Succinct Permanent is *NEXP*-hard with Many Hard Instances

by

Shlomi Dolev, Nova Fandina and Dan Gutfreund

Technical Report $\#12-05$

May 2012

Succinct Permanent is *NEXP*-hard with Many Hard Instances[*]

(Extended Abstract)

Shlomi Dolev[1], Nova Fandina[1], and Dan Gutfreund[2]

[1]Department of Computer Science, Ben Gurion University of the Negev, Israel
[2]IBM Research, Tel Aviv, Israel

Abstract

Finding a problem that is both hard to solve and hard to solve on many instances is a long standing issue in theoretical computer science. In this work we prove that the Succinct Permanent $\bmod p$ is $NEXP$ time hard in the worst case (via randomized polynomial time reduction).

We find hard instance for any given heuristic, either by running the heuristic applying the techniques in [5] to $NEXP$, or using auxiliary provers to gain more efficient procedure for finding a hard instance.

We provide a technique for building an exponential set (in the number of additional bits added to the found instance) of hard instances of the problem.

1 Introduction

Finding provable hard problems with many hard instances is a challenging task in computer science. The implications maybe in cryptography, for example as a part of the Merkle's Puzzles technique [9, 4] for establishing a shared key.

Candidates for a source of such provable hard problems are $NEXP$-hard problems (we use $NEXP$ to denote $NEXPTIME$). The reason is that there are no polynomial solutions for such problems as the complexity class P is strictly within the $NEXP$ complexity class [6, 10]. In particular, Papadmitriou and Ynnakakis proved the following Theorem:

Theorem 1.1 *Let A be some computational problem on graphs. Then, if $3SAT$ can be reduced via reduction-projection to A then $Succ - A$ problem is $NEXP$ time hard.*

[*]Partially supported by Lynne and William Frankel Center for Computer Science, the Israeli Ministry of Science (Russia Israel grant), and Rita Altura Trust Chair in Computer Science. Emails:{dolev,fandina}@cs.bgu.ac.il.

Thus, solving NP-complete graph problems when the graphs are represented in a succinct form is $NEXP$-complete [11].

The *permanent* is a candidate for a problem that has many instances which could be proven to be hard. The reason is that there is a proof of random self-reducibility of the permanent, where the solution of a small random set of instances solves the given instance [8]. Thus, the permanent is hard on average as in the worst case.

Related work There are many results showing an equivalence of the worst case and average case hardness for the problems of high complexity classes, basically PSPACE and above [3, 12]. The most recent paper in this line of research is of Trevisan and Vadhan [12] where the authors show that if $BPP \neq EXP$ then EXP has a problem that is hard for all efficient algorithms with respect to uniform distribution over the inputs. These hard on average problems are obtained by encoding (using error correcting schemes in the style used in probabilistically checkable proofs) the entire truth table of another hard in the worst case problem, which we call the *source problem.* Thus, roughly speaking, ensuring the identification of the value of a certain bit in the encoded truth table requires knowledge of the solution of the worst case hard instances of the source problem. And hence every bit in the encoded truth table is hard to compute as the worst case of the source problem. Therefore, the hardness of revealing any bit in the encoded truth table is similar, implying hardness in the average case over the chosen indexes.

Unfortunately, these languages may have the following property. After investing an exponential effort on solving the worst case hard instance(s) of the source problem, one may efficiently solve all instances (of the same length) of the new encoded bits. In other words, if many instances of the problem are used (perhaps as part of cryptographic primitives) one may invest time (or is lucky to find a way to identify the hard instances) to solve the hard instances of the source problem and then nullify the hardness of all other encoded bit instances at once. Another drawback in such a reduction, is the impossibility to use the obtained hard on average bits as puzzles in the Merkle's puzzle scheme. Therefore in the sequel, we avoid the usage of the truth table encoding reduction.

It is interesting to note, that the problems with such a property belong to the class $LEARN$ that was defined by Impagliazzo and Wigderson in [7]. Namely, the problems such that it is possible to construct a boolean circuit that solves the problem for length n in time PPT^{f_n}.

Our contribution. We prove that the succinct permanent problem modulo a prime is $NEXP$-hard. We then extend existing techniques for identifying hard instances for a given heuristic. We provide a new technique that efficiently generates hard instances for superpolynomial time heuristic. Then we show that for any given hard instance of the succinct permanent modulo a prime we can produce exponentially many sets of the hard instances. We then discuss the possibility of the following property of the generated sets: the answers of the instances of some set do not reveal information concerning the answers of the instances of another set.

Organization. In the next section we prove that the decision problem of whether the value of a permanent of a matrix is zero, when the matrix is given in a succinct representation, is *NEXP* time hard. We use the obtained result to prove in Section 3 that computing the permanent modulo a prime number is *NEXP* time hard. Then in Section 4 we turn to the problem of finding a hard instance for any given heuristic. We present a polynomial search for finding hard instances in case the heuristic is polynomial, and present a polynomial search that uses two provers in case the heuristic is superpolynomial (or exponential, assuming *EXP* $\neq$ *NEXP*). Lastly we show a procedure that given a hard instance of the succinct permanent modulo a prime expands the instance to an exponential number of sets. Each set consists of hard on average instances, where the exponential growth is relative to the number of bits added to the input.

2 Zero Succinct Permanent

In this section we introduce the *Zero Succinct Permanent* problem and establish its complexity hardness.

Definition The Zero Succinct Permanent problem is defined by the following input and output:

input: An $O(\log^k n)$ sized boolean circuit C succinctly representing an $n \times n$ integer matrix A (with positive and negative polynomially bounded values) where k is some constant integer.

output: permanent$(A) == 0$.

Definition Let ϕ be a boolean formula and let $\#\phi$ denote the number of satisfying assignments of ϕ. Let C_ϕ be a succinct circuit representation of formula ϕ and $\#C_\phi$ denote the number of satisfying assignments of ϕ.

Next we prove that the Zero Succinct Permanent is *NEXP* time hard.

Theorem 2.1 *Zero Succinct Permanent is $NEXP$ time hard.*

Proof In [11] the authors proved that the Succ-3SAT decision problem is *NEXP* time hard. We reduce this problem to the Zero Succinct Permanent based on techniques presented in [13]. In this seminal paper Valiant presented a polynomial time reduction from $\#3SAT$ to the Permanent problem of integer matrix. Given an instance ϕ of the 3SAT, the reduction constructs a directed, weighted graph G, with weights $-1, 0, 1, 2, 3$, such that

$$permanent(G) = 4^{t(\phi)} \times s(\phi)$$

where $t(\phi)$ is twice the number of occurrences of literals in ϕ, minus the number of clauses in ϕ, and $s(\phi)$ is the number of satisfying assignments of ϕ.

This construction is based on the composition of graph structures in highly regular fashion. An output graph G is obtained by a combination of a polynomial number of *track*, *interchange* and *junction* structures. The order and type of structures are predefined by boolean formula and can be efficiently obtained from the formula. Namely, given two $c|\log x|$-bit integers, the indices of nodes of graph G, it can be determined in polynomial time (of the length of the integers) whether there is an edge between these nodes. The algorithm reads at most polylogarithmic number bits of ϕ. Call this algorithm A.

Let f denote the reduction transformation of Valiant. Given an O($\log^k n$)-sized boolean circuit representation of ϕ, C_ϕ, combine an algorithm A with C_ϕ to obtain a polylogarithmic description of the graph $G = f(\phi)$. Finally, using a relation between $permanent(G)$ and the number of satisfying assignments of ϕ, we obtain: if $permanent(G) == 0$, then $\#\phi = 0$, and therefore there is no satisfying assignments for ϕ; if $permanent(G) > 0$, then $\#\phi > 0$ implying the existence of satisfying assignments for ϕ. Note, that $4^{t(\phi)}$ is a positive integer number. Thus, the Zero Succinct Permanent problem is *NEXP* time hard. ▮

In the next section we discuss the complexity of another variant of the Succinct Permanent problem.

3 Succinct Permanent Modulo a Prime

Definition The Zero Succinct Permanent $\bmod\ p$ problem is defined by the following input and output:

input: An $O(\log^k n)$ sized boolean circuit C succinctly representing an $n \times n$ integer matrix A (with positive and negative polynomially bounded values) where k is some constant integer.
p is a prime number, s.t. $p = O(n^k)$, given in a binary representation.

output: (permanent(A)) $\bmod\ p$ in binary representation.

To prove the hardness of the defined problem it is enough to prove the decision version of it. Namely, the problem that decides whether the permanent of a succinctly represented integer matrix is equal to zero $\bmod\ p$. We call this problem Zero Succinct Permanent $\bmod\ p$. In the previous section, we proved that Zero Succinct Permanent Problem (the same problem, without modulo operation) is *NEXP* time hard in the worst case. Next, we build a polynomial time randomized reduction from Zero Succinct Permanent to Zero Succinct Permanent $\bmod\ p$.

Given an instance of the Zero Succinct Permanent problem, the reduction calls an oracle for the Zero Succinct Permanent $\bmod\ p$ problem to decide whether the permanent of the input matrix is zero with a high probability.

In more details, let C be a boolean circuit of size $O(\log^k n)$ (where k is some constant) encoding an $n \times n$ integer matrix A. A is a matrix with bounded integer values: $|a_{ij}| \leq n^t$ for some constant $t \leq k$. Note that $|permanent(A)| \leq n!(n^t)^n$. By the Chinese Reminder Theorem, to compute the value of $permanent(A)$ it is sufficient to compute $permanent(A) \bmod p'$, for each prime $p' \leq n^2 \times \log(n^t)$ [13]. The number of such primes is too big to be handled by a polynomial time (of the length of the succinct input) deterministic reduction. Let z denote that number of prime numbers. Define U to be the set of the first $2 \times z$ prime numbers. By the Prime Number Theorem, to build a set U it is enough to consider all primes $p' \leq n^3$. In more details, let $\pi(x)$ denote an Eulerian function. The Prime Number Theorem implies that $\pi(x) \approx \frac{x}{\ln x}$. Therefore,

$$\pi(n^3) \approx \frac{n^3}{\ln n^3} \geq 2 \times \frac{t \times n^2 \times \log n}{\ln(t \times n^2 \times \log n)} = 2 \times z$$

for big enough n. Note that the number of bits required to represent each number form the set U is logarithmic in n (therefore, polynomial in the length of the succinctly represented input). We define the set U to be the union of two sets, U_1-the set of primes that are required by the Chinese Reminder Theorem to represent the permanent of the input matrix, and the set U_2 — set of primes we added to U_1 to extend it to form a doubled sized set of primes. Thus, the size of U_1 is z and the size of U_2 is at least z.

A polynomial time randomized algorithm that solves the Zero Succinct Permanent problem is described in Fig. 1.

```
1: $p' \leftarrow$ pick a prime uniformly at random from the set $U$
2: $answer \leftarrow$ call to the oracle of Zero Succinct Permanent Problem mod $p'$
   with a given input circuit
3: **if** $answer == 0$ **then**
4:     return $permanent(A) == 0$
5: **else**
6:     return $permanent(A) \neq 0$
7: **end if**
```

Figure 1: Randomized reduction

If $permanent(A) == 0$ then for each p' in U $permanent(A) \equiv 0 \mod p'$, and therefore the answer will be correct with probability 1.

If $permanent(A) \neq 0$ then define d to be the number of primes from the set U_1, such that $permanent(A) \equiv 0 \mod p$. Note, that by the Chinese Reminder Theorem it holds $d < z$. Therefore, $z - d$ is the number of primes from the set U_1 such that $permanent(A) \not\equiv 0 \mod p$. Let d' denote the number of primes form the set U_2 such that $permanent(A) \equiv 0 \mod p$. It holds that $d' < (z - d)$, since otherwise we will get that $permanent(A) \equiv 0$, because the multiplication of these primes with d primes from U_1 is greater than the multiplication of all primes that are required by the Chinese Reminder Theorem. Hence, the number of primes from the set U_2 such that $permanent(A) \not\equiv 0 \mod p$ is greater than d. Finally, the number of primes from the whole set U that satisfy the previous equation is at least $z + 1$. Therefore, for at least half of the primes $p \in U$ the value of $permanent(A)$ modulo p is not equal to zero. Namely, in case the permanent of A is non-zero the probability of a correct answer is at least $\frac{1}{2}$.

To complete the description of the algorithm, we should approve that it works in polynomial time of the size of the input. First, we present a procedure that chooses a prime number uniformly at random from the set U that is executed in (expected) polynomial time of the (succinct) input. The procedure picks uniformly at random a number x in the range and applies a polynomial time primality test to x [1]. The random choice repeats itself until a prime number is obtained. By the Prime Number Theorem, the expected number of attempts is polynomial in the size of the (succinct) input.

Note that a single oracle call with a prime found in random yields probability $\frac{1}{2}$ for a correct answer (in case the permanent is not zero). We can exponentially enlarge the probability for correct answer by performing a polynomial number of calls to the oracle.

Lastly, we can assume that the input encoded matrices has only positive values, from the field Z_p (when p is also given as a part of the input). Given an input that encodes a matrix with negative values, we can add an additional small boolean circuit that performs an appropriate arithmetical operations to obtain an equivalent $\bmod\ p$ positive valued matrix. The permanent value of the new matrix under modulo operation is not changed.

4 Finding Hard Instance for a Given Heuristic

The hardness of $NEXP$-complete problems we have discussed above is a *worst case* hardness. Namely, given any polynomial time deterministic (or nondeterministic) algorithm claiming to solve the problem, there are infinitely many n, such that algorithm errs on solving at least one instance of length n. And that is due to the fact $P \subset NEXP$ $(NP \subset NEXP)$. An interesting question is whether we can efficiently produce hard instances of the problem. In [5] the authors present a technique that provides hard distributions of the inputs for heuristics (deterministic and randomized) attempting to solve NP-complete problems. However to produce a hard instance of some length n, their technique consumes more time than is required for heuristic to solve this instance.

In this section we will adapt the technique in [5] to provide hard distribution for heuristics that attempt to solve $NEXP$-complete problems. Obviously this technique inherits the mentioned above disadvantage. To overcome this obstacle we use an idea of two-prover interactive protocols that was proposed and discussed in [2]. We present new method to generate hard distributions of $NEXP$ problems against superpolynomial time heuristics.

To generate hard instances for the Succinct Permanent $\bmod\ p$ problem it is enough to show how to efficiently generate hard distribution of any specific $NEXP$-complete problem. To obtain a hard distribution of any other complete language we apply many-one reduction. In particular, we are considering hard distributions of Succ-3SAT problem.

This section is organized as follows: first we discuss polynomial time heuristics, both deterministic and randomized; next we observe the case of superpolynomial time heuristics.

4.1 Polynomial Time Heuristics

Deterministic Case.

Assume we are given deterministic polynomial time algorithm B claiming to solve Succ-3SAT problem. The goal is to generate hard instances for heuristic B. However the result we have established so far is considering some special type of heuristics, namely, the heuristics that have only one sided fault. Suppose we are given algorithm B such that if B answers 'yes' on the input x, then it is assumed that indeed it holds that x is in the language. For now we assume that heuristic trying to solve Succ-3SAT problem satisfies the above requirement. Next we describe a technique that generates a set of instances of the problem that B fails to solve. We use an idea that was proposed by Gutfreund, Shaltiel and Ta-Shma [5]. In their paper they describe procedure that outputs a set of hard instances of the 3SAT problem. We modify their technique to be applied to our case and formulate the following lemma, that is a $NEXP$ version of Lemma 3.1 of [5].

Lemma 4.1 *There is a deterministic procedure R, a polynomial $q()$ and a constant d such that the procedure R gets three inputs: integers* n *and* a *and a description of a deterministic machine B.*

The procedure R runs in time n^{da^2} and outputs at most two boolean circuits where the length of each circuit is either n or $q(n^a)$. Furthermore, if B is an algorithm that runs in time bounded by n^a on inputs of length n (for some constant a) then for infinitely many input lengths n, the invocation of $R(n, a, B)$ gives a set F of boolean circuits such there exists $C \in F$ with Succ-3SAT$(C) \neq B(C)$.

Proof We know that Succ-3SAT has no deterministic polynomial time algorithm. Therefore, there are infinitely many natural numbers n, such that the heuristic B errs on the input of the size n. Next we consider the statement denoted as Φ_n : *'there exists a boolean circuit C of length n such that Succ-3SAT(C)=1 and B(C) $\neq 1$'* . We define a language $Err_B = \{\Phi_n \mid n \in \mathbb{N} \; and \; \Phi_n \; is \; true\}$. Note, that Err_B is not empty (due to our assumption on one sided error of the heuristic), and the length of Φ_n is a polynomial on the terms of n. The first observation is that $Err_B \in NEXP$. Indeed, there is a deterministic exponential (in n^a) time verifier that given as a certificate the boolean circuit C, representing a 3SAT instance ϕ_C, and an assignment α (of an exponential length on the terms of n) for ϕ_C, checks whether it is the case that both α is a satisfying assignment for ϕ_C and $B(C) \neq 1$.

Therefore, we can reduce Err_B to the Succ-3SAT problem using the property of the Cook-Levin reduction, that was noted by authors in [11]. A result of Cook-Levin procedure is a boolean formula that reflects the movements of an exponential time Turing machine verifying a membership of the language Err_B. We call this formula Ψ_n. The variables of this formula are x, α, z, such that x variables describe a boolean circuit, α variables describe an assignment for the formula represented by circuit x, and z are the auxiliary variables added by the reduction. And the following holds: for any (x, α, z) that satisfies Ψ_n, x satisfies Φ_n, and α is a certificate for that. Furthermore, Ψ_n has a highly regular structure. In fact, it can be shown, that there is a polynomial time (on the terms of n^a) algorithm X_{Ψ_n} such that given any two binary strings of length $c \times n$ computes a clause-literal relation of the formula Ψ_n in polynomial in n^a time. Namely, for every statement Φ_n we match a polynomial sized (in terms of n^a) boolean circuit X_{Ψ_n} that encodes a 3SAT formula Ψ_n. We chose $q()$ to be a polynomial that is large enough so that $q(n^a)$ is bigger than the length of X_{Ψ_n}. Finally, we have reduced the language Err_B into the instances of Succ-3SAT problem. Namely, for every Φ_n (polynomially sized on n) there is X_{Ψ_n} (polynomially sized on n^a) such that $\Phi_n \in Err_B$ if and only if $X_{\Psi_n} \in Succ{-}3SAT$.

Searching procedure The hard instance for B is obtained by applying the following searching technique. Assume n is an integer such that B fails to solve an instance of the length n. Run B on X_{Ψ_n}. If B answers *'no'* then it errs on X_{Ψ_n} and we have found a hard instance. If B answers *'yes'* we start a searching process that will hopefully end with the boolean circuit C of the size n such that B errs on it. The process sequentially runs B on the inputs obtained from X_{Ψ_n}, by partial assignment of the variables of Ψ_n that describes a boolean circuit (x variables).

The process is as follows:

- define $\Psi_n^i = \Psi_n(\alpha_1, \ldots, \alpha_i, x_{i+1} \ldots x_n)$, where $\alpha_1, \ldots \alpha_i$ is a partial assignment for variables of Ψ_n that describes a boolean circuit. $\Psi_n^0 = \Psi_n$.

- suppose we have fixed a partial assignment, namely $B\left(X_{\Psi_n^i}\right) = `yes'$. Then define:
  $\Psi_n^i, 1 = \Psi_n(\alpha_1, \ldots, \alpha_i, 1, x_{i+2} \ldots x_n)$.
  $\Psi_n^i, 0 = \Psi_n(\alpha_1, \ldots, \alpha_i, 0, x_{i+2} \ldots x_n)$.

- run $B\left(X_{\Psi_n^i,1}\right)$. If it answers 'yes', define $\Psi_n^{i+1} = \Psi_n^i, 1$.
  Else, run $B\left(X_{\Psi_n^i,0}\right)$. If it answers 'yes', define $\Psi_n^{i+1} = \Psi_n^i, 0$.
  Else, B errs on one of the two $X_{\Psi_n^i,1}$, $X_{\Psi_n^i,0}$. Output them.
- at the end, we hold the whole assignment $C = \alpha_1 \dots \alpha_n$. C is a circuit B errs on. Output it.

Note, that for each i, Ψ_n^i defines a $NEXP$ language. Therefore we can reduce it in polynomial time to $X_{\Psi_n^i}$ instances.
Running time of the searching process In the worst case, the searching process will stop when the whole assignment for x variables is found. In every step of the process we run machine B on the input of length poly(n^a). Since the number of x variables is polynomial in n, and the running time of B is n^a the total time procedure R runs on the inputs n, a, B is poly(n^{a^2}).

Randomized Case.

The main motivation of this section is to produce a hard distribution of instances of the Succinct Permanent $\mod p$ problem. Since the reduction we have built for this problem from Succ-3SAT is randomized, we have to consider producing hard instances for polynomial time randomized heuristics. We assume that $NEXP$ class of problems is hard for BPP. Namely, we assume that $BPP \subset NEXP$. Informally, we want to prove that if $BPP \neq NEXP$ then for any polynomial time randomized algorithm trying to solve Succ-3SAT problem it is possible to efficiently produce two instances of that problem such that with a high probability the algorithm errs on one of them. For that again we follow the technique of generating hard distribution of the instances for randomized heuristics that was proposed in [5].

Next, we provide the results from [5] (without their proofs) and combine them with our observations to get the proof for the following lemma, that is the $NEXP$ analogous lemma to Lemma 4.1 of [5].

Lemma 4.2 *Assume that $NEXP \neq BPP$. For every constant $c > \frac{1}{2}$ there is a randomized procedure R, a polynomial $q()$ and a constant d such that the procedure R gets three inputs: integers* n, a *and a description of a randomized machine **B**. The procedure R runs in time n^{da^2} and outputs at most two boolean circuits where the length of each circuit is either n or $q(n^a)$. Furthermore, if* B *is a randomized algorithm that runs in time bounded by n^a on inputs of length n then for infinitely many input lengths n, invoking $R(n, a, B)$ results with probability $1 - \frac{1}{n}$ in a set* F *of boolean circuits such that there exists $C \in F$ with $Succ - 3SAT(C) \neq B_c(C)$.*

$B_c : \{0,1\}^* \to \{0,1,*\}$ is a deterministic function associated with the randomized machine B in the following way. Given some probabilistic machine M and some function $c(n)$ over integers, such that $\frac{1}{2} < c(n) \leq 1$, $M_c : \{0,1\}^* \to \{0,1,*\}$ is defined as follows: $M_c(x) = 1$ $(M_c(x) = 0)$ if M accepts (rejects) x with probability at least $c(|x|)$ over its coins, otherwise $M_c(x) = *$.

Proof We follow the proof of the Lemma 4.1 in [5].

First, we transform the randomized algorithm B (of the lemma) into a randomized algorithm $\overline{B}$ by amplification.

The algorithm $\overline{B}$: Given an input x of length n algorithm $\overline{B}$ uniformly chooses n^2 independent strings $v_1, \ldots, v_{n^2}$ each of them of length n^a. For every $1 \le i \le n^2$ the algorithm calls $B(x, v_i)$ and outputs the majority vote of the answers.

Next, the authors of [5] prove the following statement:

Lemma 4.3 *Let $\frac{1}{2} < c \le 1$ be some constant. With probability at least $1 - 2^{-n}$, for a randomly chosen $u \in \{0,1\}^{n^{a+2}}$ it holds that for every x of length n and $b \in \{0,1\}$:*

$$\overline{B}(x, u) = b \Rightarrow B_c(x) \in \{b, *\}$$

Now, the randomized heuristic $B(x)$ can be replaced with deterministic algorithm $\overline{B}(x, u)$, where u is a randomly chosen string. To find incorrect instances for randomized algorithm B we find incorrect instances for deterministic machine $\overline{B}(x, u)$. Using Lemma 4.3 we conclude that with high probability one of the instances is incorrect for B as well.

Randomized searching procedure As in the deterministic case, we start the searching procedure with defining the following language. Consider the statement denoted as $\Phi_{n,u}$: *"there exists a boolean circuit C of length n such that Succ-3SAT(C)=1 and $\overline{B}(C, u) \neq 1$"*, for every integer n and $u \in \{0,1\}^{n^{a+2}}$. We define the language $Err_{\overline{B}} = \left\{ \Phi_{n,u} \mid n \in \mathbb{N}, u \in \{0,1\}^{n^{a+2}} and\ \Phi_{n,u}\ is\ true \right\}$. As in the deterministic case, due to assumption of the one sided error of the heuristic B it follows that $Err_{\overline{B}}$ is not empty. In addition, it is clear thats it is a $NEXP$ language. In fact, it can be easily shown, that for infinitely many n, except for probability $\frac{1}{2n}$ for random u we have that $\Phi_{n,u} \in Err_{\overline{B}}$. Applying the same arguments as in the deterministic case, we reduce an instance of $Err_{\overline{B}}$ to the boolean circuit $X_{\Psi_{n,u}}$, with the properties as noted in the deterministic case. Next, we describe the randomized procedure R of Lemma 4.3.

The procedure R chooses at random strings $u \in \{0,1\}^{n^{a+2}}$ and $u' \in \{0,1\}^{q(n^a)^{a+2}}$. Then it runs a searching procedure from the deterministic case on the input $X_{\Psi_{n,u}}$ with the deterministic heuristic $\overline{B}$ with the random choices defined by u', with the following change: when there is a call to $B(x)$, the procedure calls $\overline{B(x, u')}$.

Following that procedure R outputs at most two instances and the following holds: for infinitely many n R outputs a set of instances, such that with probability at least $1 - \frac{1}{n}$ there is an instance C in the set, such that $B_c(C) \neq Succ - 3SAT(C)$.

The analysis of the running time of the randomized searching procedure R is the same as in the deterministic case.

4.2 Superpolynomial Heuristics

Suppose we are given superpolynomial (deterministic) time algorithm claiming to solve Succ-3SAT problem. From the hierarchy theorems of complexity theory, we know that such an algorithm cannot solve all instances of the Succ-3SAT problem correctly. Hence, we would like to efficiently generate distribution of the inputs for which the heuristic fails solving. Note, that in this case we cannot just use the previous technique, as we have no time to run the heuristic. The idea is to use an interactive two provers protocol, in order to identify whether a particular instance is in the language of the given heuristic.

According to [2] it holds that for any $NEXP$ language L that there is a randomized polynomial time verifier machine V and infinitely powerful machines $P_1, \dots P_k$ such that following holds:

1. If $x \in L$ then Pr $(P_1, \dots P_k$ cause V to accept $x) > 1 - 2^n$
2. If $x \notin L$ then for any provers $P'_1, \dots P'_k$, Pr $(P'_1, \dots P'_k$ cause V to accept $x) < 2^{-n}$

The honest provers are machines that use tableau describing the computation of a non-deterministic exponential time Turing machine on input x.

In our setting, we use an interactive proof system for the language L_B – the language of the heuristic B. The provers use two copies of the machine B, in order to interact with the verifier. The idea is to use the scheme of searching process as before, but with the following change.

Every time the searching procedure calls the heuristic B on the input x, we run the verifier-provers protocol with the input x. According to the decision V makes, the process outputs the instance such that B errs on it with high probability or continues to search for such inputs. In that way, we have a randomized polynomial time procedure that outputs at most two instances of the problem, such that B errs on one of them with a high probability.

For the randomized superpolynomial heuristic (under an assumption that $NEXP$ is hard for such class of heuristics), we use the same scheme as in the randomized polynomial case. Namely, first, by amplification argument we replace the randomized machine by deterministic one (defined by random string of choices), and then we use the idea of the two provers protocol.

5 Many Hard Instances

The number of hard instances of the Succinct Permanent $\bmod p$ grows exponentially with the input size. In Section 3 we prove that there exists (for each sufficiently large n) at least one hard instance of size $O(\log^k n)$ of the Succinct Permanent $\bmod p$ problem that requires a given heuristic exponential time to be computed correctly. In Section 4 we show how to find hard instance for any given heuristic. In this section, we use any given hard succinct permanent instance for a given heuristic (of size $O(\log n^k)$) to generate a set of $O(n)$ succinct permanent instances. The generated set is a combination of random self-reducible sets that can efficiently solve the given hard succinct instance. The number of instances in the set is exponential in the additional bits used to enlarge the succinct representation.

Given a boolean circuit C and a prime number p as an input to the Succinct Permanent $\bmod p$. Consider a matrix $A = M(C)$ that is represented by C. Let the first row of A be:

$$(x_{11}\ x_{12}\ \dots\ x_{1\log n}\ x_{1\log n+1}\ \dots\ x_{1n})$$

Then,

$$permanent(A) = x_{11} \times per_1 + \dots + x_{1\log n} \times per_{\log n} + \dots + x_{1n} \times per_n$$

where per_j is a permanent of the Adj_{1j} (adjoint) of the matrix A. We can rewrite it as follows: $permanent(A) = x_{11} \times per_1 + x_{12} \times per_2 + \dots + x_{1\log n} \times per_{\log n} + X$.

The idea is to build $O(n)$ circuits representing matrices, that are obtained from A by replacing the first $\log n$ entries in a manner, that allows computing a permanent of A modulo p in poly-logarithmic time, given permanent results modulo p of $\log n$ randomly chosen circuits from the set.

One of the possibilities to obtain such a construction is to use the features of the Vandermonde matrix. Note that in the reduction algorithm we can use the primes p', that are required by the Chinese Reminder theorem, such that $p' \geq n+1$, without changing the complexity result. Therefore, we can assume that the hard instance is C, p, s.t. the prime p satisfies $p \geq n+1$.

For each $1 \leq i \leq p$ $a_i \in Z_p$ define:

$$r_i = (a_i \; {a_i}^2 \; {a_i}^3 \; \ldots \; {a_i}^{\log n})$$

Note, that it is necessary that $p > n$ in order to construct a set of size that is exponential in the input size.

Let R_i be an $n \times n$ matrix obtained from A, by replacing the first $\log n$ entries of the first row of A with the vector r_i. We show that given the value of $permanent(R_i) \bmod p$ of any $\log n + 1$ matrices from $R_1, \ldots R_p$ there is a polynomial time algorithm that computes $permanent(A) \bmod p$.

Let a be a vector of the first $\log n$ entries of the first row of the matrix A. For simplicity, suppose we are given

$$permanent(R_1) \bmod p \equiv z_1, \ldots permanent(R_{\log n+1}) \bmod p \equiv z_{\log n+1}$$

To compute a permanent of A modulo p, one should compute the values of

$$X \bmod p, \;\; per_1 \bmod p, \;\; per_2 \bmod p, \;\; \ldots per_{\log n} \bmod p$$

We build a system of linear equations. The system contain $\log n + 1$ equations, each with $\log n + 1$ variables. All constants in the system are from the field Z_p, namely, are positive integer numbers. The matrix representation of the system is as follows:

$$\begin{pmatrix} 1 & a_1 & {a_1}^2 & \ldots & {a_1}^{\log n} \\ 1 & a_2 & {a_2}^2 & \ldots & {a_2}^{\log n} \\ \vdots & \vdots & & & \vdots \\ 1 & (a_{\log n+1}) & (a_{\log n+1})^2 & \ldots & (a_{\log n+1})^{\log n} \end{pmatrix}$$

The vector of variables of the system is: $(X \; per_1 \; per_2 \; \ldots \; per_{\log n})$, and the vector of answers is $(z_1 \; z_2 \; \ldots \; z_{\log n} \; z_{\log n+1})$.

Since the matrix is a subset of a Vandermonde matrix, there exists a unique solution to the system. Since all computations are over the field Z_p, the time needed for solving the system is polynomial in the size of the succinctly represented instance. To complete the description of the technique, we should clarify that for each matrix R_i there exists a succinct circuit representation. We construct circuit $C(R_i)$ by combining the circuit C (the succinct circuit representation of the matrix A) with succinct circuit Row_i that contains $\log n$ inputs and $\log n$ outputs. $Row_i(k)$ outputs a binary representation of ${a_i}^k \bmod p$, for $a_i \in Z_p, \; 1 \leq k \leq \log n$.

In principle, the above technique is not restricted only to the first row of the matrix. The building procedure is valid if we choose some row of the matrix (or some column) and some $\log n$ entries of the chosen row (or column) — the permanent of the matrix can be computed by each of its rows (or columns). Therefore, using this observation, we speculate that the succinct permanent problem is not in the $LEARN$ class. That is, having an oracle for the answers of $\log n$ or less instances, of the same generated set, do not reveal the answers of the hard instances of the set nor of another generated set.[1] We emphasize however, that there is an efficient algorithm such that given the answers for at least $logn + 1$ instances from the same generated set, provides in polynomial time answers for any other instance in the same set. Hence, instances should be carefully chosen from different sets. Such a strategy will not base the hardness on a small set of hard instances as done in the encoding truth tables technique of [12].

Note that we introduce a new framework in which one would like to avoid the dependencies of instances in the manner that a solution to one instance (say, after exhaustive search for a private key) reveals a solution to other (private key) instances. Not only that there is a need to introduce provable hard on average instances, it is also important to ensure *non revealing instance solutions*.
Acknowledgments: We thank with pleasure Mihalis Yanakakis and Salil Vadhan for useful inputs.

References

[1] M. Agrawal and N. Kayal and N. Saxena. PRIMES is in P. *Ann. of Math*, 2:781–793, 2002.

[2] L. Babai and L. Fortnow and C. Lund. Non-Deterministic Exponential Time has Two-Prover Interactive Protocols. *Computational Complexity*, 1:3–40, 1991.

[3] L. Babai and L. Fortnow and N. Nisan and A. Wigderson. BPP Has Subexponential Time Simulations Unless EXPTIME has Publishable Proofs. *Computational Complexity*, 3:307–318, 1993.

[4] S. Dolev and E. Korach and G. Uzan. Magnifying Computing Gaps, Establishing Encrypted Communication Over Unidirectional Channels. *Proc. of the 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2007)*, pp. 253-265, November 2007. A Method for Encryption and Decryption of Messages, US Patent 20090296931, 2009.

[5] D. Gutfreund and R. Shaltiel and A. Ta-shma. If NP Languages are Hard on the Worst-Case Then It is Easy to Find Their Hard Instances. *In Proceedings of the 20th Annual IEEE Conference on Computational Complexity*, 243–257, 2005.

[6] H. Galperin and A. Wigderson. Succinct representations of graphs. *Inf. Control*, 56:183–198, April 1984.

[1]Our speculation is based on the fact that the result of one minor does not totally reveal the result of another minor of the permanent, otherwise we may use this property to solve the permanent problem in polynomial time starting with a matrix of all zeros and adding (non zero) lines and columns one after the other calculating the delta in the permanent value.

[7] R. Impagliazzo and A. Wigderson. Randomness vs time: derandomization under a uniform assumption. *J. Comput. Syst. Sci.*, 63:672–688, December 2001.

[8] R. Lipton. New directions in testing. *Distributed Computing and Cryptography, DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, 2:191–202, 1991.

[9] R. C. Merkle. Secure Communications Over Insecure Channels. *CACM*, Vol. 21, No. 4, pp. 294-299, April 1978.

[10] C. H. Papadimitriou. *Computational Complexity* Addison-Wesley, 1994.

[11] C. H. Papadimitriou and M. Yannakakis. A note on succinct representations of graphs. *Inf. Control*, 71:181–185, December 1986.

[12] M. Sudan and L. Trevisan and S. P. Vadhan. Pseudorandom Generators without the XOR Lemma. *J. Comput. Syst. Sci.*, 62:236–266, 2001

[13] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189 – 201, 1979.