If NP languages are hard on the worst-case then it is easy to find their hard instances

Dan Gutfreund*
School of Computer Science and Engineering,
The Hebrew University of Jerusalem, Israel, 91904
danig@cs.huji.ac.il

Ronen Shaltiel†
Department of Computer Science,
University of Haifa, Israel, 31905
ronen@cs.haifa.ac.il

Amnon Ta-Shma
Department of Computer Science,
Tel-Aviv University, Israel, 69978
amnon@post.tau.ac.il

Abstract

We prove that if NP $\not\subseteq$ BPP, i.e., if some NP-complete language is worst-case hard, then for every probabilistic algorithm trying to decide the language, there exists some polynomially samplable distribution that is hard for it. That is, the algorithm often errs on inputs from this distribution. This is the first worst-case to average-case reduction for NP of any kind.

We stress however, that this does not *mean that there exists* one fixed *samplable distribution that is hard* for all *probabilistic polynomial time algorithms, which is a pre-requisite assumption needed for OWF and cryptography (even if not a sufficient assumption). Nevertheless, we do show that there is a fixed distribution on instances of NP-complete languages, that is samplable in* quasi-polynomial time *and is hard for all probabilistic polynomial time algorithms (unless NP is easy in the worst-case).*

Our results are based on the following lemma that may be of independent interest: Given the description of an efficient (probabilistic) algorithm that fails to solve SAT in the worst-case, we can efficiently generate at most three Boolean formulas (of increasing lengths) such that the algorithm errs on at least one of them.

1. Introduction

It is traditional in computational complexity to measure *worst-case* complexities, and say that an algorithm is feasible if it can be solved in worst-case polynomial time (i.e., in P or BPP). A general belief is that all NP-complete languages do not have feasible algorithms that are correct on every input. Thus under a worst-case measure of complexity, these problems are hard. However, this does not mean that *in practice* NP-complete problems are hard. It is possible that for a given problem, its hard instances are "rare", and in fact it is solvable efficiently on all instances that actually appear in practice.

1.1. Average-case complexity: the cryptographic approach

This point of view led Levin [24] to develop the theory of average-case complexity of NP problems, trying to capture the notions of easiness and hard-

*Research supported in part by the Leibniz Center, the Israel Foundation of Science, a US-Israel Binational research grant, and an EU Information Technologies grant (IST-FP5).

†Part of this research was done while staying at the Weizmann Institute and supported by the Koshland scholarship.

ness on the average. We now give a brief overview of the main concepts in this theory.

Let L be a language that we want to decide, and let $\mathcal{D}$ be a distribution over the instances.[1] We want to measure how hard it is to decide L with respect to the distribution $\mathcal{D}$. Thus we look at the complexity of the *distributional problem* $(L, \mathcal{D})$.

Trying to capture the notion of "real-life" instances, we look at input distributions that can be efficiently generated. We say that $\mathcal{D}$ is *samplable* if there exists some probabilistic polynomial-time machine that generates the distribution.[2]

This leads to a definition of the distributional equivalent of the class NP.

Definition 1. *The class DistNP contains all the distributional problems $(L, \mathcal{D})$, where $L \in NP$ and $\mathcal{D}$ is a samplable distribution.*

Levin [24] and subsequent papers (e.g. [37, 6, 12, 13]) defined the notions of reductions between distributional problems and completeness in *Dist*NP, and were able to show several complete problems in the class.

Next, we turn to define the notion of a distributional problem being "easy" on the average.[3]

Definition 2. *Let $\mathcal{C}$ be a class of algorithms and $(L, \mathcal{D})$ a distributional problem. We say that $(L, \mathcal{D}) \in Avg_{p(n)}\ \mathcal{C}$ if there is an algorithm $A_D \in \mathcal{C}$ such that for every large enough n, $\Pr_{x \in D_n}[A_D(x) = L(x)] \geq p(n)$.*

When thinking of efficient solutions, we should think of $\mathcal{C}$ as being P or BPP. Thus the average-case analogue of the statement NP $\nsubseteq$ BPP (i.e. *Dist*NP $\nsubseteq$ AvgBPP) is that there exists an NP language L, and a samplable distribution $\mathcal{D}$, such that every efficient probabilistic algorithm errs with high probability over instances drawn from $\mathcal{D}$.

This notion seems to capture the intuition of a problem being hard on the average. Furthermore, it seems to be the right notion of hardness that is needed for cryptographic applications. For example, consider a crypto-system that is based on the RSA encryption scheme. We would like the user to be able to sample efficiently composite numbers of the form $N = p \cdot q$ (where p, q are prime numbers known to the user) such that with high probability over this sample space, an adversary would not be able to factor the chosen number and to break the encryption. Thus, in particular, we require a samplable distribution that together with the problem of factoring will give a distributional problem that is not in AvgP.[4]

1.2. Average-case complexity: the algorithmic approach

While the notion of distributional problems seems to capture the intuitive meaning of *hardness on average*, we now explain that it may be insufficient to capture the intuitive meaning of *easiness on average* and may be problematic when taking an algorithmic point of view. That is, saying that a distributional problem $(L, \mathcal{D})$ is easy on the average means that there is an efficient (average-case) algorithm for the specific distribution $\mathcal{D}$. Often, we don't have precise knowledge of the distribution of the inputs, and even worse, this distribution may change in the future. A reasonable guarantee is that the inputs are drawn from

[1] Throughout this paper we will think of $\mathcal{D}$ as an ensemble of distributions $\{\mathcal{D}_n\}_{n \in \mathbb{N}}$, where $\mathcal{D}_n$ is a distribution over $\{0,1\}^n$. Levin considered a single distribution over all inputs, but Impagliazzo [17] showed that it is essentially equivalent to consider ensembles of distributions.

[2] More formally we require that there is a probabilistic polynomial time algorithm such that the output of this algorithm on input 1^n is the distribution $\mathcal{D}_n$. We also remark that Levin considered a more restricted class of distributions that are P-computable, i.e. their probability function can be calculated in polynomial time. However, Impagliazzo and Levin [18] showed that in order to capture the notions of hardness and completeness for NP distributional problems, we can consider the more general notion of samplable distributions.

[3] The actual definitions given in [24] differ in various details from the definition we give, for example whether the algorithm has a zero sided, one sided or two sided error, and whether the algorithm has a strict running time or only expected running time. The definition we give here is due to [17], where it is called *Heur* P.

[4] This requirement in itself does not suffice for this application, because we need to generate numbers according to the distribution *together with their factorization*. The point we want to make is that it is *necessary*.

some samplable distribution (which, as before, is our model of "real-life" instances). Thus we would like to design an algorithm that is guaranteed to succeed with good probability whenever the inputs are sampled from some samplable distribution. This gives rise to the following definition, due to Kabanets [22].

Definition 3. *(Pseudo classes) Let $\mathcal{C}$ be a class of algorithms and L a language. We say that $L \in Pseudo_{p(n)}\ \mathcal{C}$ if there exists an algorithm $B \in \mathcal{C}$ such that for every samplable distributions $\mathcal{D} = \{\mathcal{D}_n\}_{n\in\mathbb{N}}$ we have that for large enough n, $\Pr_{x\in\mathcal{D}_n}[B(x) = L(x)] \geq p(n)$.*

When $\mathcal{C}$ is a class of probabilistic algorithms, there are subtleties in this definition (as well as in Definition 2). In the introduction we ignore these subtleties and we refer the reader to Definition 4 that addresses these issues.

Note that if $L \subseteq \text{Pseudo}_p\ \mathcal{C}$ then for every samplable distribution D, $(L, D) \in \text{Avg}_p\ \mathcal{C}$, while the converse does not seem to hold. Definition 3 and its generalizations arise in the context of derandomization in the uniform setting [21, 22, 25, 36, 14] (the reader is referred to [23] for a recent survey). We believe this definition captures a basic notion in computational complexity. We want to mention that Trevisan and Vadhan [36] also made the distinction between definitions of average-case complexity for "hardness" and definitions for "easiness".

1.3. Worst-case to average-case reductions in NP

Cryptography has been a very successful field in the last few decades. A vast amount of cryptographic primitives are known to exist under complexity theoretic assumptions. Yet, the basis on which cryptography relies is much less understood.

The weakest assumption that is known to allow many cryptographic applications is the existence of one-way functions. However, all the current candidate constructions of one-way functions rely on the hardness of specific functions (such as factoring, discrete log or lattice problems [8, 31, 28, 3, 4, 29, 30]) that are not known to be NP–complete, and very often are *not* likely to be NP–complete [10, 2].

The desire to make the foundations of cryptography more credible, or at least better understood, drove much work in the past, and is a question of fundamental importance. In particular it was shown that the existence of OWF is equivalent to the existence of many cryptographic primitives such as: pseudo-random generators (PRG), pseudo-random functions, bit-commitments, and digital signatures [11, 19, 15, 32, 26].

Perhaps the holy grail of the area is the attempt to base the existence of OWF on a worst-case computational complexity assumption, such as NP $\neq$ P or NP $\not\subseteq$ BPP. One thing that is clear from the beginning is that the assumption that OWF exist, at the very least, implies that there exists a language in NP that is hard on average for BPP. Thus an important (and necessary) step towards basing cryptography on the NP $\not\subseteq$ BPP assumption is showing a worst-case to average-case reduction for an NP–complete problem.

Open Problem 1. *Does NP $\not\subseteq$ BPP imply DistNP $\not\subseteq$ $Avg_{1-n^{-O(1)}}$ BPP?*

Open problem 1 has been the focus of some recent research (the reader is referred to [7] for more details). We note that the error parameter $n^{-O(1)}$ can be significantly improved by known hardness amplification results [18, 27, 16, 34, 35]. Some of the aforementioned results require hardness against circuits (rather than probabilistic polynomial time algorithms).

Worst-case to average-case reductions have been shown to exist for complete languages in high complexity classes such as EXP, PSPACE, and $\sharp$P (where hardness is measured both against uniform and nonuniform classes) [5, 20, 33, 36]. In contrast, no worst-case to average-case reductions are known for NP-complete problems. In fact, there are results that rule out several classes of such reductions. Specifically, Bogdanov and Trevisan [7] (improving on Feingenbaum and Fortnow [9]) show that there is no worst-case to average-case non-adaptive reduction from an NP-complete language to itself unless the polynomial-time hierarchy collapses.[5] Viola [38] showed that there is no *black-box* worst-case to average-case transformation that is computable within the polynomial-time hierarchy.[6]

1.4. Our results

Pseudo classes have received much less attention than average classes. In fact, most of what we know about pseudo classes are "easiness results" showing that assuming the existence of problems that are hard (on the worst case), certain randomized procedures can be derandomized "on the average" [21, 22, 25, 36, 14]. In this paper we study this notion in relation to "hardness". Our main result is a worst-case to average-case reduction for PseudoBPP.

Theorem 1.

1. $NP \neq P \;\Rightarrow\; NP \not\subseteq Pseudo_{5/6}\ P$
2. $NP \neq RP \;\Rightarrow\; NP \not\subseteq Pseudo_{97/100}\ BPP$

This worst-case to average-case reduction in the algorithmic setting, stands in contrast to the failure in proving such a reduction in the cryptographic setting (for the class Avg BPP). To the best of our knowledge, it is the first worst-case to average-case reduction for NP-complete languages under a natural notion of average-case complexity. Stated in words, Theorem 1 says that if NP is hard on the worst case then for any efficient algorithm trying to solve some NP complete language it is possible to efficiently sample instances on which the algorithm errs.

We remark that once we are able to sample hard instances for one specific NP complete problem (say satisfiability) then we can sample hard instances for any complete problem. This follows immediately by using many-one reductions because applying a many-one reduction on instances drawn from a samplable distribution results in another samplable distribution.

In order to establish Theorem 1 we prove a lemma that says that given the description of an efficient algorithm that fails to decide SAT, we can efficiently generate three Boolean formulas such that the algorithm errs on at least one of the three. We stress that the lemma does not rely on any unproven assumptions and is correct whether NP = P or not.

Finally, we use Theorem 1 to show that there is a fixed distribution on instances of SAT (in fact every NP-complete language), that is samplable in quasi-polynomial time, and for every efficient algorithm generates with non-negligible probability instances on which it errs. More formally,

Theorem 2. *Let $f(n), s(n)$ be time-constructible functions such that $f(n) = n^{\omega(1)}$, and $s(n) = \omega(1)$. Then there is a distribution D that is samplable in time $f(n)$, such that for every NP-complete language L, the following holds,*

$$NP \neq RP \Rightarrow (L, D) \notin Avg_{1-1/s(n)}\ BPP$$

Note that D is the same distribution for every NP-complete language. Interestingly, this distribution is very simple to describe (see the proof of Theorem 2 in Section 5).

1.5. Perspective: Impagliazzo's worlds

In [17], Impagliazzo gives an enlightening survey where several scenarios regarding the existence of OWF and the NP vs. BPP question are explored. In particular he considers five possible worlds: Algorithmica (where NP problems are easy on the worst-case), Heuristica (where NP is hard in the worst-case but easy on the average), Pessiland (where NP is hard on the average, but yet there are no OWF), Minicrypt (where OWF exist, but not public-key cryptography), and Cryptomania (where there is public-key cryptography).

When describing his worlds, Impagliazzo considers Levin's notion of average-case complexity

[5] More precisely, a reduction is an oracle machine R that when given as oracle an algorithm that does "too well" on average, solves the NP language on the worst case. This result shows that unless the polynomial-time hierarchy collapses there are no such reductions that make non-adaptive queries to their oracles.

[6] Roughly speaking, a black-box transformation is an algorithm that transforms *any* worst-case hard problem into one that is hard on average, without relying on any specific properties of the problem.

(and its variants). However, Definition 3 suggests another possible world, located between Algorithmica and Heuristica, where NP problems are hard in the worst-case, but for every such problem there is an algorithm that does well on every samplable distribution. Let us call this world Super-Heuristica.

The difference between Heuristica and Super-Heuristica is that in the former, any algorithmic solution to some hard problem is bound to a specific distribution over the inputs. So if this distribution changes we'll have to come up with a new solution to the problem. In Super-Heuristica on the other hand, once a good heuristic for some NP-complete problem has been developed, every new problem that arises, only needs to be reduced to the original problem. Once this is done, we can forget about this problem: Hard instances for its heuristic never come up in practice, not now, nor in the future (as long as "real-life" is modeled by samplable distributions).

Theorem 1 says that Super-Heuristica does not exist. That is, if an NP-complete problem has heuristic that does well on every samplable distribution, then it has an algorithm that does well on every input (or on every distribution, samplable or not). So if we believe that NP $\not\subseteq$ BPP, then heuristics for hard problems will always have to be bound to specific distributions.

1.6. Overview of the technique

In this section we give a high level overview of the proof of Theorem 1. Let us first focus on the first item. That is, we assume that NP $\neq$ P and need to show that for any deterministic algorithm BSAT there is a samplable distribution which generates hard instances for BSAT. The main step in the proof is a lemma that shows that there is a deterministic procedure R that when given as input the description of BSAT and an input n outputs at most three formulas, and for infinitely many n, BSAT errs on at least one of the formulas. In other words, the procedure R *finds* instances such that one of them is hard for BSAT.

The basic idea. We now explain how to prove this Lemma (the main Theorem follows easily from the Lemma). We know that BSAT does not solve SAT, fix some length n on which BSAT makes an error. Our goal is to find an instance on which BSAT errs. The basic idea is to consider the following statement denoted ϕ_n: *"there exists an instance x of length n such that $BSAT(x) \neq SAT(x)$".* Note that this statement is a true statement. If this statement was an NP statement then we could reduce it into an instance of SAT and feed it to BSAT. If BSAT answers 'no' then ϕ_n is an instance on which BSAT errs. If BSAT answers 'yes' then in some sense BSAT "admits" that it makes an error on inputs of length n. We can hope to use BSAT to *find* a witness x to ϕ_n and such a witness x is a formula on which BSAT errs.

Note however, that at the moment it is not necessarily the case that deciding ϕ_n is in NP. This is because it could be the case that BSAT errs only on unsatisfiable formulas. (Say for example that BSAT always answers 'yes'.) Verifying that ϕ_n holds seems to require verifying that a given formula x is unsatisfiable.

A search algorithm. We overcome this difficulty by replacing BSAT with an algorithm SSAT that has the following properties:

- When SSAT answers 'yes' then it also outputs a satisfying assignment, and in particular it never errs when it answers 'yes'.
- If SSAT answers 'no' then BSAT answers 'no'.
- If BSAT answers 'yes' on input x then either SSAT answers 'yes' (and finds a satisfying assignment) or else SSAT outputs three formulas such that BSAT errs on at least one of them.

It is easy to construct such an algorithm SSAT by using the standard self-reducibility property of SAT. More precisely, on input x, the algorithm SSAT attempts to use BSAT to find a satisfying assignment. In every step it holds a formula x that BSAT answers 'yes' on. It then substitutes one variable of x to both "zero" and "one" and feeds these formulas to BSAT. If BSAT answers 'yes' on one of them, then the search continues on this formula. Otherwise, at least one of the answers of BSAT on x and the two derived formulas is clearly incorrect. Finally, SSAT accepts if it finds a satisfying assignment. It is easy to verify that SSAT has the properties listed above.

Finding hard instances. To find a hard instance we change ϕ_n to be the following statement: *"there exists an instance x of length n such that $SAT(x) = 1$ yet $SSAT(x) \neq$ 'yes'"*. Note that now deciding ϕ_n is in NP and therefore we can reduce it to a formula. To find hard instances we run SSAT(ϕ_n). There are three possibilities.

- SSAT finds three instances such that on one of them BSAT errs.
- SSAT answers 'no', but in this case BSAT answers 'no' and ϕ_n is a formula on which BSAT errs.
- SSAT answers 'yes' and finds a satisfying assignment x.

It is important to stress that we're not yet done in the third case. While we know that SSAT errs on x, it's not necessarily the case that BSAT errs on x. In the third case, we run SSAT on x. This time we know that the third possibility cannot occur (because we are guaranteed that SSAT does not answer 'yes' on x) and therefore we will be able to find a hard instance.

Extending the argument to the case where BSAT is randomized. We say that a randomized algorithm conforms with confidence level $2/3$ if for every input x, either the algorithm accepts x with probability $2/3$ or it rejects x with probability $2/3$. When given such an algorithm BSAT we can easily use amplification and get an algorithm $\overline{\text{BSAT}}$ that conforms with confidence level $1 - 2^{-2n}$. As in Adelman's argument [1], for almost all choices of random strings u, $\overline{\text{BSAT}}(\cdot, u)$'s answer "captures" whether BSAT accepts or rejects x. Thus, we can do the same argument as above replacing BSAT with $\overline{\text{BSAT}}(\cdot, u)$ for a uniformly chosen u. We will find hard instances for $\overline{\text{BSAT}}(\cdot, u)$, and with high probability (over the choice of u) one of the instances will be a formula on which BSAT errs with noticable probability.

In general, we cannot assume that BSAT conforms to some confidence level. (For example, BSAT is allowed to flip a coin on some instances). For the general case, we use amplified versions of BSAT and SSAT, together with a more cumbersome case-analysis to implement the idea of the deterministic case.

1.7. Outline

We give some preliminaries in Section 2. In Section 3 we prove the first item of Theorem 1. In Section 4 we prove the second item of Theorem 1. In Section 5 we prove Theorem 2. We conclude with a discussion and open problems.

2. Preliminaries

We denote by $[n]$, the set $\{1, \ldots, n\}$. For a set S, we write $i \in_R S$ for i being sampled from the uniform distribution over S. We use U_n to denote the uniform distribution on $\{0, 1\}^n$.

2.1. Satis£ability

There are many ways to encode formulas as binary strings. In this paper it is convenient to choose an encoding which can be padded. More precisely, if x is an encoding of ϕ then $x \circ 0^i$ is also an encoding of ϕ. This means for example that given x we can substitute some of the variables in x by constants and pad the obtained formula to the same length as x. We define SAT(x) to be "1" if x encodes a satisfiable formula and zero otherwise. It is sometimes convenient to think of satisfiability as a language and we write $x \in$ SAT instead of SAT(x) $= 1$.

2.2. Samplable distributions

An ensemble of distributions $\mathcal{D}$ is an infinite set of distributions $\{\mathcal{D}_n\}_{n \in \mathbb{N}}$, where $\mathcal{D}_n$ is a distribution over $\{0,1\}^n$. If A is a deterministic machine taking two inputs, $A(y;U_n)$ denotes the distribution obtained by picking x uniformly from $\{0,1\}^n$ and evaluating $A(y;x)$. We say that $\mathcal{D} = \{\mathcal{D}_n\}$ is samplable in time $f(n)$ if there exists a machine A such that for every n, $A(1^n;U_{f(n)}) = \mathcal{D}_n$, and the running time of A is bounded by $f(n)$. If $f(n)$ is a fixed polynomial we simply say that $\mathcal{D}$ is *samplable*.

2.3. The class Pseudo BPP

Let PPM denote the class of probabilistic machines that run in polynomial time. We define:

Definition 4. *(Pseudo BPP) We say that $L \in Pseudo_{p(n)}$ BPP if there exists a PPM algorithm B such that for every samplable distributions $\mathcal{D} = \{\mathcal{D}_n\}_{n\in\mathbb{N}}$ we have that for large enough n, $\Pr_{x\in\mathcal{D}_n, y\in_R\{0,1\}^r}[B(x,y) = L(x)] \geq p(n)$.*

2.4. Con£dence level of probabilistic machines

It is convenient to associate some *deterministic function* with any probabilistic machine by requiring that the machine accepts/rejects with some *confidence level*. More precisely, a confidence level is a function $c(n)$ over integers, such that for every n, $1/2 < c(n) \leq 1$. Given a machine $M \in$ PPM and a confidence level $c(\cdot)$ we define a function $M_c : \{0,1\}^* \to \{0,1,*\}$ in the following way: $M_c(x) = 1$ ($M_c(x) = 0$) if M accepts (rejects) x with probability at least $c(|x|)$ over its coins, otherwise $M_c(x) = *$. We say that M accepts (rejects) an input x with confidence level $c(|x|)$ if $M_c(x) = 1$ ($M_c(x) = 0$). Otherwise we say that x is undecided.

We say that a machine M *conforms to confidence level* $c(\cdot)$ if for every x, $M_c(x) \in \{0,1\}$. Given a function $f : \{0,1\}^* \to \{0,1\}$ we say that a machine $M \in$ PPM decides f with confidence level $c(\cdot)$ if for every x $M_c(x) = f(x)$.

We remark that given a machine $M \in$ PPM that conforms to some confidence level $c(n) > 1/2 + 1/n$ we can amplify the confidence level and construct a machine $\bar{M} \in$ PPM that defines the same function relative to confidence level $\bar{c}(n) = 1 - 2^{-n}$.

If we do not explicitly mention the confidence level its default value is 2/3.

3. The deterministic case

In this section we prove the first item of Theorem 1, that is that $\mathrm{NP} \neq \mathrm{P} \Rightarrow \mathrm{NP} \not\subseteq \mathrm{Pseudo}_{5/6}\,\mathrm{P}$. The Theorem follows from the following Lemma that may be of independent interest. Informally, this Lemma says that if NP $\neq$ P then for any algorithm that fails to solve SAT it is possible to produce three formulas where on one of them the algorithm makes an error.

Lemma 1 (Producing hard instances (deterministic version)). *Assume that $NP \neq P$. There is a deterministic procedure R, a polynomial $q(\cdot)$ and a constant d such that the procedure R gets three inputs: integers n, a and a description of a deterministic machine BSAT; the machine R runs in time $n^{d\cdot a^2}$ and outputs at most three formulas where the length of each formula is either n or $q(n^a)$. Furthermore, if BSAT is an algorithm that on inputs of length n runs in time bounded by n^a (for some constant a) then for infinitely many input lengths n, invoking $R(n, a, BSAT)$ gives a set F of formulas such that there exist $\phi \in F$ with $BSAT(\phi) \neq SAT(\phi)$.*

We first observe that the first item of Theorem 1 follows from Lemma 1.

Proof. (of Theorem 1 first item) We assume that NP $\neq$ P, and assume for the purpose of contradiction that $\mathrm{NP} \subseteq \mathrm{Pseudo}_{5/6}\,\mathrm{P}$. By this assumption there exists a deterministic algorithm BSAT that runs in time n^a for some constant a. Furthermore, for any samplable distribution $\mathcal{D} = \{\mathcal{D}_n\}_{n\in\mathbb{N}}$ we have that for large enough n, $\Pr_{x\in\mathcal{D}_n}[\mathrm{BSAT}(x) = \mathrm{SAT}(x)] \geq 5/6$. Let R be the procedure from Lemma 1 and let $q(\cdot)$ and d be the polynomial and the constant guaranteed in the Lemma. We consider the following distribution $\mathcal{D} = \{\mathcal{D}_n\}_{n\in\mathbb{N}}$ defined by a sampling algorithm.

Sampling algorithm for $\mathcal{D}$: On input 1^n we first invoke $R(n, a, \mathrm{BSAT})$ by Lemma 1 R outputs at most three formulas of length either n

or $q(n^a)$. We ignore formulas of length $q(n^a)$. We then check whether there exists a number n' such that $q((n')^a) = n$. If there exist such an n', we invoke $R(n', a, \mathrm{BSAT})$. Once again we get at most three formulas with lengths either n' or $q((n')^a) = n$. We ignore the formulas of length n'. At the end of this process we have at most six formulas $B_n = \{x_1, \ldots, x_t\}$ for $1 \leq t \leq 6$ where each one is of length n. We now uniformly choose one of these formulas and output it.

Note that each invocation of R takes time at most $n^{d \cdot a^2}$ and therefore the sampling procedure runs in polynomial time. From Lemma 1 we have that for infinitely many n, one of the at most three formulas produced by $R(n, a, \mathrm{BSAT})$ is one on which BSAT errs. For each such n, this formula is either of length n or of length $q(n)$. Therefore for infinitely many n, B_n contains a formula on which BSAT errs and with probability at least $1/6$ this formula is chosen by $\mathcal{D}(1^n)$. This is a contradiction as BSAT is supposed to be correct on *any* samplable distribution with probability $5/6$ for large enough n. $\square$

Remark 1 (A deterministic procedure for finding a hard instance). *We can prove a stronger result by using a less constructive argument. Consider* 6 *different deterministic algorithms* $R_1, \ldots, R_6$ *where for each* $1 \leq i \leq 6$, $R_i(1^n)$ *runs the procedure defined in the proof to produce* B_n *and then outputs the* i*'th element in* B_n *(if such exists). For at least one* i, $R_i(1^n)$ *produces a formula on which BSAT errs for infinitely many lengths* n. *Note that whenever* R_i *outputs such a formula it does it with probability* 1*!*

Thus under the assumption that $NP \neq P$ *we can prove a stronger result than the one claimed in Theorem 1, that there exist a* deterministic *algorithm that outputs a hard instance on infinitely many input lengths. Using the terminology of [22] this means that if* $NP \neq P$ *then* $NP \not\subseteq Pseudo_P P$.

The remainder of this section is devoted to proving Lemma 1.

3.1. A search algorithm

Let a be some constant and let BSAT be a deterministic algorithm that runs in time at most m^a on inputs of length m. We define an algorithm SSAT that gets a formula x as input, uses BSAT as an oracle, and can give one of three answers:

- 'yes' and an assignment α that is a proof that x is a satisfiable formula.
- 'no' - that means that BSAT answers 'no' on x.
- "error" and a set F of formulas - taken to mean that BSAT is an incorrect algorithm, and BSAT must be wrong on at least one formula from F.

We now describe the algorithm SSAT. Loosely speaking, SSAT is the standard algorithm that uses BSAT (and the self reducibility of SAT) to conduct a search for a witness. Algorithm SSAT answers that the formula is satisfiable only if the search concludes with a witness. A precise description follows.

The Algorithm SSAT: The input to SSAT is a Boolean formula x over the variables $v_1, \ldots, v_m$ and uses an algorithm BSAT as oracle. On input x, SSAT first asks BSAT if $x^0 = x$ is satisfiable, and if BSAT answers 'no', SSAT also answers 'no'. Otherwise, suppose SSAT has so far fixed the partial assignment $\alpha_1, \ldots, \alpha_i$ and BSAT claims $x^i = x(\alpha_1, \ldots, \alpha_i, v_{i+1}, \ldots, v_m)$ is satisfiable. SSAT asks BSAT if x^i is satisfiable when v_{i+1} is set to $false$ and when it is set to $true$. If BSAT says none is satisfiable, SSAT declares an error and outputs x^i, $x^i_0 = x(\alpha_1, \ldots, \alpha_i, 0, v_{i+2}, \ldots, v_m)$ and $x^i_1 = x(\alpha_1, \ldots, \alpha_i, 1, v_{i+2}, \ldots, v_m)$. Otherwise, BSAT "claims" at least one of the two possibilities is satisfied, and SSAT sets v_{i+1} accordingly (choosing arbitrarily in the case the two possibilities are satisfied). At the end, SSAT holds a complete assignment $\alpha = \alpha_1, \ldots, \alpha_m$ and it checks if it satisfies x. If it does, SSAT outputs 'yes' and the assignment α. Otherwise, it declares an error and outputs the constant formula $x^n = x(\alpha_1, \ldots, \alpha_n)$. By padding formulas before feeding them to BSAT we can make sure that all formulas that come up during an execution of SSAT are of the length of the initial input x.

It is useful to sum up the properties of SSAT with the following straightforward lemma (the proof is omitted).

Lemma 2. *Algorithm SSAT runs in polynomial time and furthermore:*

- *If SSAT answers 'yes' on x then x is satisfiable and SSAT outputs an assignment α that satisfies x.*
- *If SSAT answers 'no' on x then BSAT answers 'no' on x.*
- *If SSAT answers 'error' then SSAT outputs a set F of at most three formulas of length identical to that of x such that there is a formula $y \in F$ with BSAT$(y) \neq$ SAT(y).*

3.2. Finding incorrect instances

We now describe the procedure R from Lemma 1. We obtain as input numbers n, a and a description of a deterministic machine BSAT. We modify BSAT so that it stops after at most m^a steps on inputs of length m. This in turn defines a polynomially related time bound on SSAT.

For every integer n we define an NP statement ϕ_n:

$$\exists_{x \in \{0,1\}^n} \; [\; \text{SAT}(x) = 1 \;\text{ and }\; \text{SSAT}(x) \neq \text{'yes'} \;]$$

Note that indeed there is a circuit of size polynomial in n^a that given a formula x and an assignment α checks whether it is the case that both α satisfies x and SSAT$(x) \neq$ 'yes'. Using the Cook-Levin Theorem the procedure R reduces ϕ_n into a formula ϕ'_n of length polynomial in n^a over variables x, α and z (where z is the auxiliary variables added by the reduction) with the property that ϕ'_n is satisfiable if and only if ϕ_n is satisfiable. Furthermore the Cook-Levin reduction also gives that for any triplet (x, α, z) that satisfies ϕ'_n, x satisfies ϕ_n and α is a satisfying assignment for x. We choose $q(\cdot)$ to be a polynomial that is large enough so that $q(n^a)$ is bigger than the length of ϕ'_n. (Note that this can be done for a universal polynomial q that does not depend on n or BSAT and depends only on the efficiency of the Cook-Levin reduction). We then pad ϕ'_n to length $q(n^a)$.

The procedure R then runs SSAT on ϕ'_n using BSAT as an oracle. There are three cases:

- SSAT declares an error during the run and outputs three (or one) formulas. In this case the procedure outputs these formulas.
- SSAT outputs 'no'. In this case the procedure outputs ϕ'_n.
- SSAT outputs 'yes' and a satisfying assignment (x, α, z) for ϕ'_n. In particular, $x \in$ SAT but SSAT$(x) \neq$ 'yes'. The procedure then runs SSAT on x. There are now only two options (either SSAT declares an error, or answers 'no'):
  - SSAT declares an error during the run and outputs three (or one) formulas. In this case the procedure outputs these formulas.
  - SSAT outputs 'no'. In this case the procedure outputs x.

3.3. Correctness

The most time consuming step in the description of R is running SSAT on ϕ'_n. Recall that SSAT is an algorithm that runs in time poly(n^a) on inputs of length n, and we feed it an input ϕ'_n of length $q(n) =$ poly(n^a). Altogether the running time is bounded by poly(n^{a^2}) as required.

Let us consider a modification SSAT$'$ of SSAT that gives a Boolean answer: It answers 'yes' when SSAT answers 'yes' and 'no' when SSAT answers 'no' or 'error'. By our assumption NP $\neq$ P and in particular SSAT$'$ does not solve SAT. This means that there are infinitely many input lengths n on which SSAT$'$ makes errors. We show that when R is given as input n where n is such an input length then it finds instances on which BSAT is incorrect.

Lemma 3. *For any input length n on which $SSAT'$ doesn't correctly solve SAT then if R is given the triplet $(n, a, BSAT)$ as input then it outputs at most three formulas such that on at least one of them BSAT makes an error.*

Proof. Note that SSAT$'$ has one-sided error: It cannot err when it answers 'yes'. Thus, there exists a satisfiable formula x of length n such that $\mathrm{SSAT}'(x) =$ 'no'. It follows that ϕ'_n is a satisfiable formula. We now follow the description of the procedure R and show that in all cases it finds instances on which BSAT makes an error. Recall that R runs SSAT(ϕ'_n).

- If SSAT declares an error during the execution on ϕ'_n then by Lemma 2, one of the three formulas (of length $q(n^a)$) that SSAT outputs is an error of BSAT.
- If SSAT outputs 'no', then by Lemma 2, BSAT$(\phi'_n) =$ 'no' and thus BSAT makes an error on ϕ'_n.
- If SSAT outputs 'yes', then it also outputs a satisfying assignment (x, α, z) for ϕ'_n and in particular, $x \in$ SAT but SSAT$(x) \neq$ 'yes'. The procedure R then runs SSAT on x. There are now only two options (either SSAT declares an error, or answers 'no'):
  - If SSAT declares an error during the run then by Lemma 2 it outputs at most three formulas (of length n) such that one of them is an error of BSAT.
  - If SSAT outputs 'no' then by Lemma 2, BSAT$(x) =$ 'no'. However we know that x is satisfiable and therefore R finds an error of BSAT.

□

This concludes the proof of Lemma 1.

Remark 2 (Finding hard instances when assuming NP = P). *Lemma 1 shows that assuming that NP $\neq$ P, we can find hard instances for any polynomial time algorithm attempting to solve SAT. We remark that in case NP = P, it is also easy to find hard instances for any polynomial time algorithm that does not solve SAT. This is easily done by using the polynomial time SAT solver to find a witness of a formula that checks the statement "there exists an instance of length n on which the given algorithm and the SAT solver do not agree". Thus Lemma 1 actually holds unconditionally whenever BSAT fails to solve SAT in the worst-case (although for the proof of Theorem 1 it is enough to state it the way it is).*

4. The randomized case

In this section we prove the second item of Theorem 1, that is that NP $\neq$ RP $\Rightarrow$ NP $\not\subseteq$ Pseudo$_{97/100}$ BPP.

The overall structure of the proof follows that of the deterministic setting. The Theorem follows from the following Lemma that is analogous to Lemma 1 and may be of independent interest. Informally, this Lemma says that if NP $\neq$ RP then for any randomized algorithm that fails to solve SAT it is possible to produce three formulas such that with high probability on one of them the algorithm makes an error. It is important to stress that we do not assume that the given randomized machine conforms to some confidence level.

Lemma 4 (Producing hard instances (randomized version)). *Assume that NP $\neq$ RP. For every constant $c > 1/2$ there is a randomized procedure R, a polynomial $q(\cdot)$ and a constant d such that the procedure R gets three inputs: integers n, a and a description of a randomized machine BSAT; the procedure R runs in time $n^{d\cdot a^2}$ and outputs at most three formulas where the length of each formula is either n or $q(n^a)$. Furthermore, if BSAT is a randomized algorithm that on inputs of length n runs in time bounded by n^a (for some constant a) then for infinitely many input lengths n, invoking $R(n, a, BSAT)$ gives with probability $1 - 1/n$ a set F of formulas such that there exist*

$\phi \in F$ *with* $BSAT_c(\phi) \neq SAT(\phi)$.

The proof that the second item of Theorem 1 follows from Lemma 4 is very similar to that in the previous Section. (Here we also have to take into consideration the coin tosses of BSAT and the $1/n$ error probability of R.) We omit the proof.

The remainder of this section is devoted to proving Lemma 4. We assume that NP $\neq$ RP and fix some constant $c > 1/2$.

4.1. Using Adelman's method on BSAT

Given a randomized algorithm BSAT such that for any m and all choices of random coins BSAT runs in time m^a we transform the algorithm BSAT into an algorithm $\overline{\text{BSAT}}$ by amplification.

The algorithm $\overline{\textbf{BSAT}}$: When given a formula x of length n, the algorithm $\overline{\text{BSAT}}$ uniformly chooses n^2 independent strings $v_1, \ldots, v_{n^2}$ where each of them is of length n^a. For every $1 \leq i \leq n^2$, the algorithm applies BSAT(x, v_i) and it outputs the majority vote of the answers.

The purpose of this amplification is to use Adelman's argument [1] to show that many fixed choices of randomness for $\overline{\text{BSAT}}$ "capture" the behavior of BSAT.

Definition 5. *Let n be some integer and $c > 1/2$. We say that a string u of length n^{a+2} is* good for BSAT at length n and confidence c *if for every x of length n and $b \in \{0, 1\}$:*

$$\overline{BSAT}(x, u) = b \Rightarrow BSAT_c(x) \in \{b, *\}$$

The following straightforward lemma is proven by using the same arguments as in [1]. We omit the proof.

Lemma 5. *Let $1/2 < c \leq 1$ be some constant. For every large enough n, the fraction of strings of length n^{a+2} that are good for BSAT at length n and confidence c is at least $1 - 2^{-n}$.*

4.2. A search algorithm

We now define the randomized analog of the search algorithm SSAT from Section 3.1.

The Algorithm SSAT: Given a formula x of length n, the algorithm SSAT uniformly chooses a string $u \in n^{a+2}$. From that point on the algorithm operates exactly in the same way as the version described in Section 3.1 with the exception that whenever the algorithm described there wants to compute BSAT(y) for some formula y, the new algorithm computes $\overline{\text{BSAT}}(y, u)$. We use the notation SSAT(x, u) to describe the outcome of SSAT on x when using u as random coins.

It is useful to sum up the properties of SSAT with the following lemma which is analogous to Lemma 2. The proof (which is omitted) is straightforward by using the properties of a good u (Definition 5).

Lemma 6. *Algorithm SSAT runs in randomized polynomial time. Furthermore, let $1/2 < c < 1$ be some constant and n a large enough integer. Fix some string u of length n^{a+2} that is good for BSAT at length n and confidence level c, then:*

- *If SSAT(x, u) = 'yes' then x is satisfiable and SSAT outputs an assignment α that satisfies x.*
- *If SSAT(x, u) = 'no' then $BSAT_c(x) \in \{0, *\}$.*
- *If SSAT(x, u) answers 'error' then SSAT outputs a set F of at most three formulas of length identical to that of x and there exist $y \in F$ such that $BSAT_c(y) \neq SAT(y)$.*

4.3. Using Adelman's method on SSAT

We now describe the procedure R from Lemma 4. We obtain as input numbers n, a and a description of a randomized machine BSAT. We modify BSAT so that it stops after at most m^a steps on inputs of length m. This in turn defines a polynomially related time bound on SSAT.

We now also need to consider an amplified version of SSAT defined in the following way:

The Algorithm $\overline{\text{SSAT}}$: When given input x of length n the algorithm uniformly chooses n^3 strings $u_1, \ldots, u_{n^3}$ of length n^{a+2}. For every $1 \le i \le n^3$ it runs $\text{SSAT}(x, u_i)$. If at least $\frac{1}{3n} \cdot n^3 = n^2/3$ of the invocations answer 'yes' then $\overline{\text{SSAT}}$ answers 'yes' and otherwise it answers 'no'.[7] Altogether, the algorithm $\overline{\text{SSAT}}$ requires n^{a+5} bits of randomness and for a string r of that length we use $\overline{\text{SSAT}}(x, r)$ to denote the outcome of $\overline{\text{SSAT}}$ on x when using r as coins.

We now reuse Adelman's method, and define when a string r is good for SSAT.

Definition 6. *Let n be some integer. We say that a string r of length n^{a+5} is* good for SSAT *at length n if for every x of length n:*

1. $\Pr_u[SSAT(x, u) = \text{'yes'}] > 1/2n \Rightarrow \overline{SSAT}(x, r) = \text{'yes'}$.
2. $\Pr_u[SSAT(x, u) = \text{'yes'}] < 1/4n \Rightarrow \overline{SSAT}(x, r) \neq \text{'yes'}$.

The following Lemma is analogous to Lemma 5.

Lemma 7. *For every large enough n, the fraction of strings of length n^{a+5} that are good for SSAT at length n is at least $1 - 2^{-n}$.*

4.4. Finding incorrect instances

For every integer n and $r \in \{0,1\}^{n^{a+5}}$ we define an NP statement $\phi_{n,r}$:

$$\exists_{x \in \{0,1\}^n} \; [\; \text{SAT}(x) = 1 \;\text{ and }\; \overline{\text{SSAT}}(x, r) \neq \text{'yes'} \;]$$

Note that indeed there is a circuit of size polynomial in n^a that given a formula x and an assignment α checks whether it is the case that both α satisfies x and $\overline{\text{SSAT}}(x, r) \neq$ 'yes'. Using the Cook-Levin Theorem the procedure R reduces $\phi_{n,r}$ into a formula $\phi'_{n,r}$ of length polynomial in n^a over variables x, α and z (where z is the auxiliary variables added by the reduction) with the property that $\phi'_{n,r}$ is satisfiable if and only if $\phi_{n,r}$ is satisfiable. Furthermore the Cook-Levin reduction also gives that for any triplet (x, α, z) that satisfies $\phi'_{n,r}$, x satisfies $\phi_{n,r}$ and α is a satisfying assignment for x. We choose $q(\cdot)$ to be a polynomial that is large enough so that $q(n^a)$ is bigger than the length of $\phi'_{n,r}$. (Note that this can be done for a universal polynomial q that does not depend on n, r or BSAT and depends only on the efficiency of the Cook-Levin reduction). We then pad $\phi'_{n,r}$ to length $q(n^a)$.

The procedure R chooses at random strings $u \in \{0,1\}^{q(n)^{a+2}}$ and $r \in \{0,1\}^{q(n)^{a+5}}$. It then runs $\text{SSAT}(\phi_{n,r}, u)$. There are three cases:

- SSAT declares an error during the run and outputs three (or one) formulas. In this case the procedure outputs these formulas.
- SSAT outputs 'no'. In this case the procedure outputs $\phi'_{n,r}$.
- SSAT outputs 'yes' and a satisfying assignment (x, α, z) for $\phi'_{n,r}$. The procedure then uniformly chooses a string $u' \in \{0,1\}^{n^{a+2}}$ and runs $\text{SSAT}(x, u')$. There are three cases:
  - SSAT declares an error during the run and outputs three (or one) formulas. In this case the procedure outputs these formulas.
  - SSAT outputs 'no'. In this case the procedure outputs x.
  - If SSAT outputs 'yes'. In this case the procedure fails.

4.5. Correctness

The most time consuming step in the description of R is running SSAT on $\phi'_{n,r}$. Recall that SSAT is an algorithm that runs in time $\text{poly}(n^a)$ on inputs of length n, and we feed it an input $\phi'_{n,r}$ of length $q(n) = \text{poly}(n^a)$. Altogether the running time is bounded by $\text{poly}(n^{a^2})$ as required.

[7] It should be noted that SSAT is a one sided error algorithm. Thus, it is sufficient that SSAT answers 'yes' *once* to be sure that x is staisfiable. Nevertheless, in our amplification scheme we sometimes answer 'no' even when we are sure that the formula is satisfiable. Loosely speaking, this is because we are "not happy" with the confidence SSAT shows in its own outcome.

Let us consider a modification SSAT$'$ of SSAT that gives a Boolean answer: It answers 'yes' when SSAT answers 'yes' and 'no' when SSAT answers 'no' or 'error'. By our assumption NP $\neq$ RP and in particular SSAT$'$ does not solve SAT. This means that there are infinitely many input lengths n on which SSAT$'$ makes errors. More precisely, it follows that there are infinitely many lengths n such that on each such length n there is a satisfiable formula x of length n such that $\Pr_u[\text{SSAT}'(x, u) = \text{'yes'}] < 1/4n$. (The statement above uses the fact that SSAT$'$ is always correct when it answers 'yes'). We show that when R is given as input n, where n is such an input length, then it finds instances on which BSAT is incorrect. This is stated precisely in the following Lemma that is analogous to Lemma 3.

Lemma 8. *For any input length n on which there exists a satisfiable formula x of length n such that $\Pr_u[SSAT'(x, u) = \text{'yes'}] < 1/4n$, if R is given the triplet $(n, a, BSAT)$ as input then with probability at least $1 - 1/n$ it outputs a set F of at most three formulas such that there exists $y \in F$ such that $BSAT_c(y) \neq SAT(y)$.*

Proof. The procedure R makes three random selections, choosing u, r and u'. With high probability each one is "good" in the appropriate way. More precisely, by Lemmas 5, 7 and the union bound, with probability at least $1 - 3 \cdot 2^{-n}$ the procedure obtains: (1) a string u that is good for BSAT at length $q(n)$ and confidence c; (2) a string r that is good for SSAT at length n; (3) a string u' that is good for BSAT at length n and confidence level c. We fix such strings u, r, u'.

Next we argue that $\phi'_{n,r}$ is satisfiable. We know that there exists a satisfiable x of length n such that $\Pr_u[\text{SSAT}(x, u) = \text{'yes'}] < 1/4n$. It follows (by Definition 6 item 2) that $\overline{\text{SSAT}}(x, r) \neq$ 'yes'. We conclude that $\phi_{n,r}$ and $\phi'_{n,r}$ are satisfiable.

We now follow the description of the procedure R and show that in all cases it finds instances on which BSAT makes an error (relative to confidence level c). Recall that R runs $\text{SSAT}(\phi'_{n,r}, u)$.

- If SSAT declares an error during the execution on $\phi'_{n,r}$ then by Lemma 6 one of the three formulas (of length $q(n^a)$) that SSAT outputs is a formula y such that $\text{BSAT}_c(y) \neq \text{SAT}(y)$.
- If SSAT outputs 'no', then by Lemma 6, $\text{BSAT}_c(\phi'_{n,r}) \in \{*, \text{'no'}\}$ and thus $\text{BSAT}_c(\phi'_{n,r}) \neq \text{SAT}(\phi'_{n,r})$.
- If SSAT outputs 'yes', then it also outputs a satisfying assignment (x, α, z) for ϕ'_n, and in particular $\text{SAT}(x) = 1$ but $\overline{\text{SSAT}}(x, r) \neq$ 'yes'. By Definition 6 item 1, $\Pr_{u'}[\text{SSAT}(x, u') = \text{'yes'}] \leq 1/2n$. We say that a string u' of length n^{a+2} is *bad* if $\text{SSAT}(x, u') =$ 'yes'. When R chose u' it had a probability of at most $1/2n$ to choose a bad string. From now on we assume that the string u' is not bad. The procedure R then runs $\text{SSAT}(x, u')$. There are now only two options (either SSAT declares an error, or answers 'no'):
  - If SSAT declares an error during the run then by Lemma 6 it outputs a set of at most three formulas (of length n) such that one of them, y, satisfies $\text{BSAT}_c(y) \neq \text{SAT}(y)$.
  - If $\text{SSAT}(x, u)$ outputs 'no' then by Lemma 6, $\text{BSAT}_c(x) \in \{*, \text{'no'}\}$ however we know that x is satisfiable and therefore $\text{BSAT}_c(x) \neq \text{SAT}(x)$.

Altogether, we showed that the fraction of triplets u, r, u' on which R finds a hard instance is at least $1 - (3 \cdot 2^{-n} + 1/2n) \geq 1 - 1/n$ for large enough n as required. □

This concludes the proof of Lemma 4. We mention that similar (but more careful) arguments from Remark 2 can show that the Lemma holds unconditionally whenever BSAT fails to solve SAT in the worst-case.

5. A worst-case to average-case reduction for distributional problems

In this section we prove Theorem 2. Namely, we show how to swap the quantifiers and get a single (simple to describe) distribution on SAT instances that is samplable in quasi-polynomial time, and every probabilistic polynomial-time algorithm errs on it with non-negligible probability (unless NP = RP).

Proof. (of Theorem 2) Let $\mathcal{M}_k$ be the set of all probabilistic Turing machines upto the k'th machine, under a standard enumeration of Turing machines. We define the distribution $D = \{D_n\}$ by its generating algorithm: to generate instances of length n, choose uniformly a machine M from $\mathcal{M}_{3 \cdot s(n)/100}$. Run M for $f(n)$ steps on the input 1^n (providing the randomness it needs). If M hasn't halted, output 0^n. Otherwise let x be the output of M. Then pad or trim x to be of length n and output it.

Now assume NP $\neq$ RP. We show $(\mathrm{SAT}, D) \notin \mathrm{Avg}_{1-1/s(n)}\mathrm{BPP}$. Let A be a probabilistic polynomial-time algorithm trying to solve SAT. By Theorem 1, there exists a polynomial time sampler R that samples hard instances for A with a constant probability (i.o.). Let $p(n)$ be R's running time. Let n_0 be such that for every $n \geq n_0$, $R \in \mathcal{M}_{3 \cdot s(n)/100}$ and $f(n) \geq p(n)$.

There are infinitely many input lengths $n \geq n_0$ for which A errs with probability greater than $3/100$ over instances sampled by $R(1^n)$. Let n_1 be such an input length. With probability at least $100/(3 \cdot s(n_1))$, D_{n_1} picks the machine R. It then runs R on 1^{n_1} for at least $p(n_1)$ steps, and because $p(n_1) < f(n_1)$ R halts, and with probability greater than $3/100$ outputs an instance on which A errs. Overall, the probability (over the instances and A's internal coins) that A errs on D_{n_1} is at least $1/s(n_1)$ as desired. $\square$

6. Discussion and open problems

The qualitative difference between hardness against average classes and hardness against pseudo classes, is whether the samplable distribution can be stronger than the polynomial time algorithm and in particular simulate it (as in the pseudo classes), or not (as in the average classes). We believe this is a very fundamental difference. Indeed in the proof of Theorem 1, the samplable distribution runs the algorithm. We therefore believe that our techniques are not sufficient for solving Open Problem 1.

A worst-case to average-case reduction is a mapping from a language L to a language L', such that for every efficient algorithm A' that solves well L' on the average, there is an efficient algorithm A (that uses A') that solves L in the worst-case. As we mentioned in the introduction, there are works that show that certain classes of reductions cannot establish worst-case to average-case connections for NP-complete languages (under Levin's notion). Specifically, [38] show that efficient reductions that only make a black-box use in the language L and the algorithm A' do not exist (unconditionally). [7] show that reductions that start from L that is NP-complete, where A uses A' as a black-box and asks it only non-adaptive queries do not exist (unless the polynomial-time hierarchy collapses). We want to point out that our reduction is non-black-box both in L (we use the downwards self-reducibility of SAT), and in A' (we use the fact that BSAT can be computed by an efficient algorithm, and in particular we use the code of this algorithm). It would be interesting to know whether reductions that are non-black-box both in L and A' are necessary to prove Theorem 1.

A problem that is related (in fact complements) worst-case to average-case reductions, is the problem of hardness amplification. That is, whether the existence of a function that is mildly hard (i.e. every algorithm errs with probability at least $1/poly(n)$ over some samplable distribution on the inputs) implies the existence of a function that is very hard (i.e. every algorithm errs with probability almost half, and therefore is not much better than a coin toss). Here, in contrast to the problem of worst-case to average-case reductions there are many positive results for average classes. It is known that with respect to the uniform distribution (and by [18], it is enough to consider this distribution) hardness amplification can be done for various complexity classes and models of computations and in particular, can be done for functions in EXP and in NP (with various parameters) where the hardness is measured both against uniform and non-uniform classes [39, 5, 20, 33, 36, 34, 27, 16, 35].

Our worst-case to average-case reduction only gives a mildly hard function. One way to improve this would be to show hardness amplification for pseudo classes. We do not know how to do that. In its strongest form, this open question can be stated as:

Open Problem 2. *Does* $NP \not\subseteq Pseudo_{1-n^{-1}}\ BPP$ *imply* $NP \not\subseteq Pseudo_{1/2+n^{-d}}\ BPP$ *for every constant* $d \geq 0$ *?*

Finally, suppose that someone claims that he has an efficient algorithm for SAT. Can we automatically check whether the algorithm is correct? By using the reduction from BSAT to SSAT we can assume that the given algorithm is always correct when it answers 'yes'. Our results show that if NP $\neq$ RP then for infinitely many input lengths we can generate a formula that is satisfiable but the algorithm answers that it isn't. Nevertheless, when our procdure outputs a formula we cannot be sure that it has this property (it may be the case that the algorithm correctly solves SAT or that we don't succeed on the given input length). The problem is that we cannot verify that the formula is satisfiable. An interesting open problem suggested to us by Adam Smith is to come up with a "dream-breaker". This is a procedure that outputs satisfiable formulas *together with a satisfying assignment* on which the given algorithm fails. This allows to verify that the given algorithm errs on the generated formula. We remark that dream-breakers exist if one way functions exist because we can test whether the given algorithm inverts a one-way function on a random input. An interesting open problem is to construct dream-breakers from the assumption that NP is hard on the worst case.

7. Acknowledgements

We thank Salil Vadhan and Adam Smith for interesting discussions about our results. We also thank Salil and Rahul Santhanam for valuable comments on an earlier version of the manuscript.

References

[1] L. Adelman. Two theorems on random polynomial time. In *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, pages 75–83, 1978.

[2] D. Aharonov and O. Regev. Lattice problems in NP intersect coNP. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, 2004.

[3] M. Ajtai. Generating hard instances of lattice problems. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 99–108, 1996.

[4] M. Ajtai and C. Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 284–293, 1997.

[5] L. Babai, L. Fortnow, N. Nisan, and A. Wigderson. BPP has subexponential simulation unless Exptime has publishable proofs. *Computational Complexity*, 3:307–318, 1993.

[6] S. Ben-David, B. Chor, O. Goldreich, and M. Luby. On the theory of average case complexity. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 379–386, 1990.

[7] A. Bogdanov and L. Trevisan. On worst-case to average-case reductions for NP problems. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 308–317, 2003.

[8] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, 1976.

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

[10] O. Goldreich and S. Goldwasser. On the limits of non-approximability of lattice problems. *Journal of Computer and System Sciences*, 60 (3):540–563, 2000.

[11] O. Goldreich, S. Goldwasser, and S. Micali. How to construct pseudorandom functions. *Journal of the ACM*, 33(2):792–807, 1986.

[12] Y. Gurevich. Matrix block decomposition is complete for the average case. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 802–811, 1990.

[13] Y. Gurevich. Average case completeness. *Journal of Computer and System Sciences*, 42(3):346–398, 1991.

[14] D. Gutfreund, R. Shaltiel, and A. Ta-Shma. Uniform hardness vs. randoness tradeoffs for Arthur-Merlin games. *Computational Complexity*, 12:85–130, 2003.

[15] J. Håstad, R. Impagliazzo, L. Levin, and M. Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.

[16] A. Healy, S. Vadhan, and E. Viola. Using nondeterminism to amplify hardness. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, pages 192–201, 2004.

[17] R. Impagliazzo. A personal view of average-case complexity. In *Proceedings of the 10th Annual Conference on Structure in Complexity Theory*, pages 134–147, 1995.

[18] R. Impagliazzo and L. Levin. No better ways of finding hard NP-problems than picking uniformly at random. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 812–821, 1990.

[19] R. Impagliazzo and M. Luby. One-way functions are essential for complexity based cryptography. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 230–235, 1989.

[20] R. Impagliazzo and A. Wigderson. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 220–229, 1997.

[21] R. Impagliazzo and A. Wigderson. Randomness vs. time: de-randomization under a uniform assumption. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science*, pages 734–743, 1998.

[22] V. Kabanets. Easiness assumptions and hardness tests: Trading time for zero error. *Journal of Computer and System Sciences*, 63 (2):236–252, 2001.

[23] V. Kabanets. Derandomization: A brief overview. *Bulletin of the European Association for Theoretical Computer Science*, 76:88–103, 2002.

[24] L. Levin. Average case complete problems. *SIAM Journal on Computing*, 15 (1):285–286, 1986.

[25] C.-J. Lu. Derandomizing Arthur-Merlin games under uniform assumptions. *Computational Complexity*, pages 247–259, 2001.

[26] M. Naor. Bit commitment using pseudorandom generators. *J. of Cryptology*, 4:151–158, 1991.

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

[28] M. O. Rabin. Digitalized signatures and public key functions as intractable as factoring. *MIT/LCS/TR-212*, 1979.

[29] O. Regev. New lattice based cryptographic constructions. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 407–416, 2003.

[30] O. Regev and D. Micciancio. Stronger average case to worst case connections. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, 2004.

[31] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[32] J. Rompel. One-way functions are necessary and sufficient for secure signatures. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 387–394, 1990.

[33] M. Sudan, L. Trevisan, and S. Vadhan. Pseudorandom generators without the XOR Lemma. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 537–546, 1999.

[34] L. Trevisan. List decoding using the XOR lemma. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 126–135, 2003.

[35] L. Trevisan. On uniform amplification of hardness in NP. Manuscript, 2004.

[36] L. Trevisan and S. Vadhan. Pseudorandomness and average-case complexity via uniform reductions. In *Proceedings of the 17th Annual IEEE Conference on Computational Complexity*, pages 129–138, 2002.

[37] R. Venkatesan and L. Levin. Random instances of a graph coloring problem are hard. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 217–222, 1988.

[38] E. Viola. Hardness vs. randomness within alternating time. In *Proceedings of the 18th Annual IEEE Conference on Computational Complexity*, pages 53–62, 2003.

[39] A. C. Yao. Theory and applications of trapdoor functions. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982.