achieve uniform hardness amplification in NP up to constant hardness via monotone codes. Finally, in section 5, we show how to continue amplifying to hardness $\frac{1}{2} - \epsilon$, where $\epsilon = 1/\log^\alpha n$, also using monotone codes.

2 Preliminaries

2.1 Languages and average-case hardness

We assume a basic familiarity with complexity classes. The reader is referred to the Complexity Zoo [TCZ] for basic definitions and results. We refer interchangeably to Boolean functions and languages, depending on context.

Given a language L, L_n is the set of strings of length n in L. We will use the notation $L(x) = 1$ to mean $x \in L$, and $L(x) = 0$ to mean $x \notin L$. The language L defines a Boolean function family $\{f_n\}_{n \geqslant 0}$ where f_n is the characteristic function of L_n. Similarly, a Boolean function family defines a corresponding language. To simplify the notation, we use $f : \{0,1\}^n \to \{0,1\}$ to mean a function family $\{f_n\}_{n \geqslant 0}$ where $f_n : \{0,1\}^n \to \{0,1\}$.

By a BPP *algorithm* we mean a randomized Turing machine that accepts every input with probability at least $2/3$ or at most $1/3$. For a BPP algorithm A, we denote by $L(A)$ the language decided by A, i.e., $L(A) = \{x \mid \mathbf{Pr}[A(x) \text{ accepts}] \geqslant 2/3\}$. For $\delta = \delta(n)$, we say that a language L is δ-*hard* for BPP if, for every BPP algorithm A, there are infinitely many input lengths n such that $\mathbf{Pr}_{x \in \{0,1\}^n}[L(x) \neq (L(A))(x)] \geqslant \delta(n)$, where the probability is taken over the uniform distribution on $\{0,1\}^n$.

We define what it means for a language to be computable in BPP with advice. The model of advice we use is the model defined by Trevisan and Vadhan [TV02], where the advice depends on the randomness. We say $L \in \mathsf{BPP}//a(n)$, if there is a probabilistic machine M such that for any random string r used by the machine on inputs of length n, there is a string $f(r), |f(r)| = O(a(n))$ given as advice on that random string, satisfying the following. For each x of length n, if $x \in L$, then M accepts with probability at least $2/3$ over r given $f(r)$; when $x \notin L$, M rejects with probability at least $2/3$ over r given $f(r)$.

2.2 Error-correcting codes

For strings $x, y \in \{0,1\}^n$, $\Delta(x, y)$ is the relative Hamming distance between x and y, i.e., the fraction of positions on which x and y differ. A function $C : \{0,1\}^N \rightarrow \{0,1\}^M$ is a binary code with *relative distance d* if for any distinct $x, y \in \Sigma^N$, $\Delta(C(x), C(y)) \geqslant d$. Such a code will be denoted as a $[M, N, d]$-code. The *rate* of an $[M, N, d]$-code is defined as N/M. A code C is (ρ, L)-*list-decodable* if for any $w \in \{0,1\}^M$, $|\{x \mid x \in \{0,1\}^N, \Delta(C(x), w) \leqslant 1/2 - \rho\}| \leqslant L$.

On occasion, we abuse terminology by dropping the parameter d. We say a code is *balanced* if the encoding of any balanced message is balanced. We are primarily interested in families of codes $\{C_N\}_{N=1}^{\infty}$, where each $C_N : \{0,1\}^N \rightarrow \{0,1\}^{M(N)}$. We often abuse notation and use the term "code" to refer to a family of codes.

3 Monotone codes

In this section, we define and prove some basic existential results about monotone codes.

Monotone codes have the property that each codeword bit is a monotone function of the message. That is, a code $C : \{0,1\}^N \rightarrow \{0,1\}^M$ is *monotone* if for every $1 \leqslant i \leqslant M$, the function $(C(x))_i$ is a monotone Boolean function of the message $x \in \{0,1\}^N$.

For monotone codes, the notion of approximate list-decodability is of interest. Since, as we show in the next subsection, monotone codes have very poor parameters in general, we give the following definition for the case of almost balanced messages only.

By β-*balanced* message we mean a binary string of bias at most β, where the bias $bias(x)$ of a binary string x is defined as the absolute value of the difference between the fraction of 1s and the fraction of 0s in the string. A 0-balanced string will be called balanced. An $[M, N]$ code C is α-*approximately (ρ, L)-list decodable* on β-balanced messages if for each $y \in \{0,1\}^M$, there is a set $S_y \subseteq \{0,1\}^N, |S_y| \leqslant L$ such that for each $x \in \{0,1\}^N$ with $\Delta(C(x), y) \leqslant 1/2 - \rho$ and $bias(x) \leqslant \beta$, there is a string $x' \in S_y$ with $\Delta(x, x') \leqslant \alpha$. If $\alpha = 0$, we will drop "0-approximately" and simply say that such a code is list-decodable. If $L = 1$, we say that C is *approximately uniquely decodable*.

3.1 Limitations and nonconstructive existence results

It is natural to ask what kinds of parameters monotone codes can achieve, in terms of minimum distance and list-decodability. Here we observe that monotone codes have rather poor parameters in general. However the situation is much improved if we restrict our attention to almost balanced messages.

Theorem 1. *Let $C : \{0,1\}^N \rightarrow \{0,1\}^M$ be a monotone code. Then C has relative distance $\leqslant 1/N$.*

Proof. For each $i, 0 \leqslant i \leqslant N$, let X_i denote the message consisting of i consecutive 1's followed by $N-i$ 0's. Let $x \prec y$ for strings x, y of the same length denote that each bit of x is less than or equal to

the corresponding bit of y. Since code C is monotone, we have that $C(X_0) \prec C(X_1) \ldots \prec C(X_N)$. By the pigeonhole principle, this implies that there is an index i, $0 \leqslant i \leqslant N-1$ such that the relative distance between C_i and C_{i+1} is at most $1/N$. $\qquad\square$

The above proof also gives that, if we restrict our attention to messages of bias at most β, the relative distance is at most $1/\beta N$. We next show that when restricted to balanced messages, monotone codes can have good relative distance. Our result is probabilistic and non-constructive. The basic idea is to pick codewords corresponding to balanced messages at random, and then extend the code to a monotone code over the entire message space.

Theorem 2. *There are constants $\alpha > 1$ and $\beta < 1$ for which the following holds. There is an $[\alpha N, N]$ monotone code C such that for any two distinct balanced messages x and y, the relative distance between $C(x)$ and $C(y)$ is at least β.*

Proof. We first define the code on balanced messages and then extend it to unbalanced messages. Pick 2^N strings of length αN at random, for α to be fixed later. For some constant β and α large enough, by Chernoff bounds there is a way of picking those strings so that no two strings have relative distance less than βN. Order these strings arbitrarily, and let the ith string be the codeword for message i. This defines the code on all balanced messages. Next we need to extend the code to unbalanced messages so that the code is monotone. This can be done trivially by mapping all messages with weight greater than $1/2$ to the all ones string, and all messages with weight less than $1/2$ to the all zeroes string. $\qquad\square$

Theorem 2 also gives a code of rate that is optimal up to a constant. Analogous results can also be shown in the list-decoding setting, for the general case in which we wish to decode *exactly* and on messages with arbitrary bias.

Theorem 3. *Let C be an $[M, N]$ monotone code. Then C is not (ϵ, L) list decodable when $L < 2^{N(1/2-\epsilon)}$ and $1/(1/2 - \epsilon)$ is an integer. In particular, C is not (ϵ, L) list decodable when $\epsilon \leqslant 1/6$ and $L = o(2^{N/3})$.*

Proof. Define X_i for $0 \leqslant i \leqslant N$ as in the proof of Theorem 1. Let t be an integer such that $1/t = 1/2 - \epsilon$. Using monotonicity of C, define the same precedence ordering $\prec$ as in the proof of Theorem 1, such that for each $0 \leqslant i \leqslant N-1$, $C(X_i) \prec C(X_{i+1})$. By the pigeonhole principle, there is some X_i such that the Hamming ball of radius M/t centered on $C(X_i)$ contains the codewords $C(X_{i+1}), \ldots, C(X_{i+N/t})$. But this implies that that Hamming ball contains at least $2^{N/t}$ codewords, since, for any $X \in \{0,1\}^N$ such that $X_i \prec X \prec X_{i+N/t}$, we have $C(X_i) \prec C(X) \prec C(X_{i+N/t})$. There are $2^{N/t}$ such X's. $\qquad\square$

The same proof yields an impossibility result for list decoding on β-biased messages, at the cost of a factor of β in the lower bound on L. Just as in the unique decoding case, good list decoding is possible when the bias is 0. The proof is entirely analogous to the proof of Theorem 2.

Theorem 4. *There are constants $\gamma > 1$ and $\delta < 1$ such that there is a $[\gamma N, N]$ monotone code which is $(\epsilon, \text{poly}(1/\epsilon))$ list decodable on balanced messages, for any $\epsilon > 1/(\delta N)$.*

Theorem 2 and Theorem 4 are existential results. By embedding explicit binary codes in the middle of the Hamming cube, we obtain constructive versions of these results. However, our setting of hardness amplification seems to require either codes that have a more local encoding

procedure (running in time polylog(N)), or codes that work for *almost*-balanced messages (see Subsection 3.3). We next discuss a very natural, generic family of monotone codes that give rise to both approximately uniquely decodable and list decodable codes.

3.2   Generic constructions

The following generic constructions of monotone codes were inspired by the Hadamard code, where we replace the XOR function with a monotone function.

Definition 5. *Given a Boolean function $f : [N] \to \{0,1\}$, the k-wise direct product of f is a mapping $f^{(k)} : [N]^k \to \{0,1\}^k$, such that for any $x_1, x_2 \ldots x_k \in [N]$, $f^{(k)}(x_1, x_2 \ldots x_k) = f(x_1)f(x_2)\ldots f(x_k)$.*

We next define, for each monotone function g on k bits, a monotone code for which messages are encoded by composing the k-wise direct product of the message with g.

Definition 6. *For any string $x \in \{0,1\}^N$, let $fn(x) : [N] \to \{0,1\}$ be the Boolean function corresponding to x. For any monotone function g on k bits, we define the monotone code $Code_g^{N,k} : \{0,1\}^N \to \{0,1\}^{N^k}$ as follows: for any $x \in \{0,1\}^N$, $fn(Code_g^{N,k}(x)) = g \circ (fn(x))^{(k)}$.*

In the future, we often refer interchangeably to strings and functions, where the string corresponding to a function is the truth table of the function. It will be clear from the context whether we are dealing with strings or with functions. Also, when N is implicit, we omit it and refer simply to $Code_g^r$.

Our next result characterizes the list size for approximate list-decodability of the codes $Code_g^{N,k}$ precisely in terms of the *noise stability* of the monotone function g. Following [O'D04], given $x \in \{0,1\}^k$, let $Noise_\delta(x)$ be a random variable in $\{0,1\}^k$ given by flipping each bit of x independently with probability δ. We then define the noise stability of g to be

$$NoiseStab_\delta(g) \equiv \Pr_x[g(x) = g(Noise_\delta(x))],$$

and define the quantity

$$NS_\delta^*(g) \equiv 2NoiseStab_\delta(g) - 1.$$

Claim 7. *$Code_g^{N,k}$ is δ-approximately (ϵ, t) list decodable on balanced messages for*

$$t \leq \frac{1}{4\epsilon^2 - \frac{1}{2}NS_\delta^*(g)}.$$

Proof. We begin by following the proof of a similar lemma in the full version of [IJK06] given for the case of g being the XOR function. For a string $y \in \{0,1\}^N$, let $code_y$ denote the encoding of y using the code $Code_g^{N,k}$. Given a received word B, let $m_1, m_2 \ldots m_t \in \{0,1\}^N$ be balanced strings such that $\Delta(m_i, m_j) \geqslant \delta$ for each $i \neq j$, and $\Delta(code_{m_i}, B) \leqslant 1/2 - \epsilon$ for each i. Any upper bound we show on t translates to the desired upper bound on list size for δ-approximate list decoding on balanced messages.

Given $s \in [N]^k$, let

$$\epsilon_s = \Pr_{i \in [t]}[code_{m_i}(s) = B(s)] - \Pr_{i \in [t]}[code_{m_i}(s) \neq B(s)] = \frac{1}{t}\sum_{i \in [t]}(-1)^{code_{m_i}(s) \oplus B(s)}.$$

8

By assumption, $\text{Exp}_{s\in[N]^k}[\epsilon_s] = 2\epsilon$, so $4\epsilon^2 \leq (\text{Exp}_s[\epsilon_s])^2$, which is at most $\text{Exp}_s[\epsilon_s^2]$ by Jensen's Inequality. Now

$$
\begin{aligned}
\text{Exp}_s[\epsilon_s^2] &= \text{Exp}_s[\tfrac{1}{t^2}\sum_{i,j}(-1)^{code_{m_i}(s)\oplus code_{m_j}(s)}] \\
&= \tfrac{1}{t^2}\sum_{i,j}\text{Exp}_s[(-1)^{code_{m_i}(s)\oplus code_{m_j}(s)}] \\
&= \tfrac{1}{t} + \tfrac{1}{t^2}\sum_{i\neq j}\text{Exp}_s[(-1)^{code_{m_i}(s)\oplus code_{m_j}(s)}].
\end{aligned}
$$

For any balanced message $msg \in \{0,1\}^N$ and random $s = (i_1,\ldots,i_k) \in [N]^k$, $(msg(i_1),\ldots,msg(i_k))$ is a perfectly random string from $\{0,1\}^k$. Hence, if m_i and m_j differ in exactly δN positions, then, for random s, $code_{m_i}(s)$ is distributed as $g(x)$ for $x \sim_U \{0,1\}^k$ and $code_{m_j}(s)$ is distributed as $g(Noise_\delta(x))$. In this case $\text{Exp}_s[(-1)^{code_{m_i}(s)\oplus code_{m_j}(s)}] = NS_\delta^*(g)$. In general, assuming $NS_\delta^*(g)$ is nonincreasing in δ, then

$$
\text{Exp}_s[(-1)^{code_{m_i}(s)\oplus code_{m_j}(s)}] \leq NS_\delta^*(g).
$$

Finally, we have $4\epsilon^2 \leq (1/t) + (t(t-1)/2t^2)NS_\delta^*(g) \leq (1/t) + (1/2)NS_\delta^*(g)$, and so $t \leq (4\epsilon^2 - (1/2)NS_\delta^*(g))^{-1}$, as required. $\qquad\square$

In order to apply Claim 7, we need to use a monotone function g with good noise stability. Following O'Donnell [O'D04], we use the REC-MAJ$_k$ function which is defined on input length k a power of 3. The function is defined recursively as follows: For $k = 3$, REC-MAJ $\equiv$ MAJ. For $k = 3^t, t > 1$, given input X of length k, let X_1, X_2, X_3 be the first, middle and last third of X, each of length $k/3$. Then REC-MAJ$_k(X) \equiv$ MAJ(REC-MAJ$_{k/3}(X_1)$, REC-MAJ$_{k/3}(X_2)$, REC-MAJ$_{k/3}(X_3)$).

Lemma 8 ([O'D04]). *Let k be a power of 3, and $\delta \geqslant 1/1.1^{\log_3(k)}$. Then $NoiseStab_\delta^*(\text{REC-MAJ}_k) \leqslant 1/(k^{0.15}\delta^{1.1})$.*

Using Claim 7 and Lemma 8, we derive the following result showing good approximate list-decoding for a natural code.

Theorem 9. *Let $g = \text{REC-MAJ}_k$ for k a power of 3. Then there is a constant $\gamma > 0$ such that $Code_g^{N,k}$ is δ-approximately (ϵ,t) list decodable on balanced messages for $\delta = \Omega(\text{poly}(1/k))$, $\epsilon > 1/k^\gamma$ and $t = \text{poly}(1/\epsilon, 1/\delta)$.*

These codes have remarkably different properties depending on the relationship between N and k. For example, if k is polynomially bigger than N, the distance of the code approaches $1/2$ and it becomes *exactly* uniquely and list decodable. On the other hand, if N is polynomially big in k, the code has a worse list decoding radius, but becomes robust to almost-balanced messages. We will use this latter fact in Section 5. Finally, we note that if g is MAJ instead of REC-MAJ, then we can uniquely decode almost-balanced messages up to small distance even when $N = k$. This fact will be used in Section 4.

We should point out that the inequality of Claim 7 holds only when $4\epsilon^2 \geqslant NS_\delta^*(g)/2$, i.e., when $\epsilon \geqslant \sqrt{NS_\delta^*(g)/8}$. O'Donnell [O'D04], using a result of Kahn, Kalai and Linial, observes that for any monotone g and constant δ, $NS_\delta^*(g) \geqslant \Omega((\log^2 k)/k)$. This implies that Claim 7 is useless for $\epsilon = O(1/\sqrt{k})$. On the other hand, we would like to prove polynomial list size for approximate decoding from error rate as high as $1/2 - 1/n^d$ for any constant $d > 0$, where $n = k\log(N)$ is the input size of the amplified function.

To close this gap, one can consider a "derandomized" version of the direct product construction, as in [IW97, HVV04]. The basic intuition is as follows. The aforementioned limitation arises because the basic direct product construction involves taking k *independent* instances of the original function

and applying the monotone amplifier to the values of the original function on these instances. Suppose we were able to generate k instances "pseudo-randomly", i.e., using $\ll k$ random bits, while still preserving the noise-stability property to within a small additive error. Then the input size of our new amplified function would be much less than $k \log(N)$, while the error up to which the list-decodability property holds could potentially still be close to $O(1/\sqrt{k})$. As a consequence, the tolerable error could be an arbitrarily small polynomial as a function of the new input size, which would allow us to circumvent the limitation.

3.3 Connection with Hardness Amplification

The primary significance of monotone codes in complexity theory is in the connection to hardness amplification within non-deterministic classes. The connection between codes and hardness amplification was first observed by Sudan, Trevisan and Vadhan [STV01] in the context of worst-case to average-case hardness amplification for EXP, and later investigated by Trevisan [Tre03] in the hope of obtaining codes with good rate and efficient decoding from known proofs of hardness amplification. In this section, we clarify this connection in the context of hardness amplification within NP.

Theorem 10. *Assume there is a balanced $[M, N]$ monotone code C encodable in time $\mathrm{polylog}(N)$, where $M = O(2^{\mathrm{polylog}(N)})$, such that C is α-approximately uniquely decodable in time $\mathrm{polylog}(M)$ up to distance d on balanced messages. Then the following hardness amplification result holds: If there is a balanced function $f \in$ NP such that f is α-hard for BPP, then there is a balanced function $f' \in$ NP such that f' is d-hard for BPP. If the hardness assumption holds against probabilistic algorithms taking $b(n)$ bits of advice, then the derived hardness holds against probabilistic algorithms taking $b(n^{O(1)})$ bits of advice, where $n = \log(N)$.*

Proof. Let f be a balanced Boolean function in NP such that f is α-hard for BPP. Define $f' = C(f)$, i.e., the truth table of f' is the encoding of the truth table of f using the code C (we assume wlog that M is a power of 2 when N is a power of 2). Since C is a balanced code, the fact that f is balanced implies f' is balanced. Also, since C is a monotone code encodable in time $\mathrm{poly}(n)$, $f \in$ NP implies $f' \in$ NP. To see this, note that to compute any entry of the truth table of f', we only need to make $\mathrm{poly}(n)$ queries to the truth table of f, where n is the input size of f. To answer each of these queries, we run the NP algorithm for f, answering "yes" for the query iff the sequence of guesses for that query leads to acceptance by the NP algorithm. Of course, there will be sequences of guesses on which we do not obtain the correct answer for the query. The crucial fact is that since C is monotone, no sequence of guesses will lead to computing a 0-entry of f' as 1, while on the other hand, for each 1-entry of f', there will be some sequence of guesses which lead to computing the entry as a 1-entry. This fact, combined with the fact that we spend only polynomial time making and answering queries, implies that $f' \in$ NP.

Next we use the additional properties of C to show that f' has amplified hardness. Suppose, for the purpose of contradiction, that there is a BPP algorithm computing f' on a $1 - d$ fraction of inputs. Let g' be the function computed by the BPP algorithm. From the assumption that C is α-approximately uniquely decodable up to distance d via an efficient local decoding algorithm, it follows that there is a BPP algorithm computing a function g such that $\Delta(g, h) \leqslant \alpha$ for any balanced function h for which $\Delta(C(h), g') \leqslant d$. But f itself is such a balanced function, hence we get that $\Delta(f, g') \leqslant \alpha$. This implies that f is not α-hard for BPP, in contradiction to the assumption on hardness of f. $\qquad\square$

An analogous result holds for hardness amplification beyond $O(1)$-hardness, but this implies that we are in the list-decoding regime, and hence we incur a small cost in advice when we perform the amplification.

Theorem 11. *Assume there is a monotone $[M, N]$ code C encodable in time $\mathrm{polylog}(N)$, where $M = O(2^{\mathrm{polylog}(N)})$, such that C is α-approximately (ρ, L) list decodable on balanced messages in time $\mathrm{polylog}(M)$. Then the following hardness amplification holds: If there is a balanced function $f \in \mathsf{NP}$ such that f is α-hard for $\mathsf{BPP}//(b(n) + O(\log(L)))$, then there is a function $f' \in \mathsf{NP}$ such that f' is $(1/2 - \rho)$-hard for $\mathsf{BPP}//b(n^{O(1)})$.*

We emphasize that our general approach will be to start by applying a k-wise direct product, for k at most $polylog(N)$, to the truth table of the function whose hardness we want to amplify. Then, each symbol of the resulting codeword will be encoded by a binary monotone code. Because of this, the binary monotone code need not be locally encodable and can have exponentially low rate and still the concatenation of the two codes will satisfy the conditions of Theorems 10 and 11. In return, however, the binary monotone codes must work on almost-balanced messages. For instantiations of this approach, see Sections 4 and 5.

It is natural to ask if we can relate monotone codes to hardness amplification in NP starting from *any* hard function, rather than one that is balanced. Using ideas of O'Donnell [O'D04] and Trevisan [Tre03], we can achieve this if the decoding algorithm works for almost-balanced messages.

Theorem 12. *Assume there is a monotone $[M, N]$ code C encodable in time $\mathrm{polylog}(N)$, where $M = O(2^{\mathrm{polylog}(N)})$, such that C is $1/\mathrm{poly}(n)$-approximately $(\rho, \mathrm{poly}(n))$ list decodable on $1/\mathrm{poly}(n)$-almost balanced messages. Then the following hardness amplification holds: If NP is $1/\mathrm{poly}(n)$-hard for BPP, then NP is $(1/2 - \rho)$-hard for BPP.*

Proof. Trevisan [Tre05] showed that if NP is $1/\mathrm{poly}(n)$-hard for BPP, then NP is $1/\mathrm{poly}(n)$-hard for $\mathsf{BPP}//\log(n)$. O'Donnell [O'D04] showed that if NP is $1/\mathrm{poly}(n)$-hard for $\mathsf{BPP}//\log(n)$, then there is a $1/\mathrm{poly}(n)$-almost balanced function f in NP which is $1/\mathrm{poly}(n)$-hard for $\mathsf{BPP}//\log(n)$. Now applying the code C, we get the required amplification from Theorem 11, with the minor difference that we need to compute the almost-balanced function f rather than a balanced function as was the case earlier. However, this is taken care of by the fact that the decoding of C works on almost-balanced messages. $\square$

4 Uniform hardness amplification to a constant

In this section, we give a very elementary proof of the following amplification result. A qualitatively identical result has been proven by [Tre05].

Theorem 13. *There is an absolute constant $0 < c < 1/4$ such that, for any polynomial $s(n)$, if there is a language $L \in \mathsf{NP}$ that is $1/s(n)$-hard for BPP, then there is a language $L' \in \mathsf{NP}$ that is c-hard for BPP.*

We will actually show that given a *balanced* function in NP that is hard, we can obtain a harder balanced function in NP. However, it will be clear that the proof works for functions that are $1/poly(n)$-almost balanced, so we can conclude Theorem 13 using the ideas of Theorem 12.

Theorem 14. *There is an absolute constant $0 < c < 1/4$ such that, for any polynomial $s(n)$, if there is a balanced language $L \in \mathsf{NP}$ that is $1/s(n)$-hard for $\mathsf{BPP}//\log n$, then there is a balanced language $L' \in \mathsf{NP}$ that is c-hard for $\mathsf{BPP}//\log n$.*

Throughout the proof, for simplicity of exposition, we will ignore the advice. It is implicitly included throughout.

The construction of f' proceeds by iterating two steps. Step one applies a direct product to f and step two encodes each tuple of the direct product with a monotone error correcting code (this can also be viewed as the concatenation of two codes). The concatenation of the encodings of the tuples forms the truth table of a new function which is somewhat harder than the original and is still in NP. We can continue iterating this process until we finally arrive at a function f' that is c-hard. Since each function along the way is at least δ-hard, where $\delta = \Omega(1/\mathrm{poly}(n))$, the direct products we take create tuples of size at most $\mathrm{poly}(n)$. The nice thing about this approach is that we can afford to apply a monotone code of exponential rate to these tuples and not blow up the input size of the function by more than a polynomial factor. Also, the code does not need to have local-encoding to preserve the fact that the resulting function is in NP. We iterate this process a logarithmic number of times, but the size of the direct product gets smaller as the function gets harder, so that the eventual function remains in NP.

We will prove Theorem 13 in the next subsection. Here we describe the tools we need for the proof. Our first tool is the following direct product lemma. It is a very weak one, but is completely uniform and has an elementary proof. Below, when we say that a BPP algorithm $\mathcal{A}$ ϵ-computes direct product $f^{(k)}$, we mean that for at least an ϵ fraction of k-tuples $\bar{x}$, we have $\mathcal{A}(\bar{x}) = f^{(k)}(\bar{x})$ with high probability over the internal random coins of C.

Lemma 15. *For any $\epsilon, \delta \in \Omega(1/\mathrm{poly}(n))$, let $k \geq 1/2\delta$. If there is a BPP algorithm that $(7/8 + \epsilon)$-computes $f^{(k)}$, then there is a BPP algorithm that $(1 - \delta)$-computes f.*

Proof. For $x \in \{0,1\}^n$, let B_x be the set of k-tuples in $(\{0,1\}^n)^k$ that contain x. Given an algorithm $\mathcal{A}$ for $f^{(k)}$, compute f as follows: given x, choose $\Omega(1/\epsilon^2)$ random k-tuples from B_x and run $\mathcal{A}$ on each k-tuple. Each such k-tuple gives rise to a prediction for $f(x)$. Simply take the majority value of these predictions.

If the above algorithm fails to $(1 - \delta)$-compute f with non-negligible probability, then there must be a set S of density at least δ in $\{0,1\}^n$ such that, for each $x \in S$, $\mathcal{A}$ is wrong on more than a $1/2 - \epsilon$ fraction of B_x. Let $\delta' = 1/2k \leq \delta$ and let $S' \subset S$ be an arbitrary subset of density δ' in $(\{0,1\}^n)^k$. Let $T' = \cup_{x \in S'}\{y \in B_x \mid \mathcal{A}$ is wrong on $y\}$. We will argue that T' has density more than $1/8 - \epsilon$, contradicting the assumption about $\mathcal{A}$.

Let $N = 2^n$. First of all, for every x, the density of B_x is $1 - (1 - 1/N)^k \geq k/N - \binom{k}{2}/N^2$. For any distinct x and x', the density of $B_x \cap B_{x'}$ is at most k^2/N^2. By inclusion-exclusion, T' has density at least $(1/2 - \epsilon)\,\delta'N|B_x| - \binom{\delta'N}{2}|B_x \cap B_{x'}| \geq (1/2 - \epsilon)\,(1/2 - o(\epsilon)) - 1/8 > 1/8 - \epsilon$. $\qquad\square$

Our second tool is the binary monotone code we use to encode the symbols of the direct product function. Let MAJ denote the majority function. We show that $Code_{\mathrm{MAJ}}^{k,k}$ has relative distance $1/\sqrt{k}$ and a simple, efficient decoding algorithm when restricted to almost balanced messages. A similar result was discovered independently by [Aka06].

Lemma 16. *For all $d \geq 0$, there exists $a > 0$ such that the following holds for all sufficiently large odd integers k. The code $Code = Code_{\mathrm{MAJ}}^{k,k}$ is a $[k^k, k, a/\sqrt{k}]$ monotone code on $d/\sqrt{k}$-balanced messages. Moreover, it is efficiently encodable and decodable with only $\mathrm{poly}(k)$ queries to the received word.*

Proof. Let $m = m_1 \ldots m_k$ be a message of bias at most $d/\sqrt{k}$. Without loss of generality, let us assume that m contains pk ones and $(1 - p)k$ zeros, where $p \geq 1 - p$ and $p \leq 1/2 + d/\sqrt{k}$. Let $code = Code(m)$ be the encoding of m. Consider the task of decoding m_1 from a string B which

12

is a corrupted version of *code*. Let s be a random k-tuple of $[k]$ that contains 1. Let s' be the $(k-1)$-tuple that results from removing any occurrence of 1 from s. Then s' is, of course, a random $(k-1)$-tuple. Let $m(s')$ denote the $k-1$-tuple formed by indexing m at each of the indices in s'. We will argue that $m(s')$ will be balanced with reasonable probability, so that the value of m_1 will determine the value of $\text{MAJ}(m(s))$; Majority is ideal in this setting because each variable has relatively high influence. Hence, by querying $\text{MAJ}(m(s))$ for many random k-tuples s containing 1, the answers should look different depending on whether $m_1 = 1$ or $m_1 = 0$. In fact, the difference will be so pronounced that the procedure is robust to errors in the answers to the queries.

We will argue that Algorithm 1 below decodes m, with poly(k) queries to B.

For $i = 1$ to k
　　For $j = 1$ to k^2
　　　　Pick a random $k-1$-tuple of $[k]$, s'_{ij}, let s_{ij} be the result of inserting i in a random place in s'_{ij}
　　Let $\alpha_i = \frac{|\{j \mid B(s_{ij})=1\}|}{k^2}$
Let $\beta = \frac{1}{2}(\max_i \alpha_i + \min_i \alpha_i)$. If $\alpha_i \geq \beta$, set $m_i = 1$. Otherwise set $m_i = 0$.

Algorithm 1: Decoding $Code_{\text{MAJ}}^{k,k}$

For the analysis of this algorithm, we first show that the probability that $m(s')$ is balanced is $\Omega(1/\sqrt{k})$. Let $k' = k - 1$ and let $x = p - \frac{1}{2}$. Recalling that $\binom{k'}{k'/2} \geq \gamma 2^{k'}/\sqrt{k'}$ for sufficiently large k and some constant γ, we get that $\mathbf{Pr}[m(s') \text{ is balanced}]$ is

$$\binom{k'}{k'/2} p^{k'/2}(1-p)^{k'/2} \geq \gamma(4(p-p^2))^{k'/2}/\sqrt{k'} \geq \gamma(1-4x^2)/\sqrt{k'} \geq \gamma(1-4d^2/k)^{k'/2}/\sqrt{k'},$$

which is at least $(\gamma e^{-2d^2})/\sqrt{k'} \geq \Omega(1/\sqrt{k})$.

Let r be the probability that for s' being a random $(k-1)$-tuple of $[k]$, $m(s')$ has at least as many ones as zeros and let q be the probability that $m(s')$ has more ones than zeros. From the above calculation, we know $r - q \in \Omega(1/\sqrt{k})$. Now, for some fixed element of $[k]$, say 1, do the following experiment: pick a random k-tuple s that contains 1. If $m_1 = 0$, then $code(s)$ (the sth bit of the codeword *code*) will be 1 with probability q; if $m_1 = 1$, then $code(s)$ will be 1 with probability r. If, instead of querying $code(s)$, we query $B(s)$ for some string B of relative distance less than $(r-q)/10$ from *code*, then it is still true that $\mathbf{Pr}[B(s) = 1 \mid m_1 = 1] - \mathbf{Pr}[B(s) = 1 \mid m_1 = 0] \in \Omega(1/\sqrt{k})$, since 1 appears in at least a $(1 - 1/e)$-fraction of all k-tuples.

Thus decoding m_1 is reduced to distinguishing whether $B(s)$ is 1 for a $q' \in [q - \frac{r-q}{10(1-e^{-1})}, q + \frac{r-q}{10(1-e^{-1})}]$ fraction of random k-tuples s containing 1, or for a larger $r' \in [r - \frac{r-q}{10(1-e^{-1})}, r + \frac{r-q}{10(1-e^{-1})}]$ fraction. Of course, we do not know q and r. But, by Chernoff bounds, each α_i computed by Algorithm 1 will be either contained in $[q - (r-q)/5, q + (r-q)/5]$ (for $m_i = 0$) or $[r - (r-q)/5, r + (r-q)/5]$ (for $m_i = 1$) with very high probability. Since m is almost balanced, there will be both $\approx q$ and $\approx r$ values among the α_is. Hence, with high probability, the β computed by Algorithm 1 is between these two intervals, and so β can be used to distinguish large $\alpha_i \approx r$ from small $\alpha_i \approx q$. It follows that Algorithm 1 will compute each m_i correctly, with very high probability. $\square$

4.1　Proof of Theorem 14

We will apply the following lemma several times.

Lemma 17. *Let $\delta \leq \delta_0$ for some absolute constant δ_0, and let $\delta \in \Omega(1/\mathrm{poly}(n))$. Let $f : \{0,1\}^n \to \{0,1\}$ be a balanced function in* NP *that is δ-hard for* BPP. *Then there is a $k \in O(\frac{1}{\delta})$ and a balanced function $f' : \{0,1\}^{kn+k\log k} \to \{0,1\}$ in* NP *that is $\Omega(\sqrt{\delta})$-hard for* BPP.

Proof. The function f' is defined as follows. Let k be the smallest odd integer larger than $1/2\delta$ (for appropriate δ_0, then, $k \leq 1/\delta$); hence, $f^{(k)}$ cannot be, say, 9/10-approximated according to Lemma 15. Let f' be the function whose truth-table is the concatenation, for every k-tuple $\bar{x} = (x_1, \ldots, x_k) \in (\{0,1\}^n)^k$, of $Code(f^{(k)}(\bar{x}))$ where $Code = Code_{\mathrm{MAJ}}^{k,k}$. It is easy to check that f' is a function on $kn + k\log k$ bits; kn bits are used to index a particular k-tuple of inputs and the remaining $k\log k$ are used to index a bit the the code applied to that k-tuple. It is also easy to check that f' is in NP.

To see that f' is balanced, note that we can create a perfect matching on $(\{0,1\}^n)^k$, the set of all k-tuples, such that if $\bar{x}$ is matched to $\bar{x}'$, then $f^{(k)}(\bar{x})$ is the binary complement of $f^{(k)}(\bar{x}')$. This relies on the fact that f is balanced. Then the string $Code(f^{(k)}(\bar{x}))$ is exactly the complement of the string $Code(f^{(k)}(\bar{x}'))$ since the majority of a given bit string is the complement of the majority of the complement of the bit string.

Now to show the hardness of f'. Choose d such that a random string from $\{0,1\}^k$ has bias at most $d/\sqrt{k}$ with probability at least 19/20. Given that k is at least $1/2\delta_0$, tail bounds on the binomial distribution allow us to choose d a constant independent of k. Given a random k-tuple $\bar{x} \in (\{0,1\}^n)^k$, $f^{(k)}(\bar{x})$ is a random string in $\{0,1\}^k$ since f is balanced. Therefore, for at least 19/20 of all k-tuples $\bar{x}$, $f^{(k)}(\bar{x})$ has bias at most $d/\sqrt{k}$. If a δ' fraction of f' is corrupted, then, by Markov's inequality, at most 1/20 of all the k-tuples has more than a $20\delta'$-fraction of its encoding corrupted. Hence, at least a 9/10-fraction of all k-tuples have bias at most $d/\sqrt{k}$ and have at most a $20\delta'$-fraction of their encoding corrupted. By Lemma 16, there is a constant a depending on d, but not on k, such that if $20\delta' \leq a/\sqrt{k}$, then we can locally decode all of these k-tuples. Hence, there is a BPP algorithm which 9/10-approximates $f^{(k)}$, contradicting the assumed hardness of f. Therefore, $20\delta' > a/\sqrt{k}$, and so $\delta' > a\sqrt{\delta}/20$. $\square$

The proof of Theorem 14 is now fairly immediate.

Proof of Theorem 14. Let α and β be the constants hidden in the asymptotic notation of Lemma 17. That is, $k \leq \alpha/\delta$ and f' is $\beta\sqrt{\delta}$-hard. Let $\delta_1 = 1/s(n)$ and let

$$\delta_i = \beta\sqrt{\delta_{i-1}} = \beta^{\sum_{j=0}^{i-2} 2^{-j}} \delta_1^{2^{-i+1}} \geq \beta^2 \delta_1^{2^{-i+1}},$$

for $i \geq 1$. Let ℓ be the greatest number such that $\delta_\ell < \min\{\beta^2, \delta_0\}$. It is easy to see that $\ell = O(\log s(n)) = O(\log n)$.

Let $f_1 = f$ and let f_i be the result of applying the construction from Lemma 17 to f_{i-1}. Let $k_i = \alpha/\delta_i$, let $n_1(n) = n$, and finally let $n_i(n) = k_{i-1}n_{i-1}(n) + k_{i-1}\log k_{i-1} < 2k_{i-1}n_{i-1}(n)$, for n sufficiently large. Notice that f_i is a function on $n_i(n)$ bits. Since $\ell = O(\log n)$, unwinding the recurrence for $n_i(n)$, we get that $n_i(n) < 2^{i-1}\left(\prod_{j=1}^{i-1} k_j\right)n < (2\alpha/\beta^2)^{i-1}\delta_1^{-2}n$. Hence $n_{\ell+1}(n) \leq O(\mathrm{poly}(n)(s(n))^2 n) \leq \mathrm{poly}(n)$. Therefore, after ℓ applications of Lemma 17, we obtain a function f' on $\mathrm{poly}(n)$ bits that is in NP and is c-hard for BPP, where $c \geq \min\{\beta^2, \delta_0\}$. $\square$

5 Uniform hardness amplification to $1/2 - \epsilon$

In this section, we give a different, coding-theoretic proof of the following result by Trevisan [Tre05].

Theorem 18 ([Tre05]). *If* NP *contains a language L that is $1/n^d$-hard for* BPP, *for some constant d, then* NP *also contains a language L' that is $(1/2 - 1/\log^\alpha n)$-hard, for some constant $\alpha > 0$.*

Again, we show amplification starting from a balanced function, and then use Theorem 12 observing that the proof works for almost-balanced functions as well. Together with Theorem 14, this implies that proving Theorem 18 reduces to proving the following.

Theorem 19. *If* NP *contains a balanced Boolean function f that is c-hard for* BPP$//\log n$, *then* NP *contains a Boolean function f' that is $(1/2 - 1/\log^\alpha n)$-hard for* BPP, *for some constant $\alpha > 0$.*

We first outline the main ideas of the proof of Theorem 19, giving a formal proof in the next subsection. As in the previous section, the construction is done in two stages. First, we take a k-wise direct product $f^{(k)}$ of the original function f, for $k = \log n$. Then we encode each symbol of the direct product with an appropriate combinatorially list-decodable binary monotone code $Code$. The concatenation of these two encodings is the truth table of our new function f'.

For the analysis, suppose we have a BPP algorithm that correctly computes f' on more than a $1/2 + \epsilon$ fraction of inputs, for $\epsilon = 1/\log^\alpha n$. By Markov, for at least an $\epsilon/2$ fraction of inputs $\bar{x} = (x_1, \ldots, x_k)$ of $f^{(k)}$, this BPP algorithm correctly computes the codeword $Code(f^{(k)}(\bar{x}))$ for more than a $1/2 + \epsilon/2$ fraction of positions. Fix any such $\bar{x}$. Let w be the word computed by the BPP algorithm on this $\bar{x}$ such that w agrees with $Code(f^{(k)}(\bar{x}))$ for more than a $1/2 + \epsilon/2$ fraction of positions. The combinatorial list-decodability of $Code$ implies that there is a small list of at most $t \approx 1/\epsilon^2$ codewords that have agreement $1/2 + \epsilon/2$ with w. This list can be computed in polynomial time by exhaustive search. Namely, we enumerate all possible k-bit strings msg, keeping on our list only those of them where $Code(msg)$ has agreement $1/2 + \epsilon/2$ with w; since $k = O(\log n)$, this search takes time poly(n). Picking one of the strings msg on our list at random, we get the correct string $f^{(k)}(\bar{x})$ with probability at least $1/t \approx \epsilon^2$. This yields a BPP algorithm that correctly computes $f^{(k)}$ on more than about ϵ^3 fraction of inputs $\bar{x} = (x_1, \ldots, x_k)$. Finally, we use the uniform direct product amplification result of [IJK06] to obtain from this BPP algorithm for $f^{(k)}$ a new BPP$//\log n$ algorithm that computes f on more than $1 - c$ fraction of inputs, contradicting the assumed hardness of f.

There are two caveats with the outline above. First, the combinatorial list decoding for our $Code$ works only for messages $f^{(k)}(\bar{x})$ that are *almost balanced*. Of course, when we take the direct product of the original function f, not all symbols of the direct product $f^{(k)}$ will be balanced. We use the assumption that f is balanced to argue that the fraction of symbols $f^{(k)}(\bar{x})$ that are almost balanced will be high (at least $(1 - \epsilon/4)$) so that the fraction of *almost balanced* inputs $\bar{x}$ where the given BPP algorithm correctly computes the codeword $Code(f^{(k)}(\bar{x}))$ for more than a $1/2 + \epsilon/2$ fraction of positions is at least $\epsilon/2 - \epsilon/4 = \epsilon/4$. Thus we still get a BPP algorithm that correctly computes $f^{(k)}$ on more than about ϵ^3 fraction of inputs $\bar{x}$.

Secondly, the combinatorial list-decodability of $Code$ is not exact but only *approximate*. This means that our small list of candidate messages will contain a message msg that is only *close in Hamming distance* to the correct message $f^{(k)}(\bar{x})$. This yields a BPP algorithm that computes *almost* all bits in the string $f^{(k)}$ for more than an ϵ^3 fraction of $\bar{x}$. Fortunately, the direct product amplification of [IJK06] continues to work even in this case, and so we still get a BPP$//\log n$ algorithm that computes f on more than a $1 - c$ fraction of inputs.

5.1 Proof of Theorem 19

Let $f : \{0,1\}^n \to \{0,1\}$ be a balanced Boolean function in NP that is c-hard for BPP$//\log n$, for some constant c. Consider the direct product function $f^{(k)}$ for $k = \log n$. Next encode every k-bit

string $f^{(k)}(\bar{x})$ for $\bar{x} = (x_1, \ldots, x_k)$ by the monotone binary code $Code = Code_{\text{REC-MAJ}}^{k,r}$ where r is the smallest power of 3 greater than $\log^\gamma n$ for some $\gamma > 0$ to be determined later. The concatenation of all these encodings is defined to be the truth table of the function f'.

We will need the following result from [IJK06]. Below, for $0 < \epsilon, \nu \leqslant 1$ and integer k, we say that a BPP algorithm C ϵ-computes ν-direct product $f^{(k)}$ if with probability at least ϵ over the choice of a random k-tuple $\bar{x}$ and internal randomness of C, the k-bit strings $C(\bar{x})$ and $f^{(k)}(\bar{x})$ agree in at least ν fraction of positions.

Theorem 20 ([IJK06]). *Let f be any n-variable Boolean function, and let $0 < \mu \leqslant 0.4$ be any constant. Suppose a BPP algorithm C ϵ-computes $(1 - 1/k^\mu)$-Direct Product $f^{(k)}$, where $\epsilon = \Omega(\text{poly}(1/k))$. Then there is a randomized $\text{poly}(n, k)$-time algorithm A using advice of size $O(\log(1/\epsilon))$ that computes f on at least $1 - \rho$ fraction of inputs, where $\rho = k^{-\Omega(\mu)}$.*

The second result we need is the combinatorial approximate list-decodability of the monotone code based on Recursive Majority.

Lemma 21. *There exist constants $\alpha, \gamma, \mu > 0$ such that, for any k and for $\epsilon = 1/k^\alpha$ and $r \geqslant k^\gamma$, the code $Code = Code_{\text{REC-MAJ}}^{k,r}$ is $k^{-\mu}$-approximately $(\epsilon/2, 5/\epsilon^2)$-list decodable on $k^{-0.45}$-balanced messages.*

Proof. First we argue by Claim 7 and Lemma 8 that, for sufficiently small α, γ, μ, the code $Code$ is $k^{-1.1\mu}$-approximately $(\epsilon/4, 8/\epsilon^2)$-list decodable on balanced messages. Indeed, for our parameters, the list size will be at most the inverse of $4(\epsilon/4)^2 - (1/2)k^{-0.15\gamma}k^{1.21\mu}$. By choosing α, γ, μ so that $0.15\gamma - 1.21\mu \gg 2\alpha$, we can ensure that the list size is at most $5/\epsilon^2$.

Let w be any binary string of length k^r. Then, by what we just argued, there exists a list $List$ of size at most $5/\epsilon^2$ with the following property: for every balanced k-bit message m, if $Code(m)$ has agreement at least $1/2 + \epsilon/4$ with w, there is a balanced k-bit string $m' \in List$ such that $\Delta(m, m') < k^{-1.1\mu}$.

Now let msg be any $k^{-0.45}$-balanced k-bit string such that $Code(msg)$ has agreement at least $1/2 + \epsilon/2$ with w. Clearly there is a balanced k-bit string msg' such that $\Delta(msg, msg') \leqslant k^{-0.45}$. Note that $\Delta(Code(msg), Code(msg')) \leqslant r/k^{0.45}$, since, by the union bound, a random r-tuple in $[k]^r$ contains a position where msg and msg' differ with probability at most $(r \cdot \Delta(msg, msg'))$. Thus, $Code(msg')$ has agreement at least $1/2 + \epsilon/2 - r/k^{0.45}$ with w. By choosing α, γ such that $\alpha + \gamma \ll 0.45$, we can make this agreement at least $1/2 + \epsilon/4$. In this case, there exists a balanced k-bit string $m \in List$ such that $\Delta(m, msg') < k^{-1.1\mu}$. Since $\Delta(msg, msg') \leqslant k^{-0.45}$, we conclude that $\Delta(msg, m) \leqslant k^{-1.1\mu} + k^{-0.45}$, which is at most $k^{-\mu}$ for small μ. $\square$

Now we can analyze the hardness of the function f' defined above. For $k = \log n$, for α to be determined, and for $\epsilon = k^{-\alpha}$, suppose there is a BPP algorithm A that computes f' for more than a $1/2 + \epsilon$ fraction of inputs of length n, for every length n. By Markov, for at least an $\epsilon/2$ fraction of k-tuples $\bar{x} = (x_1, \ldots, x_k)$, the algorithm A is correct for at least a $1/2 + \epsilon/2$ fraction of bits in the codeword $Code(f^{(k)}(\bar{x}))$. Since f is balanced, we get by Chernoff bounds that the fraction of k-tuples $\bar{x}$ such that $f^{(k)}(\bar{x})$ is not $k^{-0.45}$-balanced is at most $e^{-k^{0.09}} < \epsilon/4$, for sufficiently large k. Thus, the fraction of k-tuples $\bar{x}$ such that both $f^{(k)}(\bar{x})$ is $k^{-0.45}$-balanced and the algorithm A is correct for at least a $1/2 + \epsilon/2$ fraction of bits in the codeword $Code(f^{(k)}(\bar{x}))$ is at least $\epsilon/2 - \epsilon/4 = \epsilon/4$.

Fix any such tuple $\bar{x}$. Let w be the string computed by the algorithm A on $\bar{x}$ such that w agrees with $Code(f^{(k)}(\bar{x}))$ on at least a $1/2 + \epsilon/2$ fraction of positions. Choose α, γ, μ as in Lemma 21. Then there is a list $List$ of size at most $5/\epsilon^2$ such that the following holds. For every $k^{-0.45}$-balanced

k-bit string msg such that $Code(msg)$ agrees with w on at least a $1/2 + \epsilon/2$ fraction of positions, there is a string $msg' \in List$ such that $\Delta(msg, msg') < k^{-\mu}$. Such a list can be easily constructed in time poly(n) by exhaustive search. Once the list is found, we output a random k-bit string on the list. With probability at least $\epsilon^2/5$, this string will be $k^{-\mu}$-close to $f^{(k)}(\bar{x})$. Since this is true for each of at least $\epsilon/4$ of all tuples $\bar{x}$, we get a randomized polynomial-time algorithm that $\Omega(\epsilon^3)$-computes $1 - k^{-\mu}$-direct product $f^{(k)}$. By Theorem 20, we conclude that f can be computed by a BPP$//\log \log n$ algorithm for more than a $1 - O(k^{-\Omega(\mu)}) > 1 - c$ fraction of inputs, contradicting the assumed hardness of f.

Remark 1. *The real bottleneck in getting a significant improvement of Theorem 18 is the lack of an* efficient *approximately list-decoding algorithm for the combinatorially list-decodable monotone code $Code_{Rec\text{-}Maj}^{k,r}$. Any algorithm better than exhaustive search would yield an improved hardness amplification result, since then we could take k larger than $\log n$ and use Theorem 20 to get bigger hardness for $f^{(k)}$. In particular, an approximately list-decoding algorithm for almost balanced messages which runs in time polynomial in the message size, would allow us to take $k = $ poly(n), yielding a near-optimal amplification result.*

6   Conclusion

We have shown a strong connection between monotone codes and uniform hardness amplification in NP. Monotone codes with good (information-theoretic) approximate list-decodability are sufficient to achieve significant hardness amplification, but the obvious open question is to find a monotone code that can be *efficiently* approximately list-decoded. Such a code would improve the hardness amplification parameters. Moreover the code need not have good rate or be locally encodable, and it needs to work only on almost-balanced messages. Certainly the code we use, $Code_{\text{REC-MAJ}}^{N,k}$ (for appropriate N and k), seems like a good candidate.

It remains to be seen if monotone codes are of interest in general coding theory, but they do seem to have applications elsewhere in hardness amplification. For example, monotone circuits are one of the most general models of computation for which good lower bounds are known [Raz85, And85, AB87]. The techniques used for these lower bounds seem to be able to prove only worst-case lower bounds (or very mild average case [HT04]), whereas there exist monotone functions for which even exponential-size monotone circuits cannot achieve advantage better than $1/\sqrt{n}$ (this is an application of [O'D04], which allows amplification from mild average-case hardness to almost optimal hardness, as described by [HT04]). The obstacle to showing an explicit function in, say, EXP, that has this much average-case hardness is the lack of a worst-case to mild-average-case amplification result. A good-rate, locally decodable monotone code that has a monotone decoding algorithm with appropriate distance parameters would achieve such a result.

References

[AB87]   Noga Alon and Ravi B. Boppana. The monotone circuit complexity of boolean functions. *Combinatorica*, 7(1):1–22, 1987.

[AGGM06]   A. Akavia, O. Goldreich, S. Goldwasser, and D. Moshkovitz. On basing one-way functions on NP-hardness. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, pages 701–710, 2006.

[Aka06]   A. Akavia. Personal Communication, 2006.

[And85]   A. E. Andreev. On a method for obtaining lower bounds for the complexity of individual monotone functions. *Sov. Math. Dokl.*, 31:530–534, 1985.

[BDCGL92]   S. Ben-David, B. Chor, O. Goldreich, and M. Luby. On the theory of average-case complexity. *Journal of Computer and System Sciences*, 44(2):193–219, 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.

[BT03]   A. Bogdanov and L. Trevisan. On worst-case to average-case reductions for NP problems. In *Proceedings of the Forty-Fourth Annual IEEE Symposium on Foundations of Computer Science*, pages 308–317, 2003.

[FF93]   J. Feigenbaum and L. Fortnow. Random-self-reducibility of complete sets. *SIAM Journal on Computing*, 22(5):994–1005, 1993.

[GL89]   O. Goldreich and L.A. Levin. A hard-core predicate for all one-way functions. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pages 25–32, 1989.

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

[HT04]   E. Hazan and I. Tourlakis. An average-case monotone circuit lower bound for NP. Manuscript, 2004.

[HVV04]   A. Healy, S. Vadhan, and E. Viola. Using nondeterminism to amplify hardness. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, pages 192–201, 2004.

[IJK06]   R. Impagliazzo, R. Jaiswal, and V. Kabanets. Approximately list-decoding direct product codes and uniform hardness amplification. In *Proceedings of the Forty-Seventh Annual IEEE Symposium on Foundations of Computer Science*, pages 187–196, 2006.

[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.

[Imp02]   R. Impagliazzo. Hardness as randomness: A survey of universal derandomization. *Proceedings of the ICM*, 3:659–672, 2002. (available online at arxiv.org/abs/cs.CC/0304040).

[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.

[MV99]   P.B. Miltersen and N.V. Vinodchandran. Derandomizing Arthur-Merlin games using hitting sets. In *Proceedings of the Fortieth Annual IEEE Symposium on Foundations of Computer Science*, pages 71–80, 1999.

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

[O'D04]   R. O'Donnell. Hardness amplification within NP. *Journal of Computer and System Sciences*, 69(1):68–94, 2004. (preliminary version in STOC'02).

[Raz85]   A. A. Razborov. Lower bounds on the monotone complexity of some boolean functions. *Sov. Math. Dokl.*, 31:354–357, 1985.

[STV01]   M. Sudan, L. Trevisan, and S. Vadhan. Pseudorandom generators without the XOR lemma. *Journal of Computer and System Sciences*, 62(2):236–266, 2001. (preliminary version in STOC'99).

[TCZ]   The Complexity Zoo. `http://qwiki.caltech.edu/wiki/Complexity_Zoo`.

[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.

[Tre05]   L. Trevisan. On uniform amplification of hardness in NP. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, pages 31–38, 2005.

[TV02]   L. Trevisan and S. Vadhan. Pseudorandomness and average-case complexity via uniform reductions. In *Proceedings of the Seventeenth Annual IEEE Conference on Computational Complexity*, pages 103–112, 2002.

[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.