J. Cryptology (2003) 16: 71–93
DOI: 10.1007/s00145-002-0038-x

Journal of
CRYPTOLOGY

On the Security of Modular Exponentiation with Application to the Construction of Pseudorandom Generators*

Oded Goldreich
Department of Computer Science and Applied Mathematics,
Weizmann Institute of Science, Rehovot, Israel
oded@wisdom.weizmann.ac.il

Vered Rosen
Check Point Software Technologies Ltd.,
3A Jabotinsky St., Diamond Tower,
Ramat Gan, Israel
vrosen@checkpoint.com

Communicated by Joan Feigenbaum

Received December 2000 and revised September 2002
Online publication 6 December 2002

Abstract. Assuming the intractability of factoring, we show that the output of the exponentiation modulo a composite function $f_{N,g}(x) = g^x \bmod N$ (where $N = P \cdot Q$) is pseudorandom, even when its input is restricted to being half the size (i.e. $x < \sqrt{N}$). This result is equivalent to the simultaneous hardness of the upper half of the bits of $f_{N,g}$, proven by Håstad, Schrift and Shamir. Yet, we provide a different proof that is significantly simpler than the original one. In addition, we suggest a pseudorandom generator that is more efficient than all previously known factoring-based pseudorandom generators.

Key words. Modular exponentiation, Discrete logarithm, Hard-core predicates, Simultaneous security, Pseudorandom generator, Factoring assumption.

1. Introduction

One-way functions play an extremely important role in modern cryptography. Loosely speaking, these are functions that are easy to evaluate but hard to invert. A number-theoretic function which is widely believed to be one-way, is the exponentiation function over a finite field. Its inverse, the discrete logarithm function, is the basis for numerous cryptographic applications. Most applications use a field of prime cardinality, though many of them can be adapted to work in other algebraic structures as well.

* This write-up is based on the Master's Thesis of the second author (supervised by the first author).

A concept tightly connected to one-way functions is the notion of *hard-core predicates*, introduced by Blum and Micali [BM]. A polynomial-time predicate b is called a hard-core of a function f, if all efficient algorithms, given $f(x)$, can guess $b(x)$ with success probability only negligibly better than half. Blum and Micali showed the importance of hard-core predicates in pseudorandom bit generation. Specifically, they showed that the modular exponentiation function over a field of prime cardinality, $f_{P,g}(x) = g^x \bmod P$, has a hard-core predicate, and used it in order to construct a pseudorandom bit generator. The study of hard-core predicates of $f_{P,g}$ has culminated in the work of Håstad and Näslund [HN], showing that all bits of $f_{P,g}$ are individually secure.

1.1. *Hard-Core Functions*

The concept of a hard-core function (or the simultaneous security of bits) is a generalization of hard-core predicates. Intuitively, a sequence of bits associated to a one-way function f is said to be simultaneously secure if no efficient algorithm can gain any information about the given sequence of bits in x, when only given $f(x)$. Proving the simultaneous security of a sequence of bits (rather than a single bit) in $f_{P,g}$ is a desirable result, enabling the construction of more efficient pseudorandom generators as well as improving other applications. However, the best known result regarding the simultaneous security of bits in $f_{P,g}$ is due to Long and Wigderson [LW], Kaliski [K2] and Peralta [P], who showed that $O(\log n)$ bits are simultaneously secure, where n is the length of the modulus P.

Stronger results were demonstrated when the modulus was taken to be a composite, thus allowing one to relate (simultaneous) hardness of bits in the argument to the factoring problem. Denote by $f_{N,g}$ the exponentiation modulo a composite function, defined as $f_{N,g}(x) = g^x \bmod N$, where N is an n-bit composite equal to the multiplication of two large primes and g is an element in the multiplicative group $\bmod N$. Håstad et al. showed that under the factoring intractability assumption, all the bits in $f_{N,g}$ are individually hard, and that the upper $\lceil n/2 \rceil$ bits and lower $\lceil n/2 \rceil$ bits are simultaneously hard [HSS].

In the same setting (and under the same assumption that factoring is hard), we show that no efficient algorithm can tell apart $f_{N,g}(r)$ from $f_{N,g}(R)$, where r is a random $\lceil n/2 \rceil$-bit string and R is a random n-bit string.[1] That is, one can work with an exponent x of half the length, and still obtain an element that "seems random" to all efficient algorithms. Note that all the cryptographic tools that use exponentiation in Z_N^* (and base their security on the discrete logarithm assumption) can greatly benefit from this fact, since the time consumed for exponentiation grows linearly with the length of the exponent (and is thus cut by a factor of two). Our result is in fact equivalent to the result of Håstad et al. [HSS] on the simultaneous hardness of the upper $\lceil n/2 \rceil$ bits of $f_{N,g}$. Nevertheless, we give an alternative proof for it while using some of their ideas and techniques. Our approach significantly simplifies the proof given in [HSS] and sheds a new light on it.

We mention that our work also implies that no efficient algorithm can tell apart $f_{N,g}(r \cdot 2^k)$ from $f_{N,g}(R)$, where r and R are as above and k is polynomial in n. For

[1] As a matter of fact, in the exact formulation of our result, R is uniformly distributed over the range of naturals smaller than the order of g (on the group Z_N^*). However, the above claim (with R uniformly distributed in $\{0, 1\}^n$) holds as well, as an implication of Theorem 3.4.

details, see [R3]. In particular, when taking k to be $\lceil n/2 \rceil$, this result is equivalent to the simultaneous hardness of the lower $\lceil n/2 \rceil$ bits of $f_{N,g}$.

Another implication of our work (to be further discussed below) is the construction of a pseudorandom bit generator based on the computational indistinguishability of $f_{N,g}(r)$ from $f_{N,g}(R)$. Our generator is somewhat more efficient than all previously known factoring-based pseudorandom generators.

1.2. *An Efficient Pseudorandom Generator*

The notion of a pseudorandom bit generator, introduced by Blum and Micali [BM], plays a central role in cryptography. It enables the user to expand a short random seed into a longer sequence of bits, that can be used in any efficient application instead of a truly random bit sequence. Blum and Micali presented a pseudorandom bit generator based on the discrete log problem. Using the fact that the exponentiation function over a field of prime cardinality has a hard-core predicate, they suggested an iterative generator that yields one bit of output per exponentiation. Furthermore, they conceived a general paradigm that constructs an iterative pseudorandom generator, given any length-preserving one-way permutation f, and a hard-core predicate b for f.

The Blum–Blum–Shub pseudorandom generator [BBS], hereafter referred to as the "BBS generator", is based on the above paradigm, taking f to be the modular squaring function, where the modulus N is a Blum integer.[2] Since, as shown by Rabin [R1], the problem of factoring N can be reduced to the problem of extracting square roots in the multiplicative group mod$N$, the function f is a one-way function assuming the intractability of factoring Blum integers. Additionally, Blum et al. showed that f induces a permutation over the set of quadratic residues in the multiplicative group mod$N$, and using the results of Alexi et al. [ACGS] and Vazirani and Vazirani [VV], this implies that the least significant bit constitutes a hard-core predicate for f. The BBS generator is by far more efficient than the Blum–Micali generator.[3] In particular, for every polynomial $P(\cdot)$, the BBS generator stretches an n-bit seed into a $P(n)$-bit pseudorandom string using $P(n)$ modular multiplications.

Another generator whose pseudorandomness is based on factoring, was suggested by Håstad, Schrift and Shamir [HSS], and will be referred to as the "HSS generator". The HSS generator relies on the simultaneous hardness of half of the bits in the exponentiation modulo a composite function $f_{N,g}$. Loosely speaking, the HSS generator takes an n-bit random seed x (where n is the size of the modulus N), and outputs $f_{N,g}(x)$ followed by the lower half of the bits of x.[4] Observe that from an n-bit seed, the HSS generator obtains $1.5n$ bits of output, using n modular multiplications on the worst case, and $0.5n$ modular multiplications on the average case (when assuming that the terms $g^{2^0}, \ldots, g^{2^n}$ are pre-computed together with the other parameters of the generator).

[2] A Blum integer is equal to the multiplication of two primes of equal size, each congruent to 3 mod 4.

[3] The Blum–Micali generator obtains each bit of output at the cost of one modular exponentiation that is implemented by n modular multiplications, as opposed to one modular multiplication per output bit needed by the BBS generator.

[4] As a matter of fact, in order to achieve true pseudorandomness, universal hashing is applied. The actual construction is presented in Section 4.

Even though our main result is equivalent to the simultaneous hardness of half of the bits in $f_{N,g}$, our result gives rise to a pseudorandom generator that is (in a sense) more natural than the HSS generator, as well as more efficient than it. Informally, we suggest a generator that takes a random seed x of size $\lceil n/2 \rceil$, and outputs $f_{N,g}(x)$. Observe that our generator doubles the length of its input. In particular, it obtains n bits of output from an $0.5n$-bit seed using $0.5n$ modular multiplications on the worst case, and $0.25n$ modular multiplications on the average case (once again, assuming that the terms $g^{2^0}, \ldots, g^{2^{\lceil n/2 \rceil}}$ are pre-computed).

The following table compares the three factoring-based generators discussed above, each having the same security parameter n (the size of the modulus N). Note that the "cost" column refers to the average number of multiplications done in every application of the generator, and the "amortized cost" column refers to the average number of multiplications divided by the number of additional output bits of the generator (i.e. the amortized cost is the cost divided by the difference between the output length and the seed length).[5]

	Seed length	Output length	Cost	Amortized cost
BBS construction	n	$P(n) \quad (\forall P)$	$P(n)$	$\frac{P(n)}{P(n)-n} \approx 1$
HSS construction	n	$1.5n$	$0.5n$	$\frac{0.5n}{1.5n-n} = 1$
Our construction	$0.5n$	n	$0.25n$	$\frac{0.25n}{n-0.5n} = 0.5$

We mention that our generator (as well as the HSS generator) has an efficient parallel implementation in time $O(\log n)$.[6] This is opposed to the BBS generator which is not known to have a fast parallel implementation (i.e. any faster than the straightforward sequential implementation).

Another pseudorandom generator that is related to our work was proposed independently by Genarro [G1]. This generator is proved to be secure assuming that discrete exponentiation modulo a prime is hard to invert even when the exponent is of poly-logarithmic length (this assumption is called Discrete-Log with Short Exponents; in short, the DLSE-Assumption). Gennaro's generator is more efficient than ours: given a prime modulus of size n, every l modular multiplication yields $n - l - 1$ bits of output, where l is poly-logarithmic in the size of n. However, Gennaro's generator is based on a strong and relatively new assumption (compared with the Factoring Assumption

[5] Above, we compare the various generators with respect to the same security parameter. Still, one might consider a comparison with respect to the same seed length. In order to do that we must normalize the input/output sizes of our generator so that its seed length will be n. Thus, the output produced by our generator will be of length $2n$, the cost will be $0.5n$ and the amortized cost will again be 0.5 multiplications per an additional output bit. Note, however, that the size of the security parameter in our construction will be twice its size in the BBS and the HSS constructions. Thus, our construction will be safer. On the other hand, each multiplication will involve twice as big numbers.

[6] The parallel implementation uses $\lceil n/2 \rceil$ processors $P_1, \ldots, P_{\lceil n/2 \rceil}$, where the input of each processor P_i is the ith bit of the seed, s_i, and the output is the multiplication of the values $g^{2^{i-1} \cdot s_i}$ contributed by each processor.

which has undergone years of extensive research). We mention that, as was shown in [OW], the DLSE-Assumption is not secure for a random prime modulus, so that one needs to restrict the primes to *safe primes*. Both generators (Gennaro's and ours) are based on a similar observation, except that ours is an immediate implication of our main result (Theorem 3.2) and Gennaro's generator is an immediate implication of the DLSE-Assumption.

1.3. *Organization*

The rest of this work is organized as follows. Basic definitions and notations are given in Section 2. In Section 3 we state and prove the main theorem (regarding the pseudorandomness of exponentiation with a short exponent), show its equivalence to the [HSS] result and discuss the difference between the two proofs. In Section 4 we address the issue of constructing a pseudorandom generator based on results such as ours and those in [HSS]. The issues addressed may be of general interest beyond the specific hard problem on which our construction is based (i.e. factoring).

2. Preliminaries

2.1. *General*

Probability ensembles. Let I be a countable index set. A **probability ensemble indexed by** I is a sequence of random variables indexed by I. Namely, $X = \{X_i\}_{i \in I}$, where the X_i's are random variables, is a probability ensemble indexed by I.

In our applications, we use $\mathbb{N}$ as an index set, and let each X_n (in an ensemble of the form $\{X_n\}_{n \in \mathbb{N}}$) range over strings of length n. In particular, we denote by U_n the random variable that is uniformly distributed over $\{0, 1\}^n$.

Statistical difference. A basic notion from probability theory is the *statistical difference* between probability ensembles $\{X_n\}_{n \in \mathbb{N}}$ and $\{Y_n\}_{n \in \mathbb{N}}$. The statistical difference measures the distance between distributions and is defined to be

$$SD(X_n, Y_n) = \frac{1}{2} \cdot \sum_{\alpha} |\Pr[X_n = \alpha] - \Pr[Y_n = \alpha]|.$$

Probability ensembles $\{X_n\}_{n \in \mathbb{N}}$ and $\{Y_n\}_{n \in \mathbb{N}}$ are called **statistically close** if their statistical difference is *negligible* in n (we say that a function $\mu\colon \mathbb{N} \to [0, 1]$ is **negligible** if for every positive constant c and all sufficiently large n's, $\mu(n) < 1/n^c$).

Computational indistinguishability. A weaker notion of closeness between probability ensembles is the notion of indistinguishability by all efficient algorithms. When no efficient algorithm (which may be probabilistic) can tell apart the two ensembles, we call them computationally indistinguishable. Formally,

Definition 2.1. We say that two ensembles $\{X_n\}_{n \in \mathbb{N}}$ and $\{Y_n\}_{n \in \mathbb{N}}$ are **computationally indistinguishable**, if for every probabilistic polynomial-time algorithm D, for every positive constant c and for all sufficiently large n's,

$$|\Pr[D(X_n, 1^n) = 1] - \Pr[D(Y_n, 1^n) = 1]| < \frac{1}{n^c}.$$

A notation. Let A be a finite set, then $a \in_R A$ denotes that the element a is uniformly chosen from the set A (i.e. each value is obtained with probability $1/|A|$).

2.2. *The Factoring Assumption*

We denote by N_n the set of all n-bit integers $N = P \cdot Q$, where P and Q are two odd primes of equal length. The collection N_n can be sampled efficiently. Specifically, given input 1^n, it is possible to pick a random element in N_n in polynomial time (using a polynomial number of coin tosses).

The problem of factoring integers is widely believed to be intractable. Integers belonging to the set N_n are considered to be particularly hard to factor. Note that N_n is a non-negligible fraction of all n-bit integers. Currently, the best algorithm known can factor an integer picked randomly from N_n in a (heuristic) running-time of $e^{1.92 n^{1/3} \log^{2/3} n}$ [LLMP]. We use the following assumption.

Assumption 1 (Factoring Assumption). *Let A be a probabilistic polynomial-time algorithm. Then, for every constant $c > 0$ and all sufficiently large n's,*

$$Pr[A(P \cdot Q) = P] < \frac{1}{n^c},$$

where $N = P \cdot Q$ is selected uniformly from N_n.

2.3. *The Group Z_N^**

For a composite N, denote by Z_N^* the multiplicative group that consists of all the naturals that are smaller than N and are relatively prime to it. We represent the elements in Z_N^* by binary strings of size $n = \lceil \log N \rceil$. Further notations we use are:

- Let $x < N$, and let $1 \le j \le i \le n$. We denote by x_i the ith bit in the binary representation of x, and by $x_{i,j}$ the substring of x including the bits from position j to position i.
- Denote by $ord_N(g)$ the order of an element g in Z_N^*; that is, the minimal $k \ge 1$ for which $g^k = 1 (\bmod N)$.
- Denote by $\langle g \rangle$ the subgroup of Z_N^* generated by g. That is, $\langle g \rangle$ is the set of all elements of the form $g^x \bmod N$ for some x.
- Denote by P_n the set of pairs $\langle N, g \rangle$ where $N \in N_n$ and $g \in Z_N^*$. Note that P_n is efficiently samplable.

We now define the *exponentiation modulo a composite* function and its inverse the *discrete logarithm modulo a composite* function.

Definition 2.2. Let $\langle N, g \rangle$ be a pair in P_n. We define the exponentiation modulo a composite function $f_{N,g}$: $\{0, 1\}^* \to \langle g \rangle$ to be $f_{N,g}(x) = g^x \bmod N$.

Definition 2.3. Let $\langle N, g\rangle$ be a pair in P_n. We define the discrete log modulo a composite function $DL_{N,g}\colon \langle g\rangle \to [0, ord_N(g))$, where $DL_{N,g}(y)$ is defined to be the unique natural $x < ord_N(g)$ for which $f_{N,g}(x) = y$.

3. Exponentiation with a Short Exponent Is Pseudorandom

We introduce two probability ensembles, which we show to be computationally indistinguishable assuming the intractability of factoring.

Definition 3.1. Let $\langle N, g\rangle$ be a uniformly distributed pair in P_n. Let R be uniformly distributed in $[0, ord_N(g))$ and let r be uniformly distributed in $\{0,1\}^{\lceil n/2\rceil}$. We denote by $Full_n$ the distribution $\langle N, g, g^R \bmod N\rangle$ and by $Half_n$ the distribution $\langle N, g, g^r \bmod N\rangle$.

Theorem 3.2. *Suppose that factoring is intractable. Then the ensembles* $\{Half_n\}_{n\in\mathbb{N}}$ *and* $\{Full_n\}_{n\in\mathbb{N}}$ *are computationally indistinguishable.*

3.1. *Proof Outline*

We use the hybrid technique in order to prove the indistinguishability of $Full_n$ and $Half_n$. We define hybrid distributions H_n^i and show that, assuming the intractability of factoring, for every $i \geq n/2$, the distributions H_n^i and H_n^{i+1} are computationally indistinguishable.

Definition 3.3. Let $\langle N, g\rangle$ be a uniformly distributed pair in P_n, and let x be uniformly distributed in $\{0,1\}^i$. We denote by H_n^i the distribution $\langle N, g, g^x \bmod N\rangle$ (see Fig. 1).

Clearly, the distribution $H_n^{\lceil n/2\rceil}$ is identical to $Half_n$. In Section 3.4 we show that the hybrid $H_n^{n+\omega(\log n)}$ is statistically close to $Full_n$. Thus, in order to establish Theorem 3.2, it is sufficient to prove the following result.

Theorem 3.4. *Suppose that Factoring is intractable. Then, for* $i \geq \lceil n/2\rceil$, *the distributions* H_n^i *and* H_n^{i+1} *are computationally indistinguishable.*

We begin the proof of Theorem 3.4 by restricting the hybrid distribution defined above to a specific choice of N and g. Specifically,

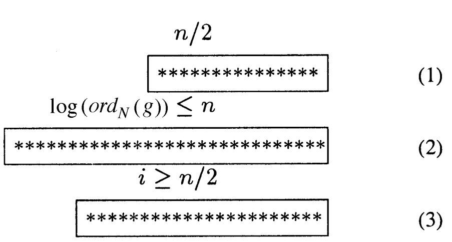

Fig. 1. We denote random bits by "*" and the length of the binary expansion of $ord_N(g)$ by m. Lines (1), (2) and (3) show the exponents of $Half_n$, $Full_n$ and the hybrid H_n^i, respectively.

Definition 3.5. Let $\langle N, g\rangle$ be any pair in P_n and let x be uniformly distributed in $\{0, 1\}^i$. We denote by $H^i_{N,g}$ the distribution $\langle g^x \bmod N\rangle$.

We prove that a polynomial-time algorithm D that distinguishes $H^i_{N,g}$ and $H^{i+1}_{N,g}$ with a non-negligible advantage can be transformed into a polynomial-time algorithm D' that is able to extract the discrete-log of elements of the form $g^x \bmod N$ where $|x| \leq i + 1$, with respect to these N and g. That is:

Lemma 3.6. *Let $\langle N, g\rangle$ be a pair in P_n and let i be polynomial in n. There exists an explicit transformation from a polynomial-time algorithm D that distinguishes $H^i_{N,g}$ and $H^{i+1}_{N,g}$ with a non-negligible advantage to a polynomial-time discrete-log finder D', that on input of the form $g^x \bmod N$, where $|x| \leq i+1$, finds x with overwhelming probability.*

The last ingredient in the proof of Theorem 3.4 relies on the connection between factoring N and computing the discrete logarithm modulo N. It is well known that in order to factor N it is sufficient to compute $S = DL_{N,g}(g^N \bmod N)$, for a random $g \in Z^*_N$. An important feature of S is that its size (denoted $|S|$) is relatively small, specifically, $|S| = \lceil n/2\rceil + 1 = \lceil \log N\rceil/2 + 1$ (see Section 3.2 for further details). Using Lemma 3.6, contradiction to Theorem 3.4 implies the ability to compute S and thus factor N.

3.2. *Factoring versus Discrete Logarithm in Z^*_N*

As mentioned above, there is a tight connection between factoring N and revealing the discrete logarithm of a suitably chosen element in Z^*_N. Specifically, in order to factor a random integer $N = P \cdot Q$ in N_n, it is sufficient to find the discrete log of $g^N \bmod N$ for a randomly chosen $g \in Z^*_N$. This is due to the following trivial fact:

Fact 1. *Let $\langle N, g\rangle$ belong to P_n (say that $N = P \cdot Q$). Then, if $ord_N(g) > P + Q - 1$, the discrete logarithm $S \overset{\text{def}}{=} DL_{N,g}(g^N)$ is equal to $P + Q - 1$.*

Proof. Recall that the order of g divides the order of the group Z^*_N, which equals $\varphi(N) = (P-1)(Q-1)$. Therefore, $g^N \equiv g^{N-\varphi(N)} \equiv g^{P+Q-1} (\bmod N)$. Consequently, if $ord_N(g) > P + Q - 1$, then $S = P + Q - 1$. □

The following proposition, established by Håstad et al. [HSS], asserts that an element chosen uniformly in Z^*_N is very likely to be of high order:

Proposition 3.7 (Håstad et al.). *Let $\langle N, g\rangle$ be uniformly distributed in P_n, where $N = P \cdot Q$. Then, for every k,*

$$\Pr\left[ord_N(g) < \frac{1}{n^k} \cdot (P-1)(Q-1)\right] \leq O\left(\frac{1}{n^{(k-4)/3}}\right).$$

The only use we make of the above proposition, is to show that with very high probability, $ord_N(g)$ cannot be too small. Specifically, Proposition 3.7 implies that with overwhelming probability $ord_N(g)$ is greater than $P + Q - 1$. Therefore, as was first observed by Chor [C], if we obtain S, then we can solve the two equations $P + Q - 1 = S$ (according to Fact 1) and $P \cdot Q = N$ for the unknowns P and Q and thus factor N.

3.3. *Transforming a Hybrid Distinguisher into a Discrete-Log Finder*

The proof of Lemma 3.6 is basically a reduction. We show how to use the algorithm D that distinguishes $H^i_{N,g}$ and $H^{i+1}_{N,g}$ as an oracle that gives us information on the $(i + 1)$st bit of an $(i + 1)$-long exponent. We then show how to manipulate g^x in order to extract x.

3.3.1. *Using D to Discover the $(i + 1)$st Bit of the Exponent*

Assume that for some non-negligible ε the following holds:

$$\begin{aligned} &|\Pr[D(N, g, g^x) = 1 \mid x \in_R \{0, 1\}^i] \\ &\quad - \Pr[D(N, g, g^x) = 1 \mid x \in_R \{0, 1\}^{i+1}]| \geq \varepsilon. \end{aligned} \tag{1}$$

Observe that

$$\begin{aligned} &\Pr[D(N, g, g^x) = 1 \mid x \in_R \{0, 1\}^{i+1}] \\ &\quad = \tfrac{1}{2} \cdot \Pr[D(N, g, g^x) = 1 \mid x \in_R \{0, 1\}^i] \\ &\quad\quad + \tfrac{1}{2} \cdot \Pr[D(N, g, g^{2^i+x}) = 1 \mid x \in_R \{0, 1\}^i]. \end{aligned} \tag{2}$$

From (1) and (2) we obtain the following:

$$\begin{aligned} &|\Pr[D(N, g, g^x) = 1 \mid x \in_R \{0, 1\}^i] \\ &\quad - \Pr[D(N, g, g^{2^i+x}) = 1 \mid x \in_R \{0, 1\}^i]| \geq 2\varepsilon. \end{aligned} \tag{3}$$

Denote by $\bar{H}^i_{N,g}$ the distribution g^{2^i+x} where x is drawn uniformly from $\{0, 1\}^i$. Another way to state inequality (3) is to say that the distinguisher D has advantage at least 2ε in distinguishing the distributions $H^i_{N,g}$ and $\bar{H}^i_{N,g}$. Let β and γ be the acceptance probabilities of D on input taken from $H^i_{N,g}$ and $\bar{H}^i_{N,g}$, respectively. That is, let

$$\beta \overset{\text{def}}{=} \Pr[D(N, g, g^x) = 1 \mid x \in_R \{0, 1\}^i] \tag{4}$$

and

$$\gamma \overset{\text{def}}{=} \Pr[D(N, g, g^{2^i+x}) = 1 \mid x \in_R \{0, 1\}^i]. \tag{5}$$

Without loss of generality assume that $\gamma > \beta$. Note that γ (respectively β) denotes the probability that the "1-answer" given by D is correct (respectively wrong); i.e. the $(i + 1)$st bit is indeed 1. A good approximation of β and γ can be easily obtained (in polynomial time) by performing a priori tests on D, using samples taken from $H^i_{N,g}$ and $\bar{H}^i_{N,g}$.

In what follows we will use the distinguisher D as an oracle, that enables us to "peek" into a 1-bit window on the $(i + 1)$st location of an unknown exponent of length $(i + 1)$. Specifically, we will use D in order to derive the $(i + 1)$st bit of an $(i + 1)$-bit string x, given g^x. This is straightforward in case $\gamma = 1$ and $\beta = 0$, but we will show how to extract valuable information also in the general case of $\gamma > \beta$.

3.3.2. *Discovering x: a Naive Implementation*

Suppose for a moment that we had a "perfect" oracle, that given input $Z = g^x$, where x is of length $(i+1)$, would supply us, with success probability 1, the $(i+1)$st bit of x. It would then enable us to extract x, using two simple operations:

Shifting to the left: By squaring Z we shift x by one position to the left.
Zeroing the *j*th bit: By dividing Z by $g^{2^{j-1}}$ we zero the jth position in x, in case it is known to be 1.

Therefore, we can extract x from the most significant to the least significant bit by "moving" it under the $(i+1)$st window. Specifically, we query the oracle and determine the $(i+1)$st bit of x and zero it in case it equals 1. Next, we shift x by one position to the left, query again the oracle to discover the next bit, and so on.

However, as the oracle might give us erroneous answers and all we are guaranteed is that there is a $\gamma - \beta$ gap (which is greater than 2ε) between the probability to get a correct 1-answer and the probability to get an erroneous 1-answer, our implementation needs to be more careful.

When dealing with a general oracle, we randomize our queries to the oracle and learn the correct answer by comparing the proportion of 1-answers with β and γ. A straightforward way to learn the $(i+1)$st bit of x given $Z = g^x$, would be to query the oracle on polynomially many random multiples $Z \cdot g^{r_k}$ for known r_k's chosen uniformly from $\{0,1\}^i$, and based on the fraction of 1-answers to decide between 0 and 1. However, this approach fails, since despite our knowledge of r_k, we cannot tell whether a carry from the addition of the i least significant bits of the known r_k and the unknown x effects the $(i+1)$st bit of their sum. Thus we cannot gain any information on the $(i+1)$st bit of x from the answer of the oracle on $Z \cdot g^{r_k}$.

3.3.3. *Discovering x: the Actual Implementation*

We now give a rough description of a procedure that resolves this difficulty and computes x. Let us make our life easier and assume for start that x is of length $i - m$, where m is logarithmic in n (say, $m = \lceil \log n/\varepsilon \rceil$).[7] We first show how to extract discrete logs of exponents x of that size.

Let l be an index going down from $i-m$ to 1. The procedure consists of $i-m$ stages (a stage for each value of l), where on each stage we create a list L_l which is a subset of $\{0, \ldots, 2^{i-m-l+1}-1\}$ such that L_l will contain candidates for the $|x|-l+1 = i-m-l+1$ most significant bits of x. We want two invariants to hold for the list L_l:

1. L_l contains an element e such that $x - e \cdot 2^l$ belongs to the set $\{0, \ldots, 2^l - 1\}$. In other words, we want e to equal $x_{i-m,l}$ (i.e. $x_{i-m} \cdots x_{l+1} x_l$).
2. The size of L_l is small; that is, it contains up to a polynomial number of values (where the polynomial is set a priori).

Thus, on the last stage (when the index l equals 1), we will have a list of polynomial size that contains x.

[7] We assume that $i \geq m$, since for $i < m$ we can find x simply by trying all possibilities.

The values in each list are kept sorted. The transition from the $(l+1)$st list to the lth list is done as follows: We first let L_l contain all the values v such that $v = 2u$ or $v = 2u + 1$ where u is in L_{l+1}, thus making the size of L_l twice the size of L_{l+1}. Obviously, by this we maintain the first invariant specified above. In case the size of L_l exceeds the polynomial bound we fixed, we use repeatedly a *Trimming Rule* (to be described next) in order to throw candidates out of L_l until we are within the maximal size allowed. We stress that the Trimming Rule never throws away the correct candidate (i.e. $x_{i-m,l}$).[8]

3.3.4. *Keeping the Size of L_l Bounded*

Suppose that we decide to trim L_l whenever the difference between the largest candidate in it, denoted by $v^l_{\max}$, and the smallest candidate in it, denoted by $v^l_{\min}$, exceeds a certain polynomial, say 2^m (recall that $m = \lceil \log n/\varepsilon \rceil$ and since ε is non-negligible, we have that 2^m is polynomial in n). At least one of the values $v^l_{\max}$ and $v^l_{\min}$ is not the correct value $x_{i-m,l}$. Therefore, the Trimming Rule (to be defined in what follows) may throw one of them out of the list. For this purpose, we define a new target x' (see Section 3.3.5), for which $g^{x'}$ can be efficiently computed given $Y = g^x$, $v^l_{\max}$ and $v^l_{\min}$. We will examine a certain position in it (that is dependent on l), henceforth referred to as the *crucial position* (shortly denoted *cp*). Essentially, x' will have the following properties:

1. If $v^l_{\min}$ is the correct candidate (i.e. $x_{i-m,l} = v^l_{\min}$), then the cp-bit in x' is 0, so are m bits to its right and so are all the bits to its left.
2. If $v^l_{\max}$ is the correct candidate (i.e. $x_{i-m,l} = v^l_{\max}$), then the cp-bit in x' is 1, the m bits to its right are all 0's and so are all the bits to its left.

Consequently, in these two situations we will be able to perform the randomization we wanted. We first shift x' to the left until the cp-bit is placed in the $(i + 1)$st location (by repeated squaring). We then multiply the result by g^r for some randomly chosen $r \in \{0, 1\}^i$. The probability to have a carry into the $(i+1)$st location from the addition of r and the shifted x', is no more than $1/2^m$ (a carry might occur only when $r_{i,i-m+1} = 11 \cdots 1$). Hence, by using a polynomial number of queries to the oracle (with independently chosen r's) we are able to deduce the value of the cp-bit by comparing the fraction of 1-answers with β and γ.

As the value of the cp-bit is revealed, we can discard one of the candidates $v^l_{\min}$ or $v^l_{\max}$ from the list: if the value of the cp-bit is 1, then we are guaranteed that $v^l_{\min}$ is not correct, and if the value of the cp-bit is 0, then we are guaranteed that $v^l_{\max}$ is not correct.

Note that in case neither $v^l_{\max}$ nor $v^l_{\min}$ are correct, we cannot ensure that m bits to the right of the cp-bit in x' will be 0's, so a carry may reach the $(i + 1)$st position. Thus we can get the frequency of 1-answers altogether different from β and γ. Yet in that case, it is OK for the Trimming Rule to discard either one of the extreme values from the list.

We proceed with a formal presentation of the proof.

[8] The Trimming Rule and its analysis are exactly the same as in [HSS].

3.3.5. *Definition of x' and cp*

In order to trim the list L_l, we define the new secret x' (which is a function of l, x, $v^l_{\min}$ and $v^l_{\max}$) to be

$$x' = \left\lceil \frac{2^{2m}}{v^l_{\max} - v^l_{\min}} \right\rceil \cdot (x - v^l_{\min} \cdot 2^l).$$

Note that $g^{x'}$ can be efficiently evaluated given $Y = g^x$, $v^l_{\min}$ and $v^l_{\max}$. The crucial position in x' is defined to be

$$cp = l + 2m + 1$$

Recall that $l \le i - m$ and so $cp \le i + m + 1$. However, we need to have $cp \le i + 1$, whenever we apply the Trimming Rule. This can be assured, because we trim L_l whenever the difference between the extreme values in it exceeds 2^m, which happens only for indices $l \le i - 2m$. Thus, we have that $cp \le 2m + i - 2m + 1 = i + 1$.

3.3.6. *The Actual Algorithms and Their Analysis*

We first describe the procedure "Find x" that on input $\langle N, g\rangle \in P_n$, index i (the index of the hybrid for which the acceptance probability of D on $H^i_{N,g}$ and $\bar{H}^i_{N,g}$ differs by more than 2ε) and $Y = g^x$, where $|x| = i - m$, finds x. We proceed with an analysis of the procedure, which leads us to the exact formulation of the Trimming Rule.

```
Procedure "Find x"
On input $\langle N, g\rangle \in P_n$, index $i$ and $Y = g^x$ where $|x| = i - m$,
execute the following steps:
1. Let $L_{i-m} = \{0, 1\}$.
2. For $l = i - m - 1$ to 1 do the following:
   (a) Let $L_l \overset{\text{def}}{=} \{2u, 2u + 1\colon\ u \in L_{l+1}\}$.
       Order the resulting list from the largest element $v^l_{\max}$ to the smallest
       element $v^l_{\min}$.
   (b) If $v^l_{\max} - v^l_{\min} > 2^m$ (we are guaranteed that $v^l_{\max} - v^l_{\min} \le 2 \cdot 2^m$ by
       the previous stage) use the Trimming Rule (to be specified) repeatedly
       until the difference between the largest element in the list and the
       smallest one is no more than $2^m$.
3. Check all values $v \in L_1$ and see whether $g^v$ equals $Y$. If such a value is
   found, then it is $x$.
```

Two facts. We make two observations which lead us to the formulation of the rule by which we trim L_l (assuming that $2^m < v^l_{\max} - v^l_{\min} \le 2^{m+1}$):

Fact 2. *Suppose that $v^l_{\min}$ indeed equals $x_{i-m,l}$. Then the cp-bit in x' is* 0, *all the bits to its left are* 0*'s and $m - 1$ bits to its right are* 0*'s as well.*

Fact 3. *Suppose that $v^l_{\max}$ indeed equals $x_{i-m,l}$. Then the cp-bit in x' is* 1, *all the bits to its left are* 0*'s and so are $m - 2$ bits to its right.*

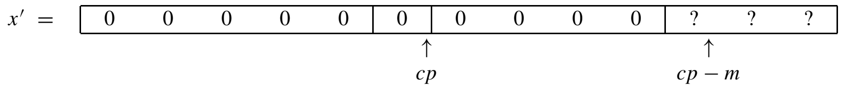

Fig. 2. The structure of x' when $v^l_{\min}$ equals $x_{i-m,l}$.

Proof of Fact 2. Using $v^l_{\max} - v^l_{\min} > 2^m$, observe that

$$\begin{aligned} x' &= \left\lceil \frac{2^{2m}}{v^l_{\max} - v^l_{\min}} \right\rceil \cdot (v^l_{\min} \cdot 2^l + x_{l,1} - v^l_{\min} \cdot 2^l) \\ &\le 2^m \cdot x_{l,1} \\ &\le 2^{m+l} \\ &= 2^{cp-m-1}. \end{aligned}$$

(See Fig. 2.) □

Proof of Fact 3. Observe that

$$\begin{aligned} x' &= \left\lceil \frac{2^{2m}}{v^l_{\max} - v^l_{\min}} \right\rceil \cdot (v^l_{\max} \cdot 2^l + x_{l,1} - v^l_{\min} \cdot 2^l) \\ &= \left\lceil \frac{2^{2m}}{v^l_{\max} - v^l_{\min}} \right\rceil \cdot ((v^l_{\max} - v^l_{\min}) \cdot 2^l + x_{l,1}) \\ &= 2^{2m+l} + \delta \cdot (v^l_{\max} - v^l_{\min}) \cdot 2^l + \left\lceil \frac{2^{2m}}{v^l_{\max} - v^l_{\min}} \right\rceil \cdot x_{l,1}, \end{aligned}$$

where

$$\delta = \left\lceil \frac{2^{2m}}{v^l_{\max} - v^l_{\min}} \right\rceil - \frac{2^{2m}}{v^l_{\max} - v^l_{\min}} \in [0, 1).$$

Let $U_1 = \delta \cdot (v^l_{\max} - v^l_{\min}) \cdot 2^l$ and let

$$U_2 = \left\lceil \frac{2^{2m}}{v^l_{\max} - v^l_{\min}} \right\rceil \cdot x_{l,1}.$$

We can write x' as

$$x' = 2^{cp-1} + U_1 + U_2.$$

Recall that $2^m < v^l_{\max} - v^l_{\min} \le 2^{m+1}$. Therefore, $U_1 \le 2^{m+1+l} = 2^{cp-m}$ and $U_2 \le 2^{m+l} = 2^{cp-m-1}$. Also, both $U_1, U_2 \ge 0$. Thus, $U_1 + U_2 \in [0, 2^{cp-m+1})$. Consequently x' is of the following form (see Fig. 3):

- The cp-bit in x' is 1 and all the bits to its left are 0's.
- The $m - 2$ bits to the right of the cp-bit in x' are 0's. □

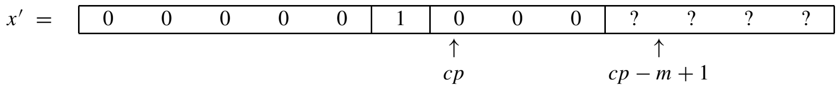

Fig. 3. The structure of x' when $v^l_{\max}$ equals $x_{i-m,l}$.

We now formally state the Trimming Rule:

Trimming Rule (on the current list L_l with $v^l_{\min}$ and $v^l_{\max}$)

1. Compute $Y' = g^{x'} = (Y \cdot g^{-v^l_{\min} \cdot 2^l})^e$ where $Y = g^x$ and $e = \lceil 2^{2m}/(v^l_{\max} - v^l_{\min})\rceil$.
2. Shift x' by $i + 1 - cp$ bits to the left (by computing $Y'' = (Y')^{2^{i+1-cp}}$), thus placing the crucial position (i.e. cp) in x' on location $i + 1$.
3. Pick $t(n) = n^4/\varepsilon^2$ random elements $r_1, \ldots, r_{t(n)} \in \{0, 1\}^{i-m}$.
4. For each $1 \leq k \leq t(n)$ query the oracle on $Y'' \cdot g^{r_k} (\bmod N)$ and denote by b_k its answer (i.e. $b_k = D(g^{x' \cdot 2^{i+1-cp}+r_k})$). Denote by M the mean $\sum_{k=1}^{t(n)} b_k/t(n)$.
5. If $M \leq (\beta + (\gamma - \beta)/2)$ discard the candidate value $v^l_{\max}$ from the list L_l. Otherwise (i.e. when $M > (\beta + (\gamma - \beta)/2)$) discard the candidate value $v^l_{\min}$.

Using the Chernoff bound one can show that the error probability of the Trimming Rule (i.e. the probability that the correct value will be discarded from the list) is exponentially small (for the exact proof see our technical report [GR]). Note that for every index l, the transition from the $(l + 1)$st list to the lth list is done in polynomial time, and thus the procedure "Find x" together with the Trimming Rule runs in polynomial time. Thus, we have showed how to extract the discrete-log of exponents x of size $i - m$.

In order to *complete the proof of Lemma* 3.6, we still need to explain how can we extract the discrete-log of exponents x of size $i + 1$. Remember that m was chosen to be logarithmic in n (i.e. $m = \lceil \log n/\varepsilon \rceil$). We can simply try all combinations for the $m + 1$ most significant bits of x. For each such combination try to zero these bits in x and run the procedure "Find x". For one of these combinations this will work and x will be found in the list L_{i-m}. Since "Find x" runs in polynomial time and since the number of combinations 2^{m+1} is polynomial, x can be derived with overwhelming probability in polynomial time. □

3.3.7. *Proof of Theorem* 3.4

We now show that the hybrids H^i_n and H^{i+1}_n are indistinguishable assuming intractability of factoring. Specifically, Theorem 3.4 can be derived from the following claim:

Claim 3.7.1. *Suppose that the gap between the acceptance probability of D on the hybrids H^i_n and H^{i+1}_n is greater than ε. Then, with probability at least $\varepsilon/8$, we can factor a composite N, uniformly distributed in N_n.*

Proof. Let $W_n \subseteq P_n$ be the set of pairs $\langle N, g\rangle$ in P_n for which it holds that D distinguishes $H^i_{N,g}$ and $H^{i+1}_{N,g}$ with advantage at least $\varepsilon/2$. For a pair $\langle N, g\rangle$ uniformly chosen from P_n (where N is equal to $P \cdot Q$) the following two facts hold:

1. With probability greater than $\varepsilon/2$, the pair $\langle N, g\rangle$ belongs to the set W_n.
2. With overwhelming probability $ord_N(g) > P + Q - 1$.

The first fact can be established from a standard averaging argument and the second fact from Proposition 3.7. Therefore, given a *random* $N = P \cdot Q$ in N_n, we can pick g uniformly in Z^*_N and with probability higher than $\varepsilon/4$ both of the above conditions hold (i.e. $\langle N, g\rangle \in W_n$ and $ord_N(g) > P + Q - 1$). Recall now that factoring N is possible by finding the discrete-log of $S = DL_{N,g}(g^N)$. According to Fact 1, we have that $S = P + Q + 1$, so by revealing S we can factor N. Since $\langle N, g\rangle \in W_n$, we can extract discrete-logs of elements of the form $g^x \bmod N$, where $x \in \{0,1\}^{i+1}$, with overwhelming probability (using Lemma 3.6). Thus, with probability at least $\varepsilon/8$ (over the uniform distribution of $N \in N_n$ and our algorithm's coin tosses), we extract S (using the fact that the size of S is $\lceil n/2\rceil + 1 \leq i + 1$) and factor N. □

Remark. In fact, Theorem 3.4 holds even for i's such that $i \geq \lceil n/2\rceil - O(\log n)$ (rather than $i \geq \lceil n/2\rceil$). Given such $i < \lceil n/2\rceil$, we try all combinations for the $m + 1 + \lceil n/2\rceil - i$ most significant bits of S (instead of just $m + 1$ such bits). The total number of combinations remains polynomial in n, so this leaves us with polynomial running time.

3.4. *Proof of the Main Theorem*

We now turn back to Theorem 3.2, and prove it using Theorem 3.4.

Recall how the distributions $Half_n$ and $Full_n$ were defined (see Definition 3.1). Clearly, the hybrid $H_n^{\lceil n/2\rceil}$ is identical to $Half_n$. Note that the distribution $H_n^{n+\omega(\log n)}$ is statistically close to $Full_n$, as asserted by the following claim:

Claim 3.7.2. *The distributions $Full_n$ and $H_n^{n+\omega(\log n)}$ are statistically close.*

Proof. Let M denote $2^{n+\omega(\log n)}$. M can be written as $k \cdot ord_N(g) + r$ where k is an integer and $0 \leq r < ord_N(g)$. We now calculate the statistical difference between the distributions $Full_n$ and $H_n^{n+\omega(\log n)}$. Note that the first equality below is implied from the fact that in $f_{N,g}(x)$ the exponent x is reduced modulo $ord_N(g)$:[9]

$$\begin{aligned} SD(Full_n, H_n^{n+\omega(\log n)}) &= \frac{1}{2}\sum_{\alpha\in[0,ord_N(g))} |\Pr[Full_n = \alpha] - \Pr[H_n^{n+\omega(\log n)} = \alpha]| \\ &= \frac{1}{2}\left[r \cdot \left|\frac{1}{ord_N(g)} - \frac{k+1}{M}\right| + (ord_N(g) - r)\right. \\ &\qquad \left. \cdot \left|\frac{1}{ord_N(g)} - \frac{k}{M}\right|\right]. \end{aligned}$$

[9] Thus, for each element $0 \leq x < r$ there will be $k + 1$ congruent elements (modulo $ord_N(g)$) in the range $[0 \cdots M - 1]$, and for each element $r \leq x < ord_N(g)$ there will be k congruent elements in $[0 \cdots M - 1]$.

Using the fact that

$$\frac{k}{M} \le \frac{1}{ord_N(g)} \le \frac{k+1}{M}$$

we may remove the absolute values and rewrite the above equation as

$$\begin{aligned}
&\frac{1}{2}\left[r \cdot \left(\frac{k+1}{M} - \frac{1}{ord_N(g)}\right) + (ord_N(g) - r) \cdot \left(\frac{1}{ord_N(g)} - \frac{k}{M}\right)\right] \\
&\quad = \frac{1}{2}\left[(ord_N(g) - 2r) \cdot \frac{r}{M \cdot ord_N(g)} + \frac{r}{M}\right] \\
&\quad \le \frac{r}{M},
\end{aligned}$$

where the inequality uses $r \ge 0$. Since

$$\frac{r}{M} < \frac{N}{M} \le \frac{2^n}{2^{n+\omega(\log n)}},$$

we have that $SD(Full_n, H_n^{n+\omega(\log n)})$ is negligible in n. □

Consequently, if there exists a probabilistic polynomial-time algorithm D, that distinguishes the ensemble $Half_n$ from $Full_n$, then D distinguishes (almost) as well $Half_n$ from $H_n^{1.5n}$.

Assume now that the gap between the acceptance probability of D on the extreme hybrids $H_n^{\lceil n/2 \rceil}$ and $H_n^{1.5n}$ is greater than δ (where δ is non-negligible). We construct an algorithm A that factors integers uniformly distributed in N_n. On input N, algorithm A picks a random i in $\{\lceil n/2 \rceil, \dots, 1.5n\}$ and runs the procedure "Find x" on (N, i). By Claim 3.7.1, the probability that "Find x" indeed factors N, is greater than one-eighth of the gap between the acceptance probabilities of D on H_n^i and H_n^{i+1}, for a random i as above. Observing that the expected gap between the ith and $(i+1)$st hybrids is at least δ/n, we derive a contradiction to Assumption 1. □

Remark. The above proof actually establishes the indistinguishability of $Half_n$ and $H_n^{l(n)}$ for any $l(n) = n + \omega(\log n)$ such that $l(n) \le \text{poly}(n)$.

3.5. *Equivalence to the HSS Result*

Theorem 3.2 is actually equivalent to the result by Håsted et al. [HSS] on the simultaneous hardness of the upper $\lceil n/2 \rceil$ bits in the exponentiation function $f_{N,g}$. In order to show that, we discuss first an alternative version of Theorem 3.2. Recall the hybrid $H_n^{l(n)}$ defined in the proof of Theorem 3.2, including triplets $\langle N, g, g^R \rangle$, where $\langle N, g \rangle$ is uniformly distributed in P_n and R is uniformly distributed in $\{0, 1\}^{l(n)}$. For any function $l\colon \mathbb{N} \to \mathbb{N}$ such that $l(n) \in \{n + \omega(\log n), \dots, \text{poly}(n)\}$ (e.g. $l(n) = 1.5n$), we denote $H_n^{l(n)}$ by $\widetilde{Full}_n$. The following is a corollary from Theorem 3.2 and from Claim 3.7.2.

Corollary 3.8. *The probability ensembles* $\{Half_n\}_{n \in \mathbb{N}}$ *and* $\{\widetilde{Full}_n\}_{n \in \mathbb{N}}$ *are computationally indistinguishable.*

We show that Corollary 3.8 is equivalent to the result of Håsted et al. [HSS]. However, first, we give the exact formulation of their result.

Definition 3.9. Let $\langle N, g\rangle$ be uniformly distributed in P_n, let x be uniformly distributed in $\{0,1\}^{l(n)}$ and let r be uniformly distributed in $\{0,1\}^{\lceil n/2\rceil}$. We define the following probability distributions:

$$X_n \overset{\text{def}}{=} \langle N, g, f_{N,g}(x), x_{l(n),\lceil n/2\rceil}\rangle$$

and

$$Y_n \overset{\text{def}}{=} \langle N, g, f_{N,g}(x), r\rangle.$$

Theorem 3.10 (Håstad et al.). *The probability ensembles $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$ are computationally indistinguishable.*[10]

3.5.1. *The Equivalence*

Theorem 3.11. *Theorem* 3.10 *holds if and only if Corollary* 3.8 *holds.*

Proof. We show how to transform a probabilistic polynomial-time algorithm D that distinguishes the ensemble $\{X_n\}$ from $\{Y_n\}$ into a probabilistic polynomial-time algorithm D' that distinguishes the ensemble $\{Half_n\}$ from $\{\widetilde{Full}_n\}$, and vice versa.

Transforming D into D'. On input $\langle N, g, y\rangle$, pick z uniformly from $\{0,1\}^{l(n)-\lceil n/2\rceil}$ and run D on $\langle N, g, y\cdot g^{z\cdot 2^{\lceil n/2\rceil}}, z\rangle$. Return D's answer as output. Observe that:

1. If $\langle N, g, y\rangle$ is taken from $Half_n$, then $y = g^r$ where $r \in \{0,1\}^{\lceil n/2\rceil}$. Therefore, we have that $\langle N, g, g^{z\cdot 2^{\lceil n/2\rceil}+r}, z\rangle$ is distributed as X_n.
2. If $\langle N, g, y\rangle$ is taken from $\widetilde{Full}_n$, then $y = g^R$, where $R \in \{0,1\}^{l(n)}$. Let U_m denote a uniformly distributed m-bit string and let $\approx$ denote statistical closeness. Then

$$(U_{l(n)} + z\cdot 2^{\lceil n/2\rceil}) \bmod ord_N(g) \;\approx\; U_{l(n)} \bmod ord_N(g).$$

   Therefore, $\langle N, g, g^{z\cdot 2^{\lceil n/2\rceil}+R}, z\rangle$ is statistically close to Y_n.

Thus, Theorem 3.10 is implied by Corollary 3.8.

Transforming D' into D. On input $\langle N, g, y, z\rangle$, run D' on $\langle N, g, y/g^{z\cdot 2^{\lceil n/2\rceil}}\rangle$ and output D''s answer. Observe that:

1. If $\langle N, g, y, z\rangle$ is taken from X_n, then $y = f_{N,g}(x) = g^x$ and $z = x_{l(n),\lceil n/2\rceil}$. Therefore, $y/g^{z\cdot 2^{\lceil n/2\rceil}} = g^{x_{\lceil n/2\rceil,1}}$ and thus $\langle N, g, y/g^{z\cdot 2^{\lceil n/2\rceil}}\rangle$ is uniformly distributed in $Half_n$.

[10] Actually, the definition of simultaneous hardness of the upper $\lceil n/2\rceil$ bits of $f_{N,g}$ is wrong in [HSS]. Håsted et al.'s definition states that the two distributions $\langle \tilde{x}_{n,\lceil n/2\rceil}, Z\rangle$ and $\langle r, Z\rangle$ are computationally indistinguishable, where $Z = g^x$ (for $x \in_R Z_N^*$), $\tilde{x} = DL_{N,g}(Z)$ and $r \in_R \{0,1\}^{\lceil n/2\rceil}$. However, this definition is flawed: at least the most significant bit in the first distribution, $\tilde{x}_n$, will always be 0, since $ord_N(g)$ is always smaller than $N/2$. Hence the above two distributions can be easily distinguished.

2. If $\langle N, g, y, z\rangle$ is taken from Y_n, then $y = f_{N,g}(x) = g^x$ and z is independent of x. Note that

$$(U_{l(n)} - z \cdot 2^{\lceil n/2\rceil}) \bmod ord_N(g) \approx U_{l(n)} \bmod ord_N(g).$$

Therefore, $\langle N, g, y/g^{z\cdot 2^{\lceil n/2\rceil}}\rangle$ is statistically close to $\widetilde{Full}_n$.

Thus, Theorem 3.10 implies Corollary 3.8. □

Further discussion of the above equivalence is given in Section 3.3.3 of [R3].

3.5.2. *Discussion*

Our proof of Theorem 3.2 simplifies to a great extent the proof given in [HSS] to Theorem 3.10. The main reason for this is that we use an oracle that distinguishes between the hybrids H_n^i and H_n^{i+1}, instead of an oracle that predicts the ith bit of x, given $f_{N,g}(x)$. Unlike [HSS], we can use i's as big as we want, avoid the difficulties with biased bits and eliminate the dependence on the order of g in the basic procedure (Lemma 3.6). More specifically:

- Unlike in [HSS], we do not require that the order of g in Z_N^* be very high (i.e. greater than $n^{-k} \cdot (P-1)(Q-1)$). Theorems 3.2 and 3.4 only require that the order of g be greater than $P+Q-1$, while for Lemma 3.6 the order of g is irrelevant.
- We do not need to consider separately the $O(\log n)$ most significant bits as done in [HSS] (where a very complex proof is given for these bits).
- As a consequence from the different nature of the oracles, the randomization conducted by us (randomizing the bottom i bits) is different from the randomization done in [HSS] (randomizing the full range $[0, ord_N(g))$). Therefore many of the difficulties encountered in [HSS] are not relevant in our proof. For example, we do not need to avoid a wrap around the order of g.

4. Application to Pseudorandom Generators

An immediate application of Theorem 3.2 yields an efficient factoring-based pseudorandom generator which nearly doubles the length of its input. An additional tool used is a construction by Goldreich and Wigderson of a tiny family of functions which has good extraction properties [GW]. We also discuss how the parameters of the generator (a composite $N \in N_n$ and an element $g \in Z_N^*$) can be chosen in a randomness-efficient way (which is polynomial time). In particular, we present a method of choosing a random n-bit prime using only a linear number of random bits. This translates to a hitting problem which can be solved efficiently using methods described in [G3].

4.1. *Our Construction versus the HSS Construction*

Looking at Theorem 3.2, the first construction that comes to mind is a "pseudorandom generator" that takes a seed r of length $\lceil n/2\rceil$ and outputs $g^r \bmod N$ (for a fixed pair $\langle N, g\rangle$ in P_n). However, the output of the above so-called "pseudorandom generator" is not really pseudorandom. Even though it is computationally infeasible to distinguish between it and the distribution $g^R \bmod N$ for a random R in $[0, ord_N(g))$, we are not guaranteed that it cannot be easily told apart from the uniform distribution on n-bit strings. The same applies for a "pseudorandom generator" implied directly by Theorem 3.10, which takes a seed x of length n, and outputs $g^x \bmod N$ followed by $x_{\lceil n/2 \rceil, 1}$ (again, for fixed $\langle N, g \rangle$ in P_n).

Denote by $Half_{N,g}$ the distribution $g^r \bmod N$, where r is uniformly distributed over strings of length $\lceil n/2 \rceil$, and by $Full_{N,g}$ the distribution $g^R \bmod N$, where R is uniformly distributed over $[0, ord_N(g))$. Observe that the "amount of randomness" that $Full_{N,g}$ encapsulates in it is high, in the sense that it does not assign a too large probability mass to any value. More formally, we measure the "amount of randomness" in terms of *min-entropy* (see [CG]).

Definition 4.1. Let X be a random variable. We say that X has **min-entropy** k, if for every x we have that $\Pr(X = x) \leq 2^{-k}$.

The distribution $Full_{N,g}$ has min-entropy greater than κ, where

$$\kappa \overset{\text{def}}{=} \kappa(N, g) \overset{\text{def}}{=} \lfloor \log(ord_N(g)) \rfloor.$$

The following fact is an immediate consequence of Proposition 3.7:

Fact 4. *Let $\langle N, g \rangle$ be uniformly distributed in P_n, then $\kappa \leq n - \frac{1}{2}\log^2 n$ with negligible probability.*

Using hash functions which have good extracting properties, we are able to "smoothen" the distribution $Full_{N,g}$, and extract from it an almost uniform distribution over strings of length $n - \log^2 n$. To be more formal, we use a family of functions F having an extraction property, satisfying that for all but an ε fraction of the functions in F, a distribution over strings of length n having min-entropy $n - \frac{1}{2}\log^2 n$ is mapped to a distribution over strings of length $n - \log^2 n$ which is ε-close to uniform (we refer to ε, which is generally taken to be negligible in n, as the quality-parameter of the extraction property achieved by F). The price we pay for the use in extractors is reflected in a lower expansion factor of the pseudorandom generators. Specifically, we need to use a part of the random seed in order to choose a random function in the family F that we are using.

Håstad et al. [HSS] used a universal family of hash functions [CW1] in their construction of a pseudorandom generator. The quality parameter achieved by this family of functions is exponentially small in n (and therefore has the best possible quality). However, a universal family of hash functions has to be large: exponential in n. Thus the number of random bits needed to generate (and represent) a function in this family is polynomial in n, resulting in a considerably large loss in the expansion factor of their generator.

Instead, we use an explicit construction due to Goldreich and Wigderson [GW] of a family of functions, which exhibits a tradeoff between the size of the family and the quality parameter ε of the extraction property it achieves. Specifically, they demonstrate a construction of a family of functions of size $\text{poly}(n/\varepsilon)$ achieving the extraction property with quality ε. Taking, for example, $\varepsilon = n^{-\log n}$, yields a family of functions of very good quality (not exponentially small in n but still negligible in n), where each function in the family can be represented using $O(\log^2 n)$ bits. Let F be such a family, where each $f \in F$ maps $\{0, 1\}^n$ to $\{0, 1\}^{n-\log^2 n}$. Thus, we get

Construction 4.2. We define the mapping $G_{N,g}$: $\{0, 1\}^{\lceil n/2 \rceil + O(\log^2 n)} \to \{0, 1\}^n$ as follows: Let $x \in \{0, 1\}^{\lceil n/2 \rceil}$ and $f \in F$. Then

$$G_{N,g}(f, x) \overset{\text{def}}{=} (f, f(g^x \bmod N)).$$

Theorem 4.3. *$G_{N,g}$ is a pseudorandom generator.*

Proof Sketch. Let U_m denote the uniform distribution over $\{0, 1\}^m$. Let r and R be uniformly distributed in $\{0, 1\}^{\lceil n/2 \rceil}$ and $[0, ord_N(g))$, respectively. By Theorem 3.2, $(f, f(g^r \bmod N))$ is indistinguishable from $(f, f(g^R \bmod N))$, which in turn is statistically close to $U_{O((\log n)^2)+(n-(\log n)^2)}$. □

For further details, see our Technical Report [GR].

Increasing the expansion factor of the generator. The pseudorandom generator described above almost doubles the length of its input. However, such a small expansion factor has limited value in practice. Still, it is well known that even a pseudorandom generator G producing $n + 1$ bits from an n-bit seed can be used in order to construct a pseudorandom generator G' having any arbitrary polynomial expansion factor (see, e.g. Theorem 3.3.3 in Section 3.3 of [G2]). Unfortunately, the cost of the latter transformation is rather high: producing each bit in G''s output requires one evaluation of G. Nevertheless, since our generator $G_{N,g}$ has an expansion factor of nearly 2 to start with, we can do better than that: $G_{N,g}$ can be used to construct a generator $G'_{N,g}$ having an arbitrary polynomial expansion factor, such that for every $n/2 - O(\log^2 n)$ bits of output, one evaluation of $G_{N,g}$ is required. We remark that the issue of increasing the expansion factor of $G_{N,g}$ is relevant mostly due to the need to pick the parameters N and g randomly, which requires $O(n)$ additional random bits (as will be explained in the subsequent subsection). Our suggestion is to pick randomly N and g, set them once and for all, and construct a pseudorandom generator having a large expansion factor using this specific $G_{N,g}$. This way the cost of picking N and g becomes negligible (compared with our "profit" from the new generator). We describe now how in general one uses a generator G: $\{0, 1\}^n \to \{0, 1\}^{n+l(n)}$ (for an integer function l) to construct a generator G': $\{0, 1\}^n \to \{0, 1\}^{l(n) \cdot p(n)}$, for any arbitrary polynomial $p(\cdot)$.

Construction 4.4. Let l: $\mathbb{N} \to \mathbb{N}$ be an integer function satisfying $l(n) > 0$ for every $n \in \mathbb{N}$, let $p(\cdot)$ be a polynomial and let G: $\{0, 1\}^n \to \{0, 1\}^{n+l(n)}$ be a deterministic polynomial-time algorithm. Define $G'(s) = \tau_1 \cdots \tau_{p(n)}$, where $s_0 \overset{\text{def}}{=} s$, the string s_i is the n-bit long suffix of $G(s_{i-1})$ and τ_i is the $l(n)$-bit long prefix of $G(s_{i-1})$, for every $1 \leq i \leq p(n)$ (i.e., $\tau_i s_i = G(s_{i-1})$).

Theorem 4.5. *If G is a pseudorandom generator, then so is G'.*

Theorem 4.5 is a generalization of Theorem 3.3.3 proven in [G2] (regarding a generator producing $n + 1$ bits from an n bit seed). Observe that for every $l(n)$ output bits of G', one evaluation of G is required. Using our generator $G_{N,g}$ as the building block, we obtain a generator $G'_{N,g}$ that expands input of size $n/2 + O(\log^2 n)$ to output of size n^c using approximately $n^c/(n/2)$ applications of $G_{N,g}$.

4.2. *An Efficient Choice of the Parameters* (N *and* g)

In order to use the generator $G_{N,g}$ we need to generate the parameters N and g from a primary seed in an "efficient" way, where by "efficient" we mean that both the running time and the amount of randomness used should be as small as possible. The major challenge is to generate efficiently two uniformly distributed primes P and Q, in order to obtain a random $N = P \cdot Q$ in N_n. A random element g in Z_N^* can be chosen using $O(n)$ random coins by picking a random number in $\{0, 1\}^{n+\log^2 n}$ and reducing it modulo N (only with negligible probability the element obtained will not be relatively prime to N). We describe now a general method by which we can pick a random n-bit prime in polynomial time, using only a linear number of random coins.

The trivial algorithm to choose a random n-bit prime is to repeat the following two stages until a prime x is output:

1. Choose a random integer x in $\{0, 1\}^n$.
2. Test whether x is a prime. If it is, stop and output x.

Since the density of primes in $\{0, 1\}^n$ is approximately $1/n$, the expected number of times that the above loop is performed is approximately n. Even assuming that we have a deterministic primality test, the above algorithm requires an expected $O(n^2)$ random bits. We now show how to perform poly(n) dependent iterations of the loop using only $O(n)$ random bits (rather than doing $O(n)$ independent iterations using $O(n^2)$ random bits). We use, however, a probabilistic primality tester of Bach [B], which is a randomness-efficient version of the Miller–Rabin [M], [R2] primality tester.

Theorem 4.6 (Randomness-Efficient Primality Tester [B]). *There exists a probabilistic polynomial-time algorithm that on input P uses $|P|$ random bits so that if P is a prime, then the algorithm always accepts, and otherwise (i.e. P is a composite) the algorithm accepts with probability at most* $1/\sqrt{P}$.

Combining the above procedures, we have

Corollary 4.7. *There exists a probabilistic polynomial-time algorithm that uses* $2n$ *random coins such that*:

1. *With probability* $\Theta(1/n)$ *outputs an n-bit prime. Furthermore, the probability to output a specific prime is* 2^{-n}.
2. *With probability* $1 - \Theta(1/n) - \exp(-n)$ *outputs a special failure sign, denoted* $\perp$.
3. *With probability at most* $2^{-n/2}$ *outputs a composite.*

We refer to the algorithm guaranteed from Corollary 4.7 as a black-box. We associate every string $s \in \{0, 1\}^{2n}$ with the output of the black-box given s as its random coins.

Denote by W the set of strings in $\{0,1\}^{2n}$ that are associated with an n-bit prime. Corollary 4.7 implies that the density of W within $\{0,1\}^{2n}$ is $\Theta(1/n)$. The problem of uniformly picking an n-bit prime translates to a hitting problem, where we need to find a string $s \in W$ (which is subsequently used as random input for the black-box in order to yield a prime). An additional requirement is that the distribution of primes obtained in this way will be very close to uniform. Our goal is to find an algorithm that hits W, whose randomness complexity is linear in n. This goal can be achieved using standard techniques; see our Technical Report [GR].

Acknowledgments

We are grateful to the anonymous reviewers for various helpful comments.

References

[ACGS] W. B. Alexi, B. Chor, O. Goldreich and C. P. Schnorr, RSA and Rabin functions: certain parts are as hard as the whole, *SIAM J. Comput.*, vol. 17, no. 2, 1988, pp. 194–209.

[AKS] M. Ajtai, J. Komlos and E. Szemerédi, Deterministic simulation in LogSpace, *Proceedings of the 19th ACM Symposium on the Theory of Computing*, 1987, pp. 132–140.

[B] E. Bach, How to generate factored random numbers, *SIAM J. Comput.*, vol. 17, no. 2, 1988, pp. 179–193.

[BBS] L. Blum, M. Blum and M. Shub, A simple secure unpredictable pseudo-random number generator, *SIAM J. Comput.*, vol. 15, 1984, pp. 364–383.

[BM] M. Blum and S. Micali, How to generate cryptographically strong sequence of pseudo-random bits, *SIAM J. Comput.*, vol. 13, 1984, pp. 850–864.

[C] B. Chor, *Two Issues in Public Key Cryptography: RSA Bit Security and a New Knapsack Type System*, MIT Press, Cambridge, MA, 1986.

[CG] B. Chor and O. Goldreich, On the power of two-point based sampling. *J. Complexity*, vol. 5, 1989, pp. 96–106.

[CW1] L. Carter and M. Wegman, Universal hash functions, *J. Comput. System Sci.*, vol. 18, 1979, pp. 143–154.

[CW2] A. Cohen and A. Wigderson, Dispensers, deterministic amplification, and weak random sources, *Proceedings of the 30th IEEE Symposium on the Foundations of Computer Science*, 1989, pp. 14–19.

[G1] R. Gennaro, An improved pseudo-random generator based on discrete-log, *CRYPTO* '2000, LNCS 1880, Springer-Verlag, Berlin, 2000, pp. 469–481.

[G2] O. Goldreich, *Foundations of Cryptography—Basic Tools*, Cambridge University Press, Cambridge, 2001.

[G3] O. Goldreich, A sample of samplers: a computational perspective on sampling, ECCC TR97-020, 1997.

[GG] O. Gaber and Z. Galil, Explicit constructions of linear size superconcentrators, *J. Comput. System Sci.*, vol. 22, 1981, pp. 407–420.

[GIL$^+$] O. Goldreich, R. Impagliazzo, L. A. Levin, R. Venkatesan, and D. Zuckerman, Security preserving amplification of hardness, *Proceedings of the 31st IEEE Symposium on Foundation of Computer Science*, pp. 318–326, 1990.

[GR] O. Goldreich and V. Rosen, On the security of modular exponentiation with application to the construction of pseudorandom generators, ECCC, TR02-049, 2002.

[GW] O. Goldreich and A. Wigderson, Tiny families of functions with random properties: a quality-size trade-off for hashing, *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing*, 1994, pp. 574–583.

[HN] J. Håstad and M. Näslund: The security of individual RSA bits, *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 1998, pp. 510–519.

[HSS] J. Håstad, A. W. Schrift and A. Shamir, The discrete logarithm modulo a composite hides $O(n)$ bits, *J. Comput. System Sci.*, vol. 47, 1993, pp. 376–404.

[K1] N. Kahale, Eigenvalues and expansions of regular graphs, *J. Assoc. Comput. Mach.*, vol. 42, no. 5, 1995, pp. 1091–1106.

[K2] B. Kaliski, Jr., A pseudo-random bit generator based on elliptic logarithms. In A. Odlyzko, editor, *Advances in Cryptology: Proceedings of CRYPTO '*86, 1987, pp. 84–103.

[LLMP] A. K. Lenstra, H. W. Lenstra, M. Manasse, and J. M. Pollard, The number field sieve, *Proceedings of the* 22*nd Annual ACM Symposium on Theory of Computing*, 1990, pp. 564–572.

[LW] D. L. Long and A. Wigderson, The discrete logarithm hides $O(\log n)$ bits, *SIAM J. Comput.*, vol. 17, no. 2, 1988, pp. 363–372.

[M] G. L. Miller, Riemann's hypothesis and tests for primality, *J. Comput. System Sci.*, vol. 13, 1976, pp. 300–317.

[OW] P. C. Van Oorschoot and M. Wiener, On Diffie–Hellman key agreement with short exponents, *Advances in Cryptology - EUROCRYPT '*96, LNCS 1070, Springer-Verlag, Berlin, 1996, pp. 332–343.

[P] R. Peralta, Simultaneous security of bits in the discrete log, *Advances in Cryptology - EUROCRYPT* '85, LNCS 219, Springer-Verlag, Berlin, 1986, pp. 62–72.

[R1] M. O. Rabin, Digitalized signatures and public-key functions as intractable as factorization, Technical Report, TR-212, MIT Laboratory for Computer Science, 1979.

[R2] M. O. Rabin, Probabilistic algorithm for testing primality, *J. Number Theory*, vol. 12, 1980, pp. 128–138.

[R3] V. Rosen, On the security of modular exponentiation, Technical Report MCS00-20, Faculty of Mathematics and Computer Science, Weizmann Institute of Science, Rehovot, 2000. See also ECCC Technical Report TR01-007, 2001.

[VV] U. V. Vazirani and V. V. Vazirani, Efficient and secure pseudo-random number generators, *Proceeding of* 25*th IEEE Symposium on Foundations of Computer Science*, 1984, pp. 458–463.