Universität Paderborn
Fachbereich 17 - Mathematik/Informatik
Warburger Straße 100
33095 Paderborn

Diplomarbeit

Construction of pseudorandom functions

Birgitta Fricke

1. Mai 2002

Prof. Dr. Johannes Blömer

“What was that voice?” shouted Arthur.
“I don’t know,” yelled Ford,“I don’t know. It sounded like a measurement of probability.”
“Probability? What do you mean?”
“Probability. You know, like two to one, three to one, five to four against. It said two to the power of hundred thousand to one against. That’s pretty improbable you know.”
A million-gallon vat of custard upended itself over them without warning.
“But what does it mean?” cried Arthur.
“What, the custard?”
“No, the measurement of probability!”
“I don’t know. I don’t know at all.”

[from Douglas Adams, “The Hitch Hiker’s Guide to the Galaxy”,1980]

Inhaltsverzeichnis

1 Introduction

A major subject in cryptography is, to ensure secure communication over an insecure channel. One possible way of ensuring secure communication is a block cryptosystem. Such a system works by dividing the message that is to be sent into blocks of equal length. Then it assigns a unique index to each message block. Next, each block is encrypted seperately and sent over the insecure channel. Tagged to each block is its index so that the receiver can identify each block.
A simple and secure way to encrypt a message block is, to take the XOR of the message block in bit representation and a random bit string of the same length. The message block is decrypted by taking the XOR of the encrypted message block and the random bit string that was used to encrypt it. As long as the random bit string stays secret this kind of encryption is absolutely secure. That is easy to verify, as a message encrypted that way can, without the proper key, be decrypted to any message of the same length. And an adversary has no way of knowing, which was the original one.
In practice there are pseudorandom bit strings used instead of random ones, as truly random bits are hard to get efficiently. A pseudorandom generator can be used to generate the random bits needed for this system. We just need to partition the output of the generator into blocks of the same length as the message blocks. Then each block is used to encrypt the corresponding message block. If all participating parties know the pseudorandom generator and the seed that is used, then only the encrypted message blocks and their indizes need to be send over the insecure channel. Each party then can compute the corresponding pseudorandom bit string to decrypt each message block.
This is just one example, why generating pseudorandomness is one of the most important issues in cryptography. Pseudorandomness is produced by pseudorandom generators and pseudorandom functions, which are two of the most important cryptographic primitives.
In the further examination of the example we will see, why the concept of pseudorandom functions is the more powerfull one.
When a block cryptosystem is realised with a pseudorandom generator, the efficiency is a problem. To get the random bits for the k'th block it is necessary to compute all the random bits for the first $(k-1)$'th blocks. This is especially a problem in a multiplayer enviroment where each party gets only some of the blocks and not all of them. The solution to this problem are pseudorandom functions. With a pseudorandom function it is possible to directly access the needed bit strings

without having to compute all the previous ones. The different parties only need to agree on the pseudorandom function they use for encryption and decryption. Then each receiving party can take the index of the message block they want to decrypt as input for the agreed pseudorandom function and use its output as the bit string required for decryption. Therefore single blocks can be decrypted without the need to compute the keys for the previous blocks.

Motivated by the above, in this thesis we will prove the existence of pseudorandom generators and discuss different constructions of pseudorandom functions.
There are general constructions of pseudorandom generators based on one-way-functions. Furthermore there are general constructions of pseudorandom function families based on pseudorandom generators. But there are also special constructions of pseudorandom function families based on the assumed intractability of explicit number theoretic problems, e.g. the factoring of integers and the discrete logarithm. In this thesis, all those constructions are described. In particular, the similarities and differences of the general and the special constructions of pseudorandom function families based on pseudorandom generators are being examined.

- **Chapter 2:**
  Here we introduce the main definitions and basic notations which we will use in the following chapters.

- **Chapter 3:**
  This chapter starts with the describtion of the major tool we will use in the following chapters. The main theorem of this chapter is the Hybrid Theorem. We will prove it here so that we can use it in all the following chapters. The main literature we use is the book of Michael Luby [15]. But as the Hybrid Theorem is a common tool when dealing with probability distributions it can be found in a varity of books and papers.

- **Chapter 4:**
  Two cryptographic primitives which will be needed several times are distinguishers and predictors. We show how closely related those two notions are. We will come to the conclusion that they can, with minor modifications, be constructed from each other.

- **Chapter 5:**
  Many applications in cryptography need “randomness”. For the sake of efficiency “pseudorandomness” is used in practice. A pseudorandom generator is a tool to get this “pseudorandomness”. Such a pseudorandom generator can be constructed from a one-way-function. By way of such a construction we prove the existence of pseudorandom generators. As in Chapter 3, the main literature we use for this chapter is the book of Michael Luby [15].

- **Chapter 6:**
  In many applications pseudorandom generators are not efficient enough. There, pseudorandom functions are used instead. We will show how to construct a pseudorandom function family from a pseudorandom generator. This construction can be found in a variety of books and papers. In this chapter we mainly work with the books of Michael Luby [15] and Oded Goldreich [11] and an additional paper from him with Shafi Goldwasser and Silvio Micali [12]. This construction requires a length doubling pseudorandom generator. The resulting pseudorandom functions can be evaluated at any given point at the cost of one evaluation of the pseudorandom generator per outputbit.

- **Chapter 7:**
  In this chapter we describe the construction and prove the security of a pseudorandom function family that relies on the security of the Diffie-Hellman-Assumption. The major result of this construction is, that computing the value of such a function at any given point involves only two multiple products, where a multiple product is the product over a polynomial number of factors. Furthermore, such a function is computable by a constant depth circuit consisting of a polynomial number of treshold gates. This construction was introduced by Moni Naor and Omer Reingold [20].

- **Chapter 8:**
  Later Moni Naor, Omer Reingold and Alon Rosen introduced another construction of a pseudorandom function family [22] which we present here. This construction's security is based on the intractability of factoring of Blum-Integers. Its efficiency nearly matches the efficiency of the previous construction. In addition to the two multiple products, this construction needs one modular multiplication per output bit. The advantage this construction has over the previous one is the fact that its output can be extended with only a small overhead in computational complexity. Each new output bits costs one additional modular multiplication.

2 Basic Definitions

2.1 Notation

We will use the following standard notations:

- The set of all natural numbers is denoted by $\mathbb{N}$
- λ denotes the empty word
- $\forall\ N \in \mathbb{N}$ the multiplicative group modulo N is denoted by $\mathbb{Z}_N^*$
- The order of $\mathbb{Z}_N^*$, that is the number of $x \in \{0, 1, \ldots, N-1\}$ with $gcd(x, N) = 1$, is denoted by $\varphi(N)$
- U_k is the uniform distribution in the range $\{0, 1\}^k$
- $a \in_A \{0, 1\}^n$ denotes that a is an n-bit string sampled according to the probability distribution A
- $s \in_U \{0, 1\}^n$ denotes that s is an n-bit string sampled according to the uniform distribution U_n in the range $\{0, 1\}^n$
- $A(s)$ denotes the probability for the string s to be sampled under the probability distribution A.
- $F_{n,l}$ denotes the set of all functions with domain $\{0, 1\}^n$ and range $\{0, 1\}^l$
- $f \in_A F_{n,l}$ denotes that f is a function in $F_{n,l}$ sampled according to the probability distribution A
- "easy to compute" or "efficiently computable" means "computable in probabilistic polynomial time"
- "hard to compute" or "intractable" means "not computable in probabilistic polynomial time"
- "non-negligible probability" means there exists a $k \in \mathbb{N}$, such that for all sufficiently large n the probability is larger than $\frac{1}{n^k}$, where $\{0, 1\}^n$ is the range of the probability distribution

In the following chapters we will use probabilistic algorithms and calculate their success probabilities. Whenever we look at those probabilities, they are taken over the internal coin tosses of those algorithms. When they are taken over additional variables or events that is explicitly mentioned.

2.2 Definitions

The most important tools in cryptography are the so-called cryptographic primitives. One of them is the one-way function. A one-way function is a function, which is easy to compute but whose inverse is intractable.

Definition 2.2.1 (One-Way Function)
Let $l : \mathbb{N} \rightarrow \mathbb{N}$ *be an integer function, such that for all* $n \in \mathbb{N} : l(n) > n$.
Let $f : \{0,1\}^n \rightarrow \{0,1\}^{l(n)}$.
Then f *is a* one-way function, *if* $f(x)$ *can be computed from* $x \in \{0,1\}^n$ *in probabilistic polynomial time and* $x = f^{-1}(y)$ *cannot be computed from* $f(x)$ *in probabilistic polynomial time.*

A special kind of one-way function that we will need is the so-called hard core predicate.

Definition 2.2.2 (Hard Core Predicate)
Let $l : \mathbb{N} \rightarrow \mathbb{N}$ *be an integer function, such that for all* $n \in \mathbb{N} : l(n) > n$.
Let $B : \{0,1\}^n \rightarrow \{0,1\}$, $x \rightarrow B(x)$ *be a boolean predicate for a function* $f : \{0,1\}^n \rightarrow \{0,1\}^{l(n)}$.
B *assigns a single bit* $B(x)$ *to each* $x \in \{0,1\}^n$.
Then B *is a* hard core predicate *for* f, *if* $B(x)$ *can be computed from* $x \in \{0,1\}^n$ *in probabilistic polynomial time and* $B(x)$ *cannot be computed from* $y = f(x)$ *in probabilistic polynomial time.*

This thesis is about generating probability distributions and distinguishing them. To work with this concept, it is necessary to formalize, when two probability distributions are distinguishable and when they are indistinguishable. We now formalize how to tell probability distributions apart. To do that we will introduce the notion of distinguishers. A distinguisher is a probabilistic polynomial time algorithm that distinguishes between probability distributions.

Definition 2.2.3 (Distinguisher for Bit Strings)
Let A and B be two probability distributions on $\{0,1\}^n$.
Let D be a probabilistic polynomial time algorithm that gets as input an n-bit string s. D outputs a single bit, where output 1 means, that D declares s to be sampled according to B, and output 0 means, that D declares s to be sampled according to A. Without loss of generality let $prob[D(s) = 1] > prob[D(s) = 0]$.
Then

$$\delta_D(n) = prob[D(s) = 1 | s \in_B \{0,1\}^n] - prob[D(s) = 1 | s \in_A \{0,1\}^n]$$

is the success probability of D.
If there exists a fixed $k \in \mathbb{N}$ such that for all n big enough this success probability $\delta_D(n)$ is greater than $\frac{1}{n^k}$, then D is called a distinguisher for A and B.

The distinguisher D has success probability $\delta_D(n)$ in computing from which probability distribution the bit string was sampled. The idea behind this definition of success probability is as follows. The distinguisher D tries to determine whether a given bit string was sampled according to A or B. While doing that, the algorithm can use random choices as it is a probabilistic algorithm. The output represents D's decision. Output 1 means that D thinks that it is more likely that s was sampled according to B. Output 0 means that D thinks that it is more likely that s was sampled according to A. Therefore the success rate of D is the probability that D "is right", when saying that the input was sampled according to B minus the probability that D "is wrong", when saying that the input was sampled according to B. And an algorithm D is a distinguisher for A and B, when it is right significantly more often than it is wrong. In the definition, we assume that $\text{prob}[D(s) = 1] > \text{prob}[D(s) = 0]$. If we would assume that $\text{prob}[D(s) = 1] < \text{prob}[D(s) = 0]$, we would get an analogous definition of success probability. The success probability then would be $\text{prob}[D(s) = 0 | s \in_A \{0,1\}^n]$ - $\text{prob}[D(s) = 0 | s \in_B \{0,1\}^n]$. On the other hand, any distinguisher D with $\text{prob}[D(s) = 1] < \text{prob}[D(s) = 0]$ can easily be modified into a distinguisher D' with $\text{prob}[D'(s) = 1] > \text{prob}[D'(s) = 0]$. The only thing for D' to do would be to run D and negate the output of D. Therefore, in the definition we can assume $\text{prob}[D(s) = 1]$ is greater than $\text{prob}[D(s) = 0]$ without loss of generality.
The concept of a distinguisher leads directly to the next definition.

Definition 2.2.4 (Polynomial Indistinguishability)
Two probability distributions A and B are called polynomially indistinguishable, *if there exists no distinguisher for A and B. In other words, there exists no probabilistic polynomial time algorithm that can determine with non-negligible success probability, whether a given bit string was sampled according to A or according to B.*

We now come to the special case that one of the probability distributions in question is the uniform distribution. That is in fact what we will deal with mostly. We will want to get distributions of "random-looking" bit strings. These bit strings are called pseudorandom. If a probability distribution is polynomially indistinguishable from the uniform distribution, it is called pseudorandom.

Definition 2.2.5 (Pseudorandomness)
Let s be a bit string sampled from a probability distribution other than the uniform distribution. Then s is called pseudorandom *if there exists no distinguisher that can distinguish s from a bit string sampled uniformly in the same range.*
A probability distribution A is pseudorandom*, if all bit strings sampled according to A are pseudorandom, i.e., there exists no distinguisher for A and U, the uniform distribution in the same range.*

A reliable source of randomness is often needed in cryptography. For example, keys are often required to be randomly chosen in their respective key space. And key or information exchange protocols often require random numbers, too. But true randomness, like coin tosses, is expensive and time-consuming. Therefore, to these ends pseudorandom bit strings are used instead of random ones. Now, of course, the next question is how to generate such pseudorandomness. To get pseudorandomness, the previous concept of one-way functions was enlarged to the concept of pseudorandom generators. Basically a pseudorandom generator is a randomness-expander. A pseudorandom generator takes as input a short random sequence, the so-called seed. The pseudorandom generator then expands this short bit sequence into a polynomial long "random-looking", pseudorandom bit sequence. A pseudorandom generator is a probabilistic polynomial-time algorithm, that stretches a random n-bit seed into a pseudorandom bit string of polynomial length. We will now give the formal definition of a pseudorandom generator.

Definition 2.2.6 (Pseudorandom Generator)
Let $l(n) : \mathbb{N} \rightarrow \mathbb{N}$ be an integer function, such that for all $n \in \mathbb{N} : l(n) > n$.
Then $G : \{0,1\}^n \rightarrow \{0,1\}^{l(n)}$ is a pseudorandom generator*, if for all probabilistic polynomial-time algorithms D, for all $k \in \mathbb{N}$ and for all sufficiently large n:*

$$prob[D(y) = 1 | y \in_G \{0,1\}^n] - prob[D(y) = 1 | y \in_U \{0,1\}^n] < \frac{1}{n^k}$$

i.e., there exists no distinguisher D for the probability distribution generated by G and the uniform probability distribution in the same range with success probability greater than $\frac{1}{n^k}$.

A pseudorandom generator produces one polynomially long pseudorandom bit string per input. That may not be enough in all applications. So the next idea is to look at functions. That is to say, look at groups of functions, so-called function families. The description of a random function is exponential in the size of its domain and can be regarded as an exponentially long bit string. Therefore taking a random function and evaluating it at a random point is like taking a random bit in an exponentially long random bit string. In that sense, this would give us access to exponentially long random bit strings. But as before with random bit strings, taking truly random functions is much too inefficient. Therefore we are dealing with pseudorandomness again. But before we formalize that concept, let us first formalize function families.

Definition 2.2.7 (Efficiently Computable Function Family)
Let $k, l : \mathbb{N} \rightarrow \mathbb{N}$ be integer functions, such that for all $n \in \mathbb{N} : l(n) > n$ and $k(n) > n$.
Let $F : \{0,1\}^n \times \{0,1\}^{k(n)} \rightarrow \{0,1\}^{l(n)}$ be a collection of functions, such that for all $i \in \{0,1\}^n$ there is a unique function $f_i : \{0,1\}^{k(n)} \rightarrow \{0,1\}^{l(n)}$.
Then $F = \{F_n\}_{n \in \mathbb{N}}$ is called an efficiently computable function family*, if:*

(1) Indexing:
Each function $f_i \in F_n$ has a unique n-bit index associated with it. Thus, picking randomly a function $f \in F$ is easy, if n random bits are available.

(2) Polynomial Time Evaluation:
There exists a polynomial time algorithm that (for all $n \geq 1$), upon input of an index $i \in \{0,1\}^n$ and an argument x, computes $f_i(x)$.

Each function family is a probability distribution on the set of all functions with the same domain and range. Each function family is defined by the keys to its functions. Thus, it is the keys, that define the probability distribution on the set of all possible functions. What we need next, is a way to tell different function families apart. We want to know, from which distribution of functions a given function is. Therefore we need a way to tell different function families apart. We need a distinguisher for functions.
Let A and B be two probability distributions of functions in the same domain range. Let F denote the set of all functions in this domain and range.

Definition 2.2.8 (Distinguisher for Functions)
Let D be a probabilistic polynomial time algorithm that gets as input oracle access to a function in F. That means, that D can make a polynomial number of queries on the function. D outputs a single bit, where output 1 means, that D declares f to be sampled according to B, and output 0 means, that D declares f to be sampled according to A.

Without loss of generality let prob$[D(s) = 1] > prob[D(s) = 0]$.
Then

$$\delta(n) = prob[D(f) = 1 | f \in_B F] - prob[D(f) = 1 | f \in_A F]$$

is the success probability of D.
If there exist a fixed $k \in \mathbb{N}$ *such that for all* n *large enough this success probability* $\delta(n)$ *is greater than* $\frac{1}{n^k}$, *then* D *is called a* distinguisher for A and B.

The distinguisher D has success probability $\delta(n)$ in computing according to which probability distribution the function f was sampled. The idea behind this definition of success probability is exactly the same as for the distinguisher for bit strings.
The distinguisher D tries to determine whether a given function was sampled according to A or B. While doing that, the algorithm can use random choices as it is a probabilistic algorithm. The distinguisher gets the function as a black box. It does not have access to the key that identifies the function. However, the distinguisher can interactively query the function on a polynomial number of points. The output represents D's decision. Output 1 means, that D thinks that it is more likely, that f was sampled according to B. Output 0 means, that D thinks that it is more likely, that f was sampled according to A. Therefore the success rate of D is the probability that D "is right", when saying that the input was sampled according to B, minus the probability that D "is wrong", when saying that the input was sampled according to B. And an algorithm D is a distinguisher for A and B when it is right significantly more often than it is wrong. In the definition, we assume that prob$[D(f) = 1] >$ prob$[D(f) = 0]$. If we would assume, that prob$[D(f) = 1] <$ prob$[D(f) = 0]$, we would get an analogous definition of success probability. The success probability then would be prob$[D'(f) = 0 | f \in_A F]$ - prob$[D'(f) = 0 | f \in_B F]$. But on the other hand, any distinguisher D with prob$[D(f) = 1] <$ prob$[D(f) = 0]$ can easily be modified into a distinguisher D' with prob$[D'(f) = 1] >$ prob$[D'(f) = 0]$. The only thing for D' to do would be to run D and negate the output of D.
The main difference between a distinguisher for bit strings and a distinguisher for functions, however, is the access D has on the subject it is trying to assign to a probability distribution. When the subject is a bit string, the distinguisher gets the whole string. But when the subject is a function, the distinguisher only gets oracle-access to it. Oracle-access means that the distinguisher gets the function as a black box which it can query only polynomially often.
Most of the time we look at only one function family. Then the question is, whether a given function from that family can be efficiently told apart from a function randomly selected in the same domain and range. If for all functions in an efficiently computable function family it is not possible to efficiently tell them apart from truly random functions that are uniformly selected in the same domain and range, the function family is called a *pseudorandom function family*. In the following chapters we will show the existence of such pseudorandom function

families, given the existence of one-way functions. We will show how to construct them from pseudorandom generators and will compare two concrete constructions. So now we will give the most important definition of this thesis:

Definition 2.2.9 (Pseudorandom Function Family)
Let $F : \{0,1\}^n \times \{0,1\}^{k(n)} \rightarrow \{0,1\}^{l(n)}$ be an efficiently computable function family. Let $f_i \in F_n$ be a function in this function family.
Then $F = \{F_n\}_{n \in \mathbb{N}}$ is a pseudorandom function family, if for all probabilistic polynomial-time algorithms D, that have black box access on a polynomial number of values of their input function, for all $k \in \mathbb{N}$ and sufficiently large n:

$$prob[D(f) = 1 | f = f_i \in F] - prob[D(f) = 1 | f \in_U F_{k(n),l(n)}] < \frac{1}{n^k}$$

i.e., there is no distinguisher D for F and $F_{k(n),l(n)}$ with success probability greater than $\frac{1}{n^k}$.

3 The Hybrid Argument

In all following chapters, we will make extensive usage of a standard proof-technique, the hybrid argument. This technique was used in nearly all the books and papers in the bibliography and can be looked up for example in [12] as well as in any basic book about cryptography. In this chapter we will explain this argument and show how and when it can be used.

Informally put, we will get a way of showing the indistinguishability of two probability distributions. The method to do that is to first define a sequence of hybrid distributions. Those hybrid distributions are defined in such a way that the extreme distributions are the original ones. The second step then is to show that if any two neighbouring distributions are polynomially indistinguishable then so are the extreme ones.

Let A and B be two probability distributions on $\{0,1\}$. Let A^n and B^n be the probability distributions on $\{0,1\}^n$ that we get when we sample n bits according to A and B, respectively. Then the hybrid argument is used to show the polynomial indistinguishability of two bit strings of length n arbitrarily sampled from the two probability distributions A^n and B^n. The only prerequisite is, that single bits sampled from A and B are polynomially indistinguishable. Now, we construct a set S of $n+1$ bit strings S_i, $i \in \{0,...,n\}$ of length n. The first string S_0 is sampled from the distribution A^n and the last string S_n is sampled from the distribution B^n. The other strings are sampled from hybrid distributions. In string S_i, the first i bits are sampled from distribution A and the last $n-i$ bits are sampled from distribution B. Therefore, $S_0 = S_B$ is sampled from B^n and $S_n = S_A$ is sampled from A^n.

$$S_B = S_0 = b_1 \dots b_n$$

$$\begin{aligned} S_1 &= a_1 b_2 \dots b_n \\ &\vdots \\ S_k &= a_1 \dots a_k b_{k+1} \dots b_n \\ &\vdots \\ S_n &= a_1 \dots a_n \quad = S_A \end{aligned}$$

where for all $i = 1 \dots n$ the bits a_i are sampled from A and the bits b_i are sampled from B.

The proof now mainly consists of showing that S_0 and S_n are polynomially indistinguishable. The proof works by leading the assumption that S_0 and S_n are polynomially indistinguishable to a contradiction. This is done by proving that if S_0 and S_n are polynomially distinguishable, then there must exist two neighbouring strings S_k and S_{k+1} that are polynomially distinguishable. But any two neighbouring strings differentiate in only one bit. In S_k, the k'th bit is sampled from probability distribution A and in S_{k-1} the k'th bit is sampled from probability distribution B. This implies, that S_k and S_{k-1} are polynomially distinguishable, if and only if those two bits are polynomially distinguishable. And that is a contradiction to the prerequisite that any two such bits are polynomially indistinguishable. Thus, any two bit strings $S_0 = S_B$ and $S_n = S_A$ that are sampled randomly from B^n and A^n respectively are polynomially indistinguishable. Therefore A^n and B^n are polynomially indistinguishable.

Let A, B, A^n and B^n be probability distributions defined as above.
Let a be a single bit sampled randomly from A.
Let b be a single bit sampled randomly from B.
Let S_A be an n-bit string sampled randomly from A and let S_B be an n-bit string sampled randomly from B.

Theorem 3.0.1 (Hybrid Theorem)
If a and b are polynomially indistinguishable, then so are S_A and S_B.
If A and B are polynomially indistinguishable, then so are A^n and B^n.

Proof:

Let D be a distinguisher for A^n and B^n. The distinguisher takes an n-bit string as input and returns a single output bit. The output 0 represents D's decision that the input string was sampled according to A^n. The output 1 represents D's decision that the input string was sampled according to B^n. Then for all n-bit strings S_A sampled from A^n and for all n-bit strings S_B sampled from B^n there exists a fixed $k \in \mathbb{N}$ such that for all n big enough:

$$\text{prob}[D(s) = 1 | s \in_{B^n} \{0,1\}^n] - \text{prob}[D(s) = 1 | s \in_{A^n} \{0,1\}^n] > \frac{1}{n^k}$$

Then we can construct a distinguisher D' for a and b, where for all bits a sampled from A and for all bits b sampled from B there exists a fixed $k' \in \mathbb{N}$, such that for all n big enough:

$$\text{prob}[D'(d) = 1 | d \in_B \{0,1\}] - \text{prob}[D'(d) = 1 | d \in_A \{0,1\}] > \frac{1}{n^{k'}}$$

Here analogously the output 1 of D' represents D''s decision that d was sampled according to B and the output 0 of D' represents D''s decision that d was sampled according to A.

Construction of D':

D' gets a single bit d as input and is supposed to determine whether this bit was sampled from A or B. D' does this by simulating D with an input-string s' that is partly chosen according to A^n and partly chosen according to B^n. The first step is to pick a random value k between 1 and n, the input length of D. Then D' samples a random bit string from the probability distribution S_k by sampling k random values in A and $n-k-1$ random values in B. D's input is the n-bit-string s' consisting of the k bits sampled from A, d and the $n-k-1$ bits sampled from B.

$$s' = \underbrace{s'_1 \dots s'_k}_{A}\, d\, \underbrace{s'_{k+2} \dots s'_n}_{B}$$

Then D' runs D with this n bit-string s' as input. If D outputs 1, i.e., if D declares s' to be sampled from A^n, then D' also outputs 1, i.e., declares d to be sampled from A. If D outputs 0, i.e., declares s' to be sampled from B^n, D' also outputs 0, i.e., declares d to be sampled from B.

Distinguisher D':

Input: $d \in \{0,1\}$
Output: $D'(d) \in \{0,1\}$

- Choose $k \in_U \{1, \dots, n\}$
- Sample s' as described above
- Run $D(s')$
- $D'(d) := D(s')$

Let q_1 be the probability, that D "is right, when choosing B", which is the probability, that D outputs 1 when the input was sampled from B^n.

$$q_1 := \text{prob}[D(s) = 1 | s \in_{B^n} \{0,1\}^n]$$

Let q_0 be the probability, that D "is wrong, when choosing B", which is the probability, that D outputs 1 when the input was sampled from A^n.

$$q_0 := \text{prob}[D(s) = 1 | s \in_{A^n} \{0,1\}^n]$$

Therefore D has success probability:

$$\delta(n) = q_1 - q_0 > 0$$

Let $p_{1,j}$ be the probability that "$k = j$ and D' is right, when choosing B", which is the probability, that D' picks j as random bit and outputs 1, when d was sampled from B.

$$p_{1,j} := \text{prob}[D'(d) = 1 | k = j \text{ and } d \in_B \{0,1\}]$$

Let $p_{0,j}$ be the probability that "$k = j$ and D' is wrong, when choosing B", which is the probability, that D' picks j as random bit and outputs 1, when d was sampled from A.

$$p_{0,j} := \text{prob}[D'(d) = 1 | k = j \text{ and } d \in_A \{0,1\}]$$

As there are n possibilities for D' to choose k, the overall probabilities for D' are

$$p_1 = \frac{1}{n} \sum_{j=1}^{n} p_{1,j} = \text{prob}[D'(d) = 1 | d \in_B \{0,1\}]$$

and

$$p_0 = \frac{1}{n} \sum_{j=1}^{n} p_{0,j} = \text{prob}[D'(d) = 1 | d \in_A \{0,1\}].$$

Then D' has success probability:

$$\delta'(n) = p_1 - p_0$$

Now the key observations are:

1. $p_{1,j+1} = p_{0,j} \ \forall j = 1 \dots n-1$
2. $p_{1,1} = q_1$
3. $p_{0,n} = q_0$

Let us look at these observations more closely:

1. $p_{1,j+1} = p_{0,j}\ \forall j = 1 \ldots n-1$

   The first observation is that $p_{1,j+1}$ equals $p_{0,j}$ for all $j = 1 \ldots n-1$.
   As

   $$p_{1,j+1} = \text{prob}[D'(d) = 1 | k = j+1 \text{ and } d \in_B \{0,1\}]$$

   this means, that the bits 1 to j in s' are sampled from A and the bits $j+2$ to n in s' are sampled from B. The $(j+1)$'th bit in s' is d and therefore it is sampled from B.

   $$s' = \underbrace{s'_1 \ldots s'_j}_{A} \underbrace{d s'_{j+2} \ldots s'_n}_{B}$$

   As

   $$p_{0,j} = \text{prob}[D'(d) = 1 | k = j \text{ and } d \in_A \{0,1\}]$$

   this means, that the bits 1 to $j-1$ in s' are sampled from A and the bits $j+1$ to n in s' are sampled from B. The j'th bit in s' is d and therefore it is sampled from A.

   $$s' = \underbrace{s'_1 \ldots s'_{j-1} d}_{A} \underbrace{s'_{j+1} \ldots s'_n}_{B}$$

   In both cases, if D' outputs 1 the bits 1 to j in s' are sampled from A and the bits $j+1$ to n in s' are sampled from B, i.e., for all $j = 1 \ldots n-1$:

   $$p_{1,j+1} = p_{0,j}$$

2. $p_{1,1} = q_1$

   The next observation is, that $p_{1,1}$ equals q_1:

   $$p_{1,1} = \text{prob}[D'(d) = 1 | k = 1 \text{ and } d \in_B \{0,1\}]$$

   As D' has chosen $k = 1$, d is the first bit in the bit string s'. As d is sampled from B, all bits in s' are sampled from B. And that is exactly the distribution given in the probability q_1. Therefore

   $$p_{1,1} = q_1$$

3. $p_{0,n} = q_0$

   The last observation is that $p_{0,n}$ equals q_0:

   $$p_{0,n} = \text{prob}[D'(d) = 1 | k = n \text{ and } d \in_A \{0,1\}]$$

   As D' has chosen $k = n$, d is the last bit in s'. As d is sampled from A, all bits in s' are sampled from A. And that is exactly the distribution given in the probability q_0. Therefore

   $$p_{0,n} = q_0$$

With these observations, we can calculate the success probability $\delta'(n)$ of D'.

$$\begin{aligned}
\delta'(n) = p_1 - p_0 &= \frac{1}{n}\sum_{j=1}^{n} p_{1,j} - \frac{1}{n}\sum_{j=1}^{n} p_{0,j} \\
&= \frac{1}{n}\sum_{j=1}^{n}(p_{1,j} - p_{0,j}) \\
&= \frac{1}{n}(\underbrace{p_{1,1}}_{=q_1} \underbrace{-p_{0,1} + p_{1,2}}_{=0} - p_{0,2} + \cdots + p_{1,n-1} \underbrace{-p_{1,n-1} + p_{1,n}}_{=0} - \underbrace{p_{1,n}}_{=q_1}) \\
&= \frac{q_1 - q_0}{n} = \frac{\delta(n)}{n}
\end{aligned}$$

Hence, we have:

$$\delta'(n) = \frac{\delta(n)}{n}$$

We know that

$$\delta(n) = q_1 - q_0 \geq \frac{1}{n^k}$$

We want to show that there exists a $k' \in \mathbb{N}$ such that for all n big enough:

$$\delta'(n) = p_1 - p_0 \geq \frac{1}{n^{k'}}$$

Define $k' := k + 1$. Then

$$\begin{aligned}\delta'(n) &= \frac{\delta(n)}{n}\\ &= \frac{q_1 - q_0}{n}\\ &> \frac{\frac{1}{n^k}}{n}\\ &= \frac{1}{n^{k+1}}\\ &= \frac{1}{n^{k'}}\end{aligned}$$

Hence, we have the desired result:

$$\begin{aligned}\delta'(n) &= p_1 - p_0\\ &= \frac{1}{n}\sum_{j=1}^{n} p_{1,j} - \frac{1}{n}\sum_{j=1}^{n} p_{0,j}\\ &= \mathrm{prob}[D'(d) = 1 | d \in_B \{0,1\}] - \mathrm{prob}[D'(d) = 1 | d \in_A \{0,1\}]\\ &> \frac{1}{n^{k'}}\end{aligned}$$

D' is a distinguisher for A and B. Hence, A and B are polynomially distinguishable. That is a contradiction. Therefore the basic assumption, that A^n and B^n are polynomially distinguishable is wrong.
A^n and B^n are polynomially indistinguishable, the Hybrid Theorem 3.0.1 on page 13 is proven.

$\square$

4 Predictor and Distinguisher

We have already defined a distinguisher for bit strings. Now we come to a closely related term, the predictor. A predictor is a probabilistic polynomial time algorithm that takes as input a part of a bit string sampled from a specific probabilistic distribution. The algorithm now tries to determine, what the next bit in this bit string is.
To formalize the definition of a predictor, we will need the following notations.
Let $\pi_j : \{0,1\}^n \to \{0,1\}^j$ denote the projection, that maps any bit string of length n to it's j first bits.
Then $\pi_j(A)$ is the marginal distribution of A on the first j bits, $j \leq n$.

Definition 4.0.10 (Predictor)
Let $x = (x_1 \dots x_n)$ be a bit string sampled according to a probability distribution B.
Let P be a probabilistic polynomial time algorithm that takes as input the bit string $\pi_{i-1}(x) = (x_1 \dots x_{i-1})$, $i < n$.
P predicts x_i with success probability

$$\sigma_P(B) = prob\ [P(\pi_{i-1}(x)) = x_i | x \in_B \{0,1\}^n]$$

P is a $(i, \sigma_P(B))$-predictor, *if P predicts the i'th bit with success probability $\sigma_P(B) \geq \frac{1}{2} + \frac{1}{n^k}$ for some $k \in \mathbb{N}$ and n big enough.I.e., P has polynomial advantage.*

The success probability of P therefore is the probability that on input $\pi_{i-1}(x)$, where x is sampled according to B, P guesses the right x_i.
For P to be a predictor, its success rate is required to be significantly greater than $\frac{1}{2}$ for the following reason. A simple coin toss would predict the next bit of a random sequence with probability exactly $\frac{1}{2}$. If the sequence is not random then it may be possible to predict the next bit with greater probability than $\frac{1}{2}$. It is not necessary to regard the case that the success probability of a prediction algorithm A is less than $\frac{1}{2}$. That is, because such an algorithm A can be easily transformed into the algorithm $\overline{A}$ with success probability greater than $\frac{1}{2}$ by simply negating the output of A.

From this definition of a predictor there closely follows another one:

Definition 4.0.11 (Computationally Unpredictable)
A probability distribution B on $\{0,1\}^n$ is called computationally unpredictable, *if for any bit string $x \in \{0,1\}^n$ chosen according to B there is no predictor for any bit in x.*

We will now prove, that the terms of predictor and distinguisher are closely related and that therefore pseudorandom probability distributions are computationally unpredictable and vice versa.
A good predictor can always be used as a distinguisher. A good distinguisher can always be transformed into a reasonably good predictor for some bit.
This can be seen best by the two following constructions.

4.1 Construction of a Distinguisher from a Predictor

It is easily verifiable, that a predictor can be used as a distinguisher.
Suppose, B is a probability distribution on $\{0,1\}^n$ and P is a predictor for the i'th bit under B. P has success probability

$$\sigma_P(B) = \text{prob}\,[P(\pi_{i-1}(x)) = x_i | x \in_B \{0,1\}^n] \geq \frac{1}{2} + \frac{1}{n^k} \tag{4.1}$$

for some $k \in \mathbb{N}$ and n big enough.

Theorem 4.1.1
Let P be a predictor for any bit under B with success probability $\sigma_P(B)$.
Then there exists a distinguisher D for B and U_n the uniform probability distribution on $\{0,1\}^n$.

Proof:

First we construct the following distinguisher D for B and U_n, the uniform distribution in the same range as B.

Distinguisher D**:**

Input: $x = (x_1, \dots x_{i-1} x_i \dots x_n) \in \{0,1\}^n, P, i$
Output: $D(x) \in \{0,1\}$

- $z := P(x_1 \dots x_{i-1})$
- $D(x) := x_i \oplus z \oplus 1$

This algorithm is efficiently computable, if $P(x_1 \dots x_{i-1})$ is efficiently computable. But that is a prerequisite and therefore, D is efficiently computable.
Let us now examine what D does. The algorithm takes the input and passes the first i bits to the predictor P. Then D compares the output of P with the i'th input bit x_i. If those two bits are equal, D decides that x was sampled according to B. If they are not equal, D decides that x was sampled randomly in $\{0,1\}^n$. Why does that work? If x is sampled according to B, then P predicts the i'th bit x_i with success probability $\sigma_P(B)$. This implies, that if $x_i = z$, where z is the output of P, it is more likely, that x is sampled according to B than to U_n. Therefore, if $x_i = z$ the output of $D(x)$ is 1. This means, that D decides x to be sampled according to B. Analogously if $x_i \neq z$ it is more likely that x is sampled uniformly in $\{0,1\}^n$ and not according to B. Therefore, the output of $D(x)$ is 0 as D decides x to be random.
We will now formalize this by looking at the success probability $\delta_D(n)$ of D:

$$\delta_D(n) = \mathrm{prob}[D(x) = 1 | x \in_B \{0,1\}^n] - \mathrm{prob}[D(x) = 1 | x \in_U \{0,1\}^n]$$

But when does D have output 1? If we look back into the algorithm, we see, that $D(x)$ is 1 if and only if $z = x_i$, where $z = P(\pi_{i-1}(x))$. Therefore

$$\delta_D(n) = \mathrm{prob}[P(\pi_{i-1}(x)) = x_i | x \in_B \{0,1\}^n] - \mathrm{prob}[P(\pi_{i-1}(x)) = x_i | x \in_U \{0,1\}^n]$$

Now let us look at the probability that P has output x_i, if x was chosen uniformly in $\{0,1\}^n$. This is the probability that P correctly predicts the uniform probability distribution. But as x_i is chosen uniformly in $\{0,1\}$, P cannot "be right" with probability more than $\frac{1}{2}$, because both possibilities 0 and 1 for x_i have exactly this probability. Also, P cannot "be right" with probability less than $\frac{1}{2}$, because P could then be turned into a predictor $\overline{P}$ for x_i with success probability greater than $\frac{1}{2}$ by simply negating the output bit. This implies that P "is right", when predicting the i'th bit under U_n with probability exactly $\frac{1}{2}$ and also P "is wrong" with probability $\frac{1}{2}$.

Therefore, we have that:

$$\text{prob}\ [P(\pi_{i-1}(x)) = x_i | x \in_U \{0,1\}^n] = \frac{1}{2}$$

With this and with equation 4.1 on page 20 we can now compute the success rate $\delta_D(n)$ of D:

$$\begin{aligned}
\delta_D(n) &= \underbrace{\text{prob}[P(\pi_{i-1}(x)) = x_i | x \in_B \{0,1\}^n]}_{\geq \frac{1}{2}+\frac{1}{n^k}} - \underbrace{\text{prob}[P(\pi_{i-1}(x)) = x_i | x \in_U \{0,1\}^n]}_{=\frac{1}{2}} \\
&\geq \frac{1}{2} + \frac{1}{n^k} - \frac{1}{2} \\
&= \frac{1}{n^k}
\end{aligned}$$

Hence, for some $k \in \mathbb{N}$ and all n big enough:

$$\delta_D(n) \geq \frac{1}{n^k}$$

That is the desired result. We now have proven Theorem 4.1.1 on page 20. D is a distinguisher for B and U_n and has a non-negligible success probability.

□

4.2 Construction of a Predictor from a Distinguisher

Proving that a predictor can be constructed from a distinguisher is more complicated. And the construction is not as general as the one in the last section. We have given a distinguisher D for a probability distribution B on $\{0,1\}^n$ and U_n, the uniform probability distribution in the same range. What we want now, is a predictor P for B. But as a predictor predicts a single bit, we need to know which bit under B we can predict, given D. We will not get a predictor for any bit under the probability distribution B. The following theorem, that was achieved by Yao in 1982 [32], therefore just states, that there exists a bit, such that it is possible to construct a predictor for this bit.

Theorem 4.2.1 (Yao's Theorem)
Let B be a probability distribution on $\{0,1\}^n$.
Let D be a distinguisher for B and U_n with success probability $\delta_D(n)$.
Then there exists an $i \in \mathbb{N} : 0 \leq i \leq n$ such that there exists an $(i, \sigma_P(B))$-predictor P for the i'th bit under B with success probability $\sigma_P(B) = \frac{\delta_D(n)}{n}$.

Proof:

The first part of the proof will be to show that there exists such an i. To do that, we will use hybrid distributions, a tool that we introduced in the previous chapter.
In the second part we will show that there exists a predictor for this i'th bit under B with success probability $\sigma_P(B) = \frac{\delta_D(n)}{n}$.

Proof Part 1:

For $0 \leq j \leq n$ let B_j be a hybrid probability distribution, defined as follows:

$$B_j := \pi_j(B) \times U_{n-j}$$

$\pi_j(B)$ is the marginal distribution of B with range $\{0,1\}^j$ and U_{n-j} is the uniform distribution with range $\{0,1\}^{n-j}$. Therefore, B_j is the hybrid distribution, where the first j bits are sampled according to B and the last $n-j$ bits are sampled uniformly in $\{0,1\}^{n-j}$. This implies, that B_n equals B and that B_0 is the uniform distribution U_n.
We know, that there exists a distinguisher D for B and U_n with success probability $\delta_D(n)$.

$$\begin{aligned}
\delta_D(n) &= \text{prob}\ [D(x) = 1 | x \in_B \{0,1\}^n] - \text{prob}\ [D(x) = 1 | x \in_U \{0,1\}^n] \\
&= \text{prob}\ [D(x) = 1 | x \in_{B_n} \{0,1\}^n] - \text{prob}\ [D(x) = 1 | x \in_{B_0} \{0,1\}^n] \\
&= \sum_{j=1}^{n} (\text{prob}\ [D(x) = 1 | x \in_{B_j} \{0,1\}^n] - \text{prob}\ [D(x) = 1 | x \in_{B_{j-1}}]) \\
&\leq n \cdot \max_{0 \leq j \leq n} (\text{prob}\ [D(x) = 1 | x \in_{B_j} \{0,1\}^n] - \text{prob}\ [D(x) = 1 | x \in_{B_{j-1}} \{0,1\}^n])
\end{aligned}$$

Hence,

$$\delta_D(n) \leq n \cdot \max_{0 \leq j \leq n} (\text{prob}\ [D(x) = 1 | x \in_{B_j} \{0,1\}^n] - \text{prob}\ [D(x) = 1 | x \in_{B_{j-1}} \{0,1\}^n])$$

and

$$\frac{\delta_D(n)}{n} \leq \max_{0 \leq j \leq n} (\text{prob}\ [D(x) = 1 | x \in_{B_j} \{0,1\}^n] - \text{prob}\ [D(x) = 1 | x \in_{B_{j-1}} \{0,1\}^n])$$

Therefore, there exists at least one i between 1 and n, such that:

$$\frac{\delta_D(n)}{n} \leq \text{prob } [D(x) = 1 | x \in_{B_i} \{0,1\}^n] - \text{prob } [D(x) = 1 | x \in_{B_{i-1}} \{0,1\}^n] \quad (4.2)$$

□

Proof Part 2 :

We will noe prove, that it is possible to construct an $(i, \sigma_P(B))$-predictor given the i, whose existence we have shown in the first part of the proof.

Let $x = (x_1 \dots x_{i-1} x_i \dots x_n) \in \{0,1\}^n$ be sampled according to B.

Predictor P:

Input: $(x_1 \dots x_{i-1}) = \pi_{i-1}(x) \in \{0,1\}^{i-1}$, D
Output: $P(\pi_{i-1}(x))$

- Choose $y_i, \dots, y_n \in_U \{0,1\}$
- $x' := (x_1 \dots x_{i-1} y_i \dots y_n) = (\pi_{i-1}(x) y_i \dots y_n) \in \{0,1\}^n$
- Run $D(x')$
- $P(\pi_{i-1}(x)) := y_i \oplus D(x') \oplus 1$

What P does here, is to sample the bit string x' according to B_{i-1} and then run the distinguisher D. It is D, that decides the output of P. The output 1 of $D(x')$ represents D's decision that x' is generated according to B, which is closer to B_i than to B_{i-1}, by at leas $\frac{\delta_D(n)}{n}$, as can be seen in equation 4.2. Therefore x' is more likely by $\frac{\delta_D(n)}{n}$ to be generated according to the probability distribution B_i. Analogously, the output 1 of $D(x')$ represents D's decision that x' is sampled uniformly in $\{0,1\}^n$, which is closer to B_{i-1} than to B_i. Therefore the probability for x' to be generated according to the probability distribution B_{i-1} is higher by $\frac{\delta_D(n)}{n}$ than the probability that x' was generated according to B_i. But the only difference between these two probability distributions B_{i-1} and B_i is the i'th bit. In B_{i-1} it is sampled randomly and in B_i it is generated like the i'th bit of a bit string generated according to B. Hence, when D has output 1, it decides that x' was sampled from B_i and P outputs its original random choice y_i for the i'th bit. But when D has output 0, it decides that x' was sample from B_{i-1} and P outputs

the negation of its original random choice for the i'th bit. That means, that P has output 1, if $y_i = D(x')$ and output 0, if $y_i = \overline{D(x')}$.
The predictor P is a probabilistic algorithm as the last $n - (i - 1)$ bits of x' are sampled uniformly in $\{0, 1\}^{n-(i-1)}$. Provided these random bits are available, all the computations in P can be done in polynomial time, given that $D(x')$ can be computed in polynomial time. But that is a prerequisite.
Now we can calculate the success probability $\sigma_P(B)$ of P. To do that, we first need to look at the probability that $D(x')$ has output 1, if x' was sampled according to the probability distribution B_{i-1}. This probability can be divided up into the cases of $x_i = y_i$ and $x_i \neq y_i$:

$$\text{prob}\ [D(x') = 1 | x' \in_{B_{i-1}} \{0,1\}^n]$$

$$\begin{aligned} = \quad & \text{prob}\ [D(x') = 1 | x \in_B \{0,1\}^n, (y_i, \ldots, y_n) \in_U \{0,1\}^{n-i}, x_i = y_i] \cdot \text{prob}\ [x_i = y_i] \\ & + \text{prob}\ [D(x') = 1 | x \in_B \{0,1\}^n, (y_i, \ldots, y_n) \in_U \{0,1\}^{n-i}, x_i \neq y_i] \cdot \text{prob}\ [x_i \neq y_i] \end{aligned}$$

As y_i is chosen randomly in $\{0, 1\}$ the probability of y_i to equal x_i is $\frac{1}{2}$ independent from the way, x_i was generated.

$$\begin{aligned} = \quad & \text{prob}\ [D(x') = 1 | x \in_B \{0,1\}^n, (y_i, \ldots, y_n) \in_U \{0,1\}^{n-i}, x_i = y_i] \cdot \frac{1}{2} \\ & + \text{prob}\ [D(x') = 1 | x \in_B \{0,1\}^n, (y_i, \ldots, y_n) \in_U \{0,1\}^{n-i}, x_i \neq y_i] \cdot \frac{1}{2} \end{aligned}$$

What we will need further on is the following reformulation of the above:

$$\begin{aligned} & \text{prob}\ [D(x') = 1 | x \in_B \{0,1\}^n, (y_i, \ldots, y_n) \in_U \{0,1\}^{n-i}, x_i \neq y_i] \cdot \frac{1}{2} \\ & = \text{prob}\ [D(x') = 1 | x' \in_{B_{i-1}} \{0,1\}^n] \qquad (4.3) \\ & - \text{prob}\ [D(x') = 1 | x \in_B \{0,1\}^n, (y_i, \ldots, y_n) \in_U \{0,1\}^{n-i}, x_i = y_i] \cdot \frac{1}{2} \end{aligned}$$

Now we can look at $\sigma_P(B)$, the success rate of P.

$$\sigma_P(B) \quad = \quad \text{prob}\,[P(\pi_{i-1}(x)) = x_i | x \in_B \{0,1\}^n]$$

This probability can also be divided up into the cases $x_i = y_i$ and $x_i \neq y_i$:

$$\text{prob}\,[D(x') = 1 | x' \in_{B_{i-1}} \{0,1\}^n]$$

$$= \text{prob}\,[P(\pi_{i-1}(x)) = x_i | x \in_B \{0,1\}^n, (y_i, \dots, y_n) \in_U \{0,1\}^{n-i}, x_i = y_i] \cdot \text{prob}\,[x_i = y_i]$$

$$+\text{prob}\,[P(\pi_{i-1}(x)) = x_i | x \in_B \{0,1\}^n, (y_i, \dots, y_n) \in_U \{0,1\}^{n-i}, x_i \neq y_i] \cdot \text{prob}\,[x_i \neq y_i]$$

Now we use the fact, that $P(\pi_{i-1}(x)) = x_i$ if and only if $D(x') = 1$ and $x_i = y_i$ or $D(x') = 0$ and $x_i \neq y_i$. Additionally, we again use the fact, that the probabilities of $x_i = y_i$ and $x_i \neq y_i$ are both $\frac{1}{2}$.

$$= \text{prob}\,[D(x') = 1 | x \in_B \{0,1\}^n, (y_i, \dots, y_n) \in_U \{0,1\}^{n-i}, x_i = y_i] \cdot \frac{1}{2}$$

$$+\text{prob}\,[D(x') = 0 | x \in_B \{0,1\}^n, (y_i, \dots, y_n) \in_U \{0,1\}^{n-i}, x_i \neq y_i] \cdot \frac{1}{2}$$

$$= \frac{1}{2} \cdot \text{prob}\,[D(x') = 1 | x \in_B \{0,1\}^n, (y_i, \dots, y_n) \in_U \{0,1\}^{n-i}, x_i = y_i]$$

$$+\frac{1}{2} \cdot (1 - \text{prob}\,[D(x') = 1 | x \in_B \{0,1\}^n, (y_i, \dots, y_n) \in_U \{0,1\}^{n-i}, x_i \neq y_i])$$

$$= \frac{1}{2} \cdot \text{prob}\,[D(x') = 1 | x \in_B \{0,1\}^n, (y_i, \dots, y_n) \in_U \{0,1\}^{n-i}, x_i = y_i]$$

$$+\frac{1}{2} - \frac{1}{2} \cdot \text{prob}\,[D(x') = 1 | x \in_B \{0,1\}^n, (y_i, \dots, y_n) \in_U \{0,1\}^{n-i}, x_i \neq y_i])$$

Now we can use equation 4.3 on page 25:

$$
\begin{aligned}
= \; & \frac{1}{2} \cdot \text{prob}\left[D(x') = 1 | x \in_B \{0,1\}^n, (y_i, \ldots, y_n) \in_U \{0,1\}^{n-i}, x_i = y_i\right] \\
& + \frac{1}{2} - (\text{prob}\left[D(x') = 1 | x' \in_{B_{i-1}} \{0,1\}^n\right] \\
& - \frac{1}{2} \cdot \text{prob}\left[D(x') = 1 | x \in_B \{0,1\}^n, (y_i, \ldots, y_n) \in_U \{0,1\}^{n-i}, x_i = y_i\right])
\end{aligned}
$$

$$
\begin{aligned}
= \; & \frac{1}{2} \cdot \text{prob}\left[D(x') = 1 | \underbrace{x \in_B \{0,1\}^n, (y_i, \ldots, y_n) \in_U \{0,1\}^{n-i}, x_i = y_i}_{x' \in_{B_i} \{0,1\}^n}\right] \\
& + \frac{1}{2} - \text{prob}\left[D(x') = 1 | x' \in_{B_{i-1}} \{0,1\}^n\right] \\
& + \frac{1}{2} \cdot \text{prob}\left[D(x') = 1 | \underbrace{x \in_B \{0,1\}^n, (y_i, \ldots, y_n) \in_U \{0,1\}^{n-i}, x_i = y_i}_{x' \in_{B_i} \{0,1\}^n}\right]
\end{aligned}
$$

$$
\begin{aligned}
= \; & \frac{1}{2} \cdot \text{prob}\left[D(x') = 1 | x' \in_{B_i} \{0,1\}^n\right] \\
& + \frac{1}{2} - \text{prob}\left[D(x') = 1 | x' \in_{B_{i-1}} \{0,1\}^n\right] \\
& + \frac{1}{2} \cdot \text{prob}\left[D(x') = 1 | x' \in_{B_i} \{0,1\}^n\right]
\end{aligned}
$$

$$
= \; \text{prob}\left[D(x') = 1 | x' \in_{B_i} \{0,1\}^n\right] + \frac{1}{2} - \text{prob}\left[D(x') = 1 | x' \in_{B_{i-1}} \{0,1\}^n\right]
$$

$$
= \; \frac{1}{2} + \text{prob}\left[D(x') = 1 | x' \in_{B_i} \{0,1\}^n\right] - \text{prob}\left[D(x') = 1 | x' \in_{B_{i-1}} \{0,1\}^n\right]
$$

And with equation 4.2 on page 24 that is

$$
\geq \; \frac{1}{2} + \frac{\delta_D(n)}{n}
$$

Altogether we get:

$$\sigma_P(B) \geq \frac{1}{2} + \frac{\delta_D(n)}{n}$$

Therefore P has polynomial advantage and we have proven that P is a $(i, \sigma_P(B))$-predictor. That finishes the proof of Yao's Theorem 22.

$\square$

5 Existence of Pseudorandom Generators

5.1 Significance of Pseudorandom Generators

Pseudorandom Generators are "Randomness-Expander". With a relatively small amount of random input, they produce a very long sequence that to any polynomial time observer "looks random". The implementation of most cryptographic applications require "random sequences". The remarkable property of pseudorandom generators is, that *any* pseudorandom generator can be used to produce the needed sequences for *any* application. If there would be an application that would suffer a noticeable degradation of performance, this application could be turned into a successful adversary for the pseudorandom generator that was used. And therefore that generator would not be a pseudorandom generator. This implies, that once an algorithm has been verified to be a pseudorandom generator, it can be used for any efficient application without the need of testing the performance of the generator in the specific new application.

5.2 Pseudorandom Generators from One-Way Functions

In this chapter, we discuss the construction of pseudorandom generators in general. In the following chapters in order to construct pseudorandom function families we will use special pseudorandom generators. Therefore, we will now prove that pseudorandom generators exist if length-preserving one-way permutations exist. Although it has been shown [15], that the existence of one-way-functions suffices, we will take the existence of this special form of one-way-functions as prerequisite. The proof works in two steps. The first step is to show the existence of a so-called "extender". This is a pseudorandom generator that extends a random seed by one bit such that the resulting bit string is pseudorandom. The existence of such an extender relies on the existence of a hard core predicate. The proof shows, that if

such an extender does not exist, then it is possible to construct a predictor for the hard core predicate. And as that is a contradiction, the existence of an extender is therefore proven.
The second step is to show that the repeated application of an extender results in a pseudorandom generator that stretches a random seed into a pseudorandom bit string which is polynomial in the seed-length. This part of the proof is an application of the Hybrid Theorem 3.0.1 on page 13 in Chapter 3.

Theorem 5.2.1 (Existence of Pseudorandom Generators)
Let $f : \{0,1\}^n \to \{0,1\}^n$ be a length preserving one-way permutation.
Then for all polynomials p there exists a pseudorandom generator
$g^p : \{0,1\}^n \to \{0,1\}^{p(n)}$.

Proof:

To prove the theorem, we need to show two things:

1. There exists a pseudorandom generator

$$g : \{0,1\}^n \to \{0,1\}^{n+1} \text{ [extender]}$$

2. For all polynomials p there exists a pseudorandom generator

$$g^p : \{0,1\}^n \to \{0,1\}^{p(n)}$$

Proof Part 1:

Let $B : \{0,1\}^n \to \{0,1\}$ be a hard core predicate for f. That is to say, given $x \in \{0,1\}^n$ it is easy to compute $B(x)$, but given $f(x)$ it is hard to compute $B(x)$. Then we define g to be

$$\begin{aligned} g \quad &: \quad \{0,1\}^n \to \{0,1\}^{n+1}, \\ g(x) \quad &:= \quad f(x) \circ B(x) \end{aligned}$$

This function is certainly computable in polynomial time. We are going to prove that it is also pseudorandom.

Let G_{n+1} be the probability distribution generated by g and let U_{n+1} be the uniform probability distribution in the same range. Let us assume that g is not pseudorandom. Then there exists a distinguisher D for G_{n+1} and U_{n+1}. This distinguisher has success probability

$$\begin{aligned}\delta(n+1) &= \text{prob}[D(x \circ b) = 1 | x = (x_1 \dots x_n) \in_U \{0,1\}^n, b = B(x)] \\ &- \text{prob}[D(x \circ b) = 1 | x = (x_1 \dots x_n) \in_U \{0,1\}^n, b \in_U \{0,1\}]\end{aligned}$$

As f is a permutation, it follows that if x is random, than so is $f(x)$. Therefore

$$\begin{aligned}\delta(n+1) &= \text{prob}[D(f(x) \circ b) = 1 | x = (x_1 \dots x_n) \in_U \{0,1\}^n, b = B(x)] \\ &- \text{prob}[D(f(x) \circ b) = 1 | x = (x_1 \dots x_n) \in_U \{0,1\}^n, b \in_U \{0,1\}] \\ &\geq \frac{1}{n^k}\end{aligned}$$

for some $k \in \mathbb{N}$ and n big enough. Now let us have a closer look at the probability that D outputs 1.

$$\begin{aligned}&\text{prob}[D(f(x) \circ b) = 1 | x = (x_1 \dots x_n) \in_U \{0,1\}^n, b \in_U \{0,1\}] \\ = \;& \text{prob}[D(f(x) \circ b) = 1 | x = (x_1 \dots x_n) \in_U \{0,1\}^n, b = B(x)] \cdot \text{prob}[b = B(x)] \\ + \;& \text{prob}[D(f(x) \circ b) = 1 | x = (x_1 \dots x_n) \in_U \{0,1\}^n, b = \overline{B(x)}] \cdot \text{prob}[b = \overline{B(x)}]\end{aligned}$$

In order to simplify notation, let

$$\begin{aligned}\alpha &:= \text{prob}[D(f(x) \circ b) = 1 | x = (x_1 \dots x_n) \in_U \{0,1\}^n, b = B(x)] \\ \beta &:= \text{prob}[D(f(x) \circ b) = 1 | x = (x_1 \dots x_n) \in_U \{0,1\}^n, b = \overline{B(x)}]\end{aligned}$$

As b is chosen randomly in $\{0,1\}$, we know that prob $[b = B(x)] = \frac{1}{2}$ and prob $[b = \overline{B(x)}] = \frac{1}{2}$. Note, that this does not depend on $B(x)$. Hence:

$$\text{prob}[D(f(x) \circ b) = 1 | x \in_U \{0,1\}^n, b \in_U \{0,1\}] = \frac{1}{2}(\alpha + \beta)$$

Now the success probability $\delta(n+1)$ of D can be expressed in terms of α and β.

$$\begin{aligned}
\delta(n+1) &= \underbrace{\text{prob}[D(f(x) \circ b) = 1 | x = (x_1 \dots x_n) \in_U \{0,1\}^n, b = B(x)]}_{\alpha} \\
&- \underbrace{\text{prob}[D(f(x) \circ b) = 1 | x = (x_1 \dots x_n) \in_U \{0,1\}^n, b \in_U \{0,1\}]}_{\frac{1}{2}(\alpha+\beta)} \\
&= \alpha - \frac{1}{2}(\alpha + \beta)
\end{aligned}$$

$$= \frac{1}{2}(\alpha - \beta) > \frac{1}{n^k} \tag{5.1}$$

The next step will be to use D in order to construct a predictor P for $B(x)$ with a polynomial advantage. As the existence of such a predictor is a contradiction to the fact that $B(x)$ is a hard core predicate for $f(x)$, we conclude, that the distinguisher D cannot exist. Therefore, we have shown that the probability distribution G_{n+1} is pseudorandom and that g is an extender, which is a pseudorandom generator.
As G_{n+1} is a probability distribution on $\{0,1\}^{n+1}$ and D is a distinguisher for G_{n+1} and U_{n+1}, the prerequisites for Yao's Theorem on page 22 are given. But we do not know, for which bit $i \in \{1, \dots, n+1\}$ we can apply the theorem. Yao's Theorem only tells us, that such a bit exists. It doesn't give us an efficient way to determine this i.
What we need here, is a predictor for the last bit $B(x)$, generated according to G_{n+1}. Therefore we will have to modify the proof of Yao's Theorem for this special case. Let us start with a description of the predictor P for the hard core predicate $B(x)$.

Predictor P:

Input: $f(x) \in \{0,1\}^n$
Output: $P(f(x)) \in \{0,1\}$

- Choose $b \in_U \{0,1\}$
- Run $D(f(x) \circ b)$
- $P(f(x)) := D(f(x) \circ b) \oplus b \oplus 1$

P takes as input a bit string $f(x) \in \{0,1\}^n$. This bit string is the first part of a bit string sampled according to G_{n+1}. The last bit is missing. What P does next, is to sample a random bit b and attach it to the input $f(x)$. Then P runs D with this concatenated bit string and the output of D decides the output of $P(f(x))$. If $D(f(x) \circ b) = 1$, D thinks, that its input was generated according to G_{n+1}. This implies, that it is more likely, that the randomly sampled bit b equals $B(x)$. Therefore, P takes its original random choice b as output. If $D(f(x) \circ b) = 0$, D thinks, that its input was not generated according to G_{n+1}. As the first n bits of the input of D are actually generated according to G_{n+1}, this implies, that it is more likely, that the randomly sampled bit b does not equal $B(x)$. Therefore, P takes the negation of its original random choice b as output.
The predictor P is a probabilistic algorithm, as b is sampled uniformly in $\{0,1\}$. If this random bit is available, all the computations in P can be done in polynomial time. Provided that, $D(f(x) \circ b)$ can be computed in polynomial time. But that is the initial assumption that we are going to lead to a contradiction.
Now let us calculate the success probability of P. The success probability $\sigma(G_{n+1})$ of P is better than guessing by the amount of the probability, that "D is right", i.e., $\sigma_P(G_{n+1}) = \delta_D(n+1) + \frac{1}{2}$. That is what we are going to prove now.

$$
\begin{aligned}
\sigma(G_{n+1}) &= \operatorname{prob}[P(f(x)) = B(x) | f(x) \circ B(x) \in_{G_{n+1}} \{0,1\}^{n+1}] \\
&= \operatorname{prob}[P(f(x)) = B(x) | x \in_U \{0,1\}^n] \\
&= \overbrace{\operatorname{prob}[D(f(x) \circ b) = 1 | x \in_U \{0,1\}^n, b = B(x)]}^{\alpha} \cdot \overbrace{\operatorname{prob}[b = B(x)]}^{\frac{1}{2}} \\
&+ \underbrace{\operatorname{prob}[D(f(x) \circ b) = 0 | x \in_U \{0,1\}^n, b = \overline{B(x)}]}_{1-\beta} \cdot \underbrace{\operatorname{prob}[b = \overline{B(x)}]}_{\frac{1}{2}} \\
&= \frac{1}{2}\alpha + \frac{1}{2}(1-\beta) \\
&= \frac{1}{2} + \frac{1}{2}(\alpha - \beta)
\end{aligned}
$$

Now we use equation 5.1 on page 32 and get:

$$
\sigma(G_{n+1}) > \frac{1}{2} + \frac{1}{n^k}
$$

This proves, that P is a $(n+1, \sigma(G_{n+1}))$-predictor.

But predicting the last bit under G_{n+1} means predicting the hard core predicate of f. And by the definition of hard core predicates, it is not possible to compute $B(x)$ from $f(x)$ in probabilistic polynomial time. Therefore, such a predictor cannot exist. Now we have a contradiction. Therefore, our first assumption that G_{n+1} is not a pseudorandom probability distribution is wrong.
This implies that

$$g : \{0,1\}^n \to \{0,1\}^{n+1} : x \to f(x) \circ B(x)$$

is an extender.

$\square$

Proof Part 2:

The second part of the proof is to show, that for all polynomials p there exists a pseudorandom generator $g^p : \{0,1\}^n \to \{0,1\}^{p(n)}$. This can be done by an application of the hybrid argument. But first we need to construct g^p. In order to do that, we first apply g on x. Next, we apply g on the first n output bits of $g(x)$, which is $f(x)$. Then we apply g on the first n output bits of $B(f(x))$, which is $f^2(x)$ and so on. Altogether that amounts to $p(n)$ many applications of g.

$$\begin{array}{llcccc}
x & \to & g(x) & = & f(x) & \circ & B(x) \\
 & \to & g(f(x)) & = & f(f(x)) & \circ & B(f(x)) \\
 & \vdots & & & & & \\
 & \to & g(f^{p(n)-1}(x)) & = & f^{p(n)}(x) & \circ & B(f^{p(n)-1}(x))
\end{array}$$

Now we can construct the generator g^p for any input $x \in \{0,1\}^n$. For any $x \in \{0,1\}^n$ the output of $g^p(x)$ is the string concatenation of all the hard core predicate bits $B(x), B(f(x)), \ldots, B(f^{p(n)-1}(x))$.

$$g^p(x) = B(x) \circ B(f(x)) \circ \cdots \circ B(f^{p(n)-1}(x))$$

This construction is called the "Blum-Micali-paradigm" [22, 4]. We will use it to prove the existence of pseudorandom generators.

Let us apply the Hybrid Theorem 3.0.1 on page 13 with the uniform distribution $U_{p(n)}$ as probability distribution $A^{p(n)}$ and $G_{p(n)}$, the probability distribution generated by $g^{p(n)}$ as $B^{p(n)}$. The prerequisite of applying the Hybrid Theorem is, that single bits sampled from A and B are indistinguishable. In our case, bits from A are uniformly distributed bits and bits from B are hard core bits. These bits are indistinguishable by prerequisite.
That finishes the proof of Theorem 5.2.1 on page 30. As A and B are indistinguishable, so are $A^{p(n)} = U_{p(n)}$ and $B^{p(n)} = G_{p(n)}$.
Thus, g^p is a pseudorandom generator and therefore for all polynomials p there exists a pseudorandom generator, that expands a random n-bit seed into a pseudorandom $p(n)$-bit string.

$\square$

6 General Construction of Pseudorandom Functions

6.1 Significance of Pseudorandom Functions

In this chapter we are going to use pseudorandom generators to construct pseudorandom function families. To justify this effort let us first look at weaknesses of pseudorandom generators. One example for a situation, where a pseudorandom generator proves to be inefficient was given in the introduction. Implementing a block cryptosystem in a multiplayer environment requires pseudorandom functions for efficiency.

But there also is a general weakness of pseudorandom generators that motivated Goldreich, Goldwasser and Micali in [12] to introduce the notion of pseudorandom functions.

A pseudorandom generator produces pseudorandom bit strings of polynomial length. With most pseudorandom generators it is easily possible to extend the output to exponentially long bit strings. And that leads to an open problem, the *easy-access problem*. It is an open question suggested in [4] that arises when those exponentially far away bits are easily accessible. The question is, whether a pseudorandom generator remains pseudorandom even when an adversary has access to those exponentially far away bits. This problem does not arise, when one deals with pseudorandom functions instead of pseudorandom generators.

A pseudorandom function is a function that to an adversary looks randomly chosen in a given domain and range. Such an adversary has black box access to that function and can evaluate the function at a polynomial number of interactively chosen inputs. It does not have access to a full description of the function in question.

The description of a true random function is exponentially long and thus in this sense the function looks like an exponentially long random bit string. An adversary gets access to a polynomial number of bits which can be chosen interactively and can be exponentially far away from each other.

And in opposition to pseudorandom generators it is provable, that pseudorandom functions stay pseudorandom when those exponentially far away bits are accessible. In this chapter we are going to give the proof of the existence of pseudorandom functions based on the existence of pseudorandom generators.

6.2 Pseudorandom Functions from Pseudorandom Generators

The general construction of pseudorandom function families works by taking a pseudorandom generator and constructing a pseudorandom function from each possible input for the pseudorandom generator. This construction can be done with any pseudorandom generator. We have proven, that pseudorandom generators with output length polynomial in its input length exist. That implies, that there exist length doubling pseudorandom generators. Therefore, it is enough to show the construction of pseudorandom function families from length doubling pseudorandom generators. That is what we will do now.

6.2.1 The Generator

Let $g : \{0,1\}^n \to \{0,1\}^{2n}, n \in \mathbb{N}$ be such a length-doubling pseudorandom generator. For every input $x \in \{0,1\}^n$ we define the output of g to be the concatenation of two n-bit strings $g_0(x)$ and $g_1(x)$:

$$g(x) = \langle g_0(x), g_1(x) \rangle$$

As g is a pseudorandom generator, there exists no distinguisher D for g that can distinguish a random string of length $2n$ from a string generated by g in probabilistic polynomial time. Let G be the probability distribution generated by g. Let $U = U_{2n}$ be the uniform probability distribution in the same range. Then for all probabilistic polynomial time algorithms D and for all $k \in \mathbb{N}$:

$$\text{prob}[D(s) = 1 | s \in_G \{0,1\}^{2n}] - \text{prob}[D(s) = 1 | s \in_U \{0,1\}^{2n}] < \frac{1}{n^k}$$

6.2.2 The Construction

Let $F = \{F_n\}_{n \in \mathbb{N}} : \{0,1\}^n \times \{0,1\}^{l(n)} \to \{0,1\}^n$ be a function family. Then for all $n \in \mathbb{N}$, for all $y \in \{0,1\}^n$, for all $x = (x_1 \ldots x_{l(n)}) \in \{0,1\}^{l(n)}$, and for all $f_y \in F_n$ we define $f_y(x)$ inductively for all $x \in \{0,1\}^{\leq l(n)}$, starting with the empty word λ as input:

$$
\begin{aligned}
f_y(\lambda) &= y \\
f_y(x_1) &= g_{x_1}(f_y(\lambda)) = g_{x_1}(y) \\
f_y(x_1, x_2) &= g_{x_2}(f_y(x_1)) \\
&\vdots \\
f_y(x_1 \dots x_i) &= g_{x_i}(f(x_1 \dots x_{i-1})) \\
&\vdots \\
f_y(x) &= f_y(x_1 \dots x_{l(n)}) = g_{x_{l(n)}}(f_y(x_1 \dots x_{l(n)-1}))
\end{aligned}
$$

Although f_y is defined for all inputs of length $\leq l(n)$, we are only interested in the inputs $x \in \{0,1\}^{l(n)}$ of length $l(n)$ for f_y.

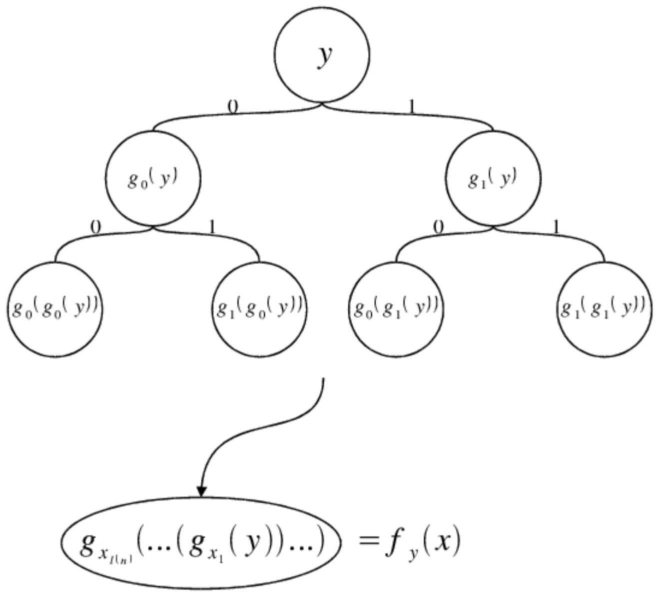


Abbildung 6.1: General Construction

Figure 6.1 represents the construction of the random function. It is a full binary tree of depth $l(n)$. In the root there is the random n-bit string y stored. In the nodes of succeeding levels, the tree stores n-bit string deterministically computed by applying g in the following way. If the n bit string s is stored in an inner node v, then $g_0(s)$ is stored in v's left son and $g_1(s)$ is stored in v's right son.
Therefore, for any $x \in \{0,1\}^{l(n)}$, the value $f_y(x)$ is stored in the leaf reachable from the root following the edge-path labeled with x.

Theorem 6.2.1 (Pseudorandom Function Family)
Let $F = \{F\}_n : \{0,1\}^n \times \{0,1\}^{l(n)} \rightarrow \{0,1\}^n$ be a function family. For all $y \in \{0,1\}^n$ and all $x \in \{0,1\}^{l(n)}$ let $f_y \in F_n, f_y(x) : \{0,1\}^{l(n)} \rightarrow \{0,1\}^n$ be defined as above. Then $F = \{F\}_n, n \in \mathbb{N}$ is a pseudorandom function family.

Proof:

Now, we need to prove that the function family constructed above is efficiently computable and pseudorandom.
In order to show that F is efficiently computable, we need to show two things. First, all $f_y \in F$ have to have an index that makes sampling them easy [Indexing]. Second, there needs to exist a polynomial time algorithm that upon input $x \in \{0,1\}^{l(n)}$ and $y \in \{0,1\}$ can compute $f_y(x)$[polynomial time evaluation].

(1) *Indexing:*
Each function f_y has a unique n-bit key y which is sampled uniformly in $\{0,1\}^n$. Thus, picking a function f_y randomly in F is easy if n random bits are available.

(2) *Polynomial Time Evaluation:*
The pseudorandom generator g is computable in polynomial time. The above inductive construction of f_y then shows, that for all $y \in \{0,1\}^n$ and for all $x \in \{0,1\}^{l(n)}$ the function $f_y(x)$ is computable in polynomial time.

As all $f_y \in F$ have the properties of indexing and polynomial time evaluation, F is efficiently computable.
Now we need to show that F is pseudorandom. When looking at function families, "pseudorandom" means that there exists no distinguisher for any function from this function family and any function selected randomly from all functions with the same domain and range. This is again a question of distinguishing two probability distributions. In order to prove the indistinguishability of these two distributions, we use the hybrid argument. In the chapters before, we used the hybrid argument to prove the indistinguishability of bit strings. The difference in this case is, that we already know that there is no distinguisher for a bit string generated by g and a random bit string of the same length. Instead, we want to prove that there is

no distinguisher for a *function* generated by g and a random function in the same domain and range. As before, the proof works by contradiction. We assume that there exists a distinguisher D for a function generated by g and a truly random function. Then, by using the hybrid argument, we will show that in consequence there exists a distinguisher D' for a bit string generated by g and a random bit string. Therefore, g cannot be a pseudorandom generator. That completes the proof as the existence of the distinguisher D' is a contradiction to the fact that g is a pseudorandom generator. Therefore, the functions in F are pseudorandom and F is a pseudorandom function family.

Construction of D':

The construction of D' with input $s = \langle s_0, s_1 \rangle \in \{0,1\}^{2n}$ works by simulating D with a hybrid function h as input. Then the output of D' is the output of D, i.e., $D'(s) = D(h)$.
This hybrid function h is constructed similarly to the pseudorandom functions f_y. Where f_y is constructed from a pseudorandom generator g, h is constructed from a function g' in the same domain and range as g, i.e.,:

$$\begin{aligned} g' &: \{0,1\}^n \to \{0,1\}^{2n} \\ g'(x) &= \langle g'_0(x), g'_1(x) \rangle \end{aligned}$$

The main difference between g and g' is, that g' starts with having truly random $2n$-bit strings as output. Then at a certain point, g' starts behaving like the pseudorandom generator g. This point of change is chosen randomly by the distinguisher D'. Let $m(n)$ be the maximum number of queries D makes on h. Each of those queries is $l(n)$ bits long. Let

$$\begin{aligned} &(q^1, \ldots, q^{m(n)}), \\ q^i &= \langle q^i_1, \ldots, q^i_{l(n)} \rangle \in \{0,1\}^{l(n)} \end{aligned}$$

be those queries. For each query D gets n output bits from h. Let

$$\begin{aligned} &(o^1, \ldots, o^{m(n)}), \\ o^i &= \langle o^i_1, \ldots, o^i_n \rangle \in \{0,1\}^n \end{aligned}$$

be those outputs. To compute the output for one query, D' needs to compute in sequence the output value of h for $l(n)$ input values. Therefore, there are $l(n)m(n)$ input/output-pairs that D' needs to calculate. The output of h with the empty word

λ as input is defined to be a random value, i.e., $h(\lambda) := r \in_U \{0,1\}^n$.
The $l(n)m(n)$ input/output-pairs are:

$$\begin{array}{ccccccc}
 & & h(\lambda) & = & r \in_U \{0,1\}^n & & \\
 & & h(q_1^1) & = & g'_{q_1^1}(h(\lambda)) = g'_{q_1^1}(r) & & \\
 & & \vdots & & & & \\
 & & h(q_1^1 \dots q_k^1) & = & g'_{q_k^1}(h(q_1^1 \dots q_{k-1}^1)) & & \\
 & & \vdots & & & & \\
h(q^1) & = & h(q_1^1 \dots q_{l(n)}^1) & = & g'_{q_{l(n)}^1}(h(q_1^1 \dots q_{l(n)-1}^1)) & = & o^1 \\
 & & h(q_1^2) & = & g'_{q_1^2}(h(\lambda)) = g'_{q_1^2}(r) & & \\
 & & \vdots & & & & \\
 & & h(q_1^2 \dots q_k^2) & = & g'_{q_k^2}(h(q_1^2 \dots q_{k-1}^2)) & & \\
 & & \vdots & & & & \\
h(q^2) & = & h(q_1^2 \dots q_{l(n)}^2) & = & g'_{q_{l(n)}^2}(h(q_1^2 \dots q_{l(n)-1}^2)) & = & o^2 \\
 & & \vdots & & & & \\
 & & \vdots & & & & \\
 & & h(q_1^{m(n)}) & = & g'_{q_1^{m(n)}}(h(\lambda)) = g'_{q_1^{m(n)}}(r) & & \\
 & & \vdots & & & & \\
 & & h(q_1^{m(n)} \dots q_k^{m(n)}) & = & g'_{q_k^{m(n)}}(h(q_1^{m(n)} \dots q_{k-1}^{m(n)})) & & \\
 & & \vdots & & & & \\
h(q^{m(n)}) & = & h(q_1^{m(n)} \dots q_{l(n)}^{m(n)}) & = & g'_{q_{l(n)}^{m(n)}}(h(q_1^{m(n)} \dots q_{l(n)-1}^{m(n)})) & = & o^{m(n)}
\end{array}$$

To make sure that h is a function, D' stores all input/output-pairs that are computed during the course of the simulation of D. Whenever D' needs to compute a new input/output-pair, D' first checks whether the input is already stored and in this case answers with the previously stored output value.

Distinguisher D':

Input : $s = \langle s_0, s_1 \rangle \in \{0,1\}^{2n}$
Output : $D'(s) \in \{0,1\}$
Output 0 means, that D' decides that s is random.
Output 1 means, that D' decides that s is pseudorandom.

1. Pick two random values $i \in_U \{0, \ldots, m(n)-1\}$ and $j \in_U \{1, \ldots, l(n)\}$

2. For $u = 1, \ldots, i$ and $v = 1, \ldots, j-1$ compute:

$$h(q_1^u \ldots q_v^u) = g'_{q_v^u}(h(q_1^u \ldots q_{v-1}^u))$$

   If $g'_{q_v^u}(h(q_1^u \ldots q_{v-1}^u))$ has been calculated before, take the stored output-value. If $g'_{q_v^u}(h(q_1^u \ldots q_{v-1}^u))$ has not been calculated before, choose a random n-bit string $r \in_U \{0,1\}^n$ and take it as output value. Store the new input/output-pair.

3. If $q_j^{i+1} = 0$ set:

$$h(q_1^{i+1} \ldots q_j^{i+1}) = g'_{q_j^{i+1}}(h(q_1^{i+1} \ldots q_{j-1}^{i+1})) := s_0$$

   if $q_j^{i+1} = 1$ set:

$$h(q_1^{i+1} \ldots q_j^{i+1}) = g'_{q_j^{i+1}}(h(q_1^{i+1} \ldots q_{j-1}^{i+1})) := s_1$$

4. For $u = i+2, \ldots, m(n)$ and $v = j+1, \ldots, l(n)$ define:

$$h(q_1^u \ldots q_v^u) = g'_{q_v^u}(h(q_1^u \ldots q_{v-1}^u)) := g_{q_v^u}(h(q_1^u \ldots q_{v-1}^u))$$

5. Run $D(h)$

6. $D'(s) := D(h)$

By picking two random values in the first step, the distinguisher D' determines one random bit, the $(i \cdot l(n) + j)$'s bit within the output of h, where the input s of D' is integrated in h.
The random choice D' makes, does not only determine from which query on h behaves pseudorandomly. The random point $i \cdot l(n) + j$ can as well be within one query.

The choice of i determines within which query the change from pseudorandom to random behaviour takes place. The choice of j determines the exact position within the $(i+1)$'th query where the change from random to pseudorandom behaviour takes place.
The generator g' of the function h first outputs random bit strings. Then at the randomly chosen point $i \cdot l(n) + j$ the bit string s determines the output of g'. From then on, g' works just like g and outputs pseudorandom bit strings.
For the function h, this has the consequence that h starts with behaving like a random function. Then D''s random choice of i and j determines the point $i \cdot l(n) + j$, where h starts to behave like a pseudorandom function. At this point $i \cdot l(n) + j$, the input s of D' is integrated into h. The effect is that, if s is random, h starts to behave pseudorandomly at the point $i \cdot l(n) + j + 1$ and therefore behaves "more randomly". If s is pseudorandom h starts to behave pseudorandomly at the point $i \cdot l(n) + j$ and therefore behaves "more pseudorandomly".
Here we can apply the hybrid argument. The function h can just as well be regarded as a hybrid distribution. Each choice of i and j leads to another hybrid function or hybrid distribution. As $i \in \{0, \ldots, m(n) - 1\}$ and $j \in \{1, \ldots, l(n)\}$ there are $l(n) \cdot m(n)$ many possible hybrid functions:

$$h_1 \;:\; \underbrace{\overbrace{\boxed{\text{1.query}}}^{l(n)} \quad \ldots \quad \overbrace{\boxed{\text{m(n).query}}}^{l(n)}}_{\text{pseudorandom}}$$

$$\vdots$$

$$h \;:\; \underbrace{\overbrace{\boxed{\text{1.query}}}^{l(n)} \quad \ldots}_{\text{random}} \; \underset{\uparrow j}{\boxed{\text{(i+1).query}}} \; \underbrace{\ldots \quad \overbrace{\boxed{\text{m(n).query}}}^{l(n)}}_{\text{pseudorandom}}$$

$$\vdots$$

$$h_{l(n)m(n)} \;:\; \underbrace{\overbrace{\boxed{\text{1.query}}}^{l(n)} \quad \ldots \quad \overbrace{\boxed{\text{m(n).query}}}^{l(n)}}_{\text{random}}$$

For the choice $i = 0$ and $j = 1$, we get the hybrid function h_1. This implies that h_1 is the function where the change takes place in the first query and the first call of g' within that query. And if s is pseudorandom, then h_1 is a pseudorandom function. For the choice $i = m(n) - 1$ and $j = l(n)$ we get the hybrid function $h_{l(n)m(n)}$. Analogously, $h_{l(n)m(n)}$ is the function where the change takes place in the

last query and within that query in the last call of g'. If s is a random bit string, then $h_{l(n)m(n)}$ is a random function.
In the application of the Hybrid Theorem 3.0.1 on page 13 these two hybrid functions are the two extreme distributions. Each hybrid function can be regarded as a string of bit strings, as can be seen in the above picture. Then the bit strings in the Hybrid Theorem 3.0.1 on page 13 correspond to these strings of bit strings here. The probability distribution B^n in the Hybrid Theorem here corresponds to the uniform probability distribution on the set of all functions with domain $l(n)$ and range n, $F_{l(n),n}$. The probability distribution A^n in the Hybrid Theorem here corresponds to the probability distribution on $F_{l(n),n}$ given by the function family F. In the Hybrid Theorem single bits from the two different probability distributions A and B are indistinguishable. Here, that corresponds to the indistinguishability of a bit string of length $2n$ generated by g or chosen randomly in $\{0,1\}^{2n}$. By applying the Hybrid Theorem, we therefore get the desired result. If g is a pseudorandom generator, then F is a pseudorandom function family.

□

Efficiency

Let us now look at the efficiency of the pseudorandom functions we just constructed. To evaluate any function $f_y \in F_n$ at any given point $x \in \{0,1\}^n$ there are $l(n)$ many evaluations of the pseudorandom generator g necessary. Therefore it is not possible to say, whether this construction is efficient enough for practical applications without knowing, which pseudorandom generator is used in the construction. That leads us to the next chapters, where we will examine special constructions of pseudorandom function families. There it is possible to examine the efficiency of the constructed pseudorandom functions more closely.

7 Special Construction of Naor and Reingold

In this chapter we will describe and examine the construction of a pseudorandom function family. The construction and therefore the security of this function family is based on the decisional version of the Diffie-Hellman-Assumption, the DDH-Assumption. We will prove the pseudorandomness of this construction given the Diffie-Hellman-Assumption.
What we will get is a pseudorandom function family whose construction is as secure as the decisional Diffie-Hellman-Assumption. The such constructed pseudorandom functions can be evaluated at any given point at the cost of two multiple products. Furthermore those functions can be constructed by a constant depth circuit consisting of a polynomial number of treshold gates.
There is an additional feature to the special algebraic structure of this constructions. This special features make it possible to give a zero-knowledge proof for statements of the form $y = f_s(x)$ and $y \neq f_s(x)$. Details on the feature are given in [20].
In addition, Naor and Reingold also give a very closely related construction of a pseudorandom function family in [20]. This second construction relies on another version of the Diffie-Hellman-Assumption, the General Diffie-Hellman-Assumption. The proof of security of this second construction is completely analogous to the one we will give here. Details can be seen in [20].
We now turn to the prerequisites of the constructions of Naor and Reingold.

7.1 Prerequisites

First, we are going to describe special versions of the Diffie-Hellman (DH)-Assumption, which are the Decisional Diffie-Hellman(DDH)-Assumption, the Computational Diffie-Hellman(CDH)-Assumption and the General Diffie-Hellman(GDH)-Assumption.
Generally speaking, all these assumptions deal with solving discrete logarithms. In order to specify them, we will need a set of variables.
Let P be a large n-bit prime number. Let Q be a large prime divisor of $P-1$. We will be working in $\mathbb{Z}_P^*$, the multiplicative group modulo P. Let g be a uniformly

chosen generator of $\mathbb{Z}_Q$, a subgroup of $\mathbb{Z}_P^*$. Let a, b, and c be uniformly chosen in $\mathbb{Z}_Q$. Then all the following exponentiations are in $\mathbb{Z}_P^*$.
There are several possibilities for choosing P, Q and g. The most obvious way is to choose all parameters uniformly at random, subject to the given constraints. A common way to choose P and Q is to let them satisfy $P = 2Q + 1$. For some applications it may be more efficient to choose a smaller Q. However, in order to keep the following construction and proof general, we will not put any additional constraints on the choice of P, Q, and g. We will assume the variables P, Q, and g to be sampled uniformly among all possible choices.
For any P, Q, and g defined as above the DDH-Assumption in a subgroup of $\mathbb{Z}_P^*$ of order Q is defined as follows:

Assumption 7.1.1 (DDH-Assumption)
Given P, Q, g, g^a and g^b it is hard to distinguish $g^{a \cdot b}$ and g^c.

This is the average-case assumption. This assumption would not hold, if g could be any element in $\mathbb{Z}_P^*$. Then it would be possible, that g generates $\mathbb{Z}_P^*$. In that case, g^a and g^b can give away information about $g^{a \cdot b}$. For example, if g^a and g^b are quadratic residues, then so is $g^{a \cdot b}$. The reason why it does not present a problem, that g is of order Q and not necessarily of order P is, that all known algorithms for computing the discrete logarithm are subexponential in n, the length of P. That does not change, when applied to the subgroup of size Q, generated by g. Therefore, it is possible to work in a smaller subgroup, without compromising the security [20, 18, 23].
The DDH-Assumption is equivalent to its worst case version. The proof is an application of the concept of random self-reducibility and can be looked up in [20]. The concept of random self-reducibility was introduced by Blum and Micali in [5]. Informally a problem is random self-reducible if solving the problem on any instance can be efficiently reduced to solving the problem on a random instance. A problem, that is random self-reducible can either be solved with overwhelming success probability for every instance or it cannot be solved for a random instance with non-negligible success probability.
When Blum and Micali introduced the concept, they also showed, that for any specific prime P and generator g the discrete logarithm is random self-reducible. Given $\langle P, g, g^a \rangle$ for any a, it is easy to generate a random instance $\langle P, g, g^{a+r} = g^a \cdot g^r \rangle$, with $r \in_U \{1, \ldots, P-1\}$. Given the solution of the random instance, $a + r$, it is easy to compute the solution of the original instance, a.
Maurer [16] showed a similar result for the CDH-Assumption. Given $\langle P, g, g^a, g^b \rangle$ for any a and b it is easy to generate a random instance $\langle P, g, g^{a+r}, g^{b+s} \rangle$, with $r, s \in_U \{1, \ldots, P-1\}$. Given the solution $z = g^{(a+r)\cdot(b+s)}$ for the random instance, it is easy to compute the solution $g^{a \cdot b} = z \cdot (g^a)^{-s} \cdot (g^b)^{-r} \cdot g^{-s \cdot r}$.

We will now formalize the Diffie-Hellman-Assumptions:

Assumption 7.1.2 (DDH-Assumption (worst case version))
Given P, Q, g, g^a, g^b, and g^c it is hard to tell whether or not $c \equiv a \cdot b \bmod Q$.

The computational version of the Diffie-Hellman assumption is:

Assumption 7.1.3 (CDH-Assumption)
Given P, Q, g, g^a, and g^b, it is hard to compute $g^{a \cdot b}$.

The general version of the Diffie-Hellman assumption extends the CDH-Assumption to several variables a_i, $i \in \{1, \dots, m\}$ which are all uniformly sampled in $\mathbb{Z}_Q$.

Assumption 7.1.4 (GDH-Assumption)
Given P, Q, g, and g^{a_i} for all $i \in \{1, \dots, m\}$, it is hard to compute $g^{\prod_{i=1}^{m} a_i}$.

Any method of exchanging even a single bit with this protocol relies on the CDH-Assumption. The extension to the General DH-Assumption allows the exchange of more bits.
The following construction of a pseudorandom function family relies on the security of the DDH-Assumption. Therefore, we are now going to briefly discuss its security. A strong indicator for the validity of the DDH-Assumption is, that breaking the it implies an efficient algorithm for factorization [3, 27]. In [28], Shoup showed that there exists no efficient "generic" algorithm for computing the discrete logarithm or the DH-problem. A "generic" algorithm is one that does not exploit any special properties of the encoding of the group elements. It works for a "black-box-group", where the encoding of each element is random. Given the encoding of a and b, it can query the encodings of $a \cdot b$ and a^{-1}. Boneh and Venkatesan showed in [7] that computing the last $\sqrt{logP}$ bits of g^{ab} is as hard as computing g^{ab}. Several works over the last ten years are based on the Diffie-Hellman-Assumptions. All but one of them rely on the average-case version of the DDH-Assumption. Then Naor and Reingold proved in [20] that there exists a polynomial time reduction between the average case and the worst case version of the DDH-Assumption. That might strengthen our confidence in it. What gives additional evidence to the validity of the DDH-Assumption is the fact that the related CDH-Assumption has been extensively studied in the last ten years. The CDH-Assumption was examined in several groups in [7, 16, 17, 27]. Additional results can be looked up in [7, 9, 28]. It is obvious, that the CDH-problem is at least as hard as the DDH-problem. Maurer [16] and Boneh and Lipton [6] proved that in several settings those problems are in fact equivalent.

7.2 Pseudorandom Functions from the DH-Assumptions

7.2.1 The Generator

In Chapter 5 we have shown that pseudorandom generators with output length polynomial in the input length exist. Let us now look at the following length doubling pseudorandom generator:

$$\overline{G}_{P,g,g^a}(b) := \langle g^b, g^{a\cdot b}\rangle = \langle \overline{G}^0(b), \overline{G}^1(b)\rangle$$

This is the pseudorandom generator that is implied by the CDH-Assumption. But remember the general construction of pseudorandom functions in Chapter 6:

$$f_y(x) = g_{x_{l(n)}}(\dots(g_{x_1}(y))\cdots)$$

As the output of G_{P,g,g^a} is a pair of values in the subgroup of $\mathbb{Z}_P^*$ generated by g and not in $\mathbb{Z}_P^*$, a straightforward application of the general construction does not work here.
Let us look at a slightly modified generator instead:

$$G_{P,g,g^a}(g^b) := \langle g^b, g^{a\cdot b}\rangle = \langle G^0_{P,g,g^a}(g^b), G^1_{P,g,g^a}(g^b)\rangle$$

This does not seem to make much sense at first sight. The first part of the output is the input. And even if we only look at the second part of the output, we have a problem. The second output part $g^{a\cdot b}$ is only efficiently computable if the DDH-Assumption is easy to break. This implies, that if G_{P,g,g^a} is efficiently computable it is not pseudorandom anymore. Therefore G_{P,g,g^a} is not a pseudorandom generator. But G_{P,g,g^a} has two useful properties. The first is, that it is efficiently computable on any input if the random bits that are necessary to compute it are available. The second property is concerning the probability distribution generated by G_{P,g,g^a}. This probability distribution can be easily calculated if the input g^b is chosen uniformly.
Let us apply the general construction of pseudorandom functions on this generator. For all tuples (P, g, a) that would result in a function f with input $x \in \{0,1\}^n$:

$$f(x) = G^{x_n}_{P,g,g^a}(\dots G^{x_1}_{P,g,g^a}(g^b)\dots)$$

With each call of the generator G_{P,g,g^a} we take the input of the generator to the power of a or 1, depending on the input x of the function. Let m be the number of ones in x. Then we would get $f(x) = g^{b \cdot a^m}$. And that is obviously not very secure. Therefore we should not always use the same a, but instead a vector $\vec{a} = (a_0 \ldots a_n) \in \mathbb{Z}_Q^{n+1}$. Then there is a different a for each call of G_{P,g,g^a}. Therefore we now have tuples $\langle P, g, \vec{a} \rangle$. For each tuple we get a different function $f_{P,g,\vec{a}}$. For all possible inputs $x \in \{0,1\}^n$ the function $f_{P,g,\vec{a}}$ is defined to be:

$$f_{P,g,\vec{a}}(x) := G^{x_n}_{P,g,g^{a_n}}(\ldots(G^{x_1}_{P,g,g^{a_1}}(g^{a_0}))\ldots) = (g^{a_0})^{\prod_{x_i=1} a_i}$$

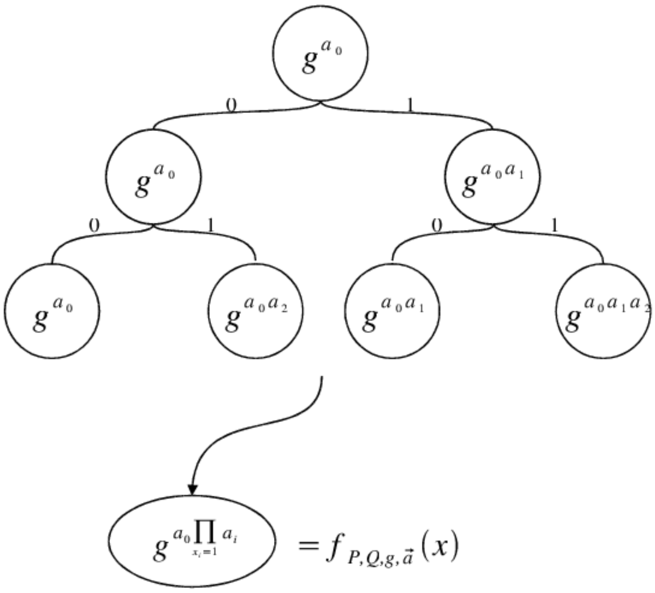


Abbildung 7.1: NR-Construction

This figure 7.1 is analogous to figure 6.1 on page 38 in the general construction. It represents the construction of the random functions $f_{P,Q,g,\vec{a}}$. For each $x \in \{0,1\}^n$, the value $f_{P,Q,g,\vec{a}}(x)$ is stored in the leaf reachable from the root following the edge-path labeled with x.

7.2.2 The NR-Construction

We will now define an efficiently computable function family $F = \{F_n\}_{n\in\mathbb{N}}$ by using the above construction for its functions.
Let P be an n-bit prime number and let Q be a prime divisor of $P-1$ and et g be of order Q in $\mathbb{Z}_P^*$. Let $\vec{a} := \langle a_0, \ldots, a_n \rangle$ be a sequence of $n+1$ elements in $\mathbb{Z}_Q$.
Then for all $n \in \mathbb{N}$ each function f_{key} in F_n has a key $key := \langle P, Q, g, \vec{a} \rangle$.
For all possible inputs $x \in \{0,1\}^n$ define the function $f_{P,Q,g,\vec{a}}$ to be:

$$f_{P,Q,g,\vec{a}}(x) := (g^{a_0})^{\prod\limits_{x_i=1} a_i}$$

Theorem 7.2.1
Then $F = \{F_n\}_{n\in\mathbb{N}}$ is an efficiently computable pseudorandom function family.

The construction of this pseudorandom function family is very similar to the general construction. But here, we will use a modified pseudorandom generator to construct the pseudorandom functions. In fact, the generator we use is not a pseudorandom generator anymore. But we will prove, that the above given function family is still a pseudorandom function family.
If we look back at the definition of pseudorandom function families 2.2.9 on page 11 we find, that this function family does not exactly match that definition. Here we have that a function $f_{P,Q,g,\vec{a}}$ in F_n has domain $\{0,1\}^n$ and range $\langle g \rangle$. Therefore different functions in F_n have different ranges. But this is a deviation from the definition does not present a problem. For many applications, e.g. private-key authentication, private-key identification and digital signatures, this derivation from the definition of pseudorandom function families does not matter. Furthermore it is easy to construct a family of hash functions such that for each pseudorandom function there is a hash function. By combining each pseudorandom function with a hash function by always taking the hash-value of the output of the pseudorandom function a new pseudorandom function family arises. This new function family is as secure as the above constructed, as can be seen in [20]. We will not give the proof here. Instead we now turn to the proof of Theorem 7.2.1

Proof

We want to prove, that F is an *efficiently computable pseudorandom function* family. Recall, that an efficiently computable pseudorandom function family has the three properties:

1. Indexing
2. Polynomial Time Evaluation
3. Pseudorandomness

The first two properties are easy to prove.

1. Indexing:
   It is clear, that F has the indexing-property. Each function in F_n has a unique index $key = \langle P, Q, g, \vec{a} \rangle$ associated with it. The vector $\vec{a}$ can be chosen randomly in $\mathbb{Z}_Q^{n+1}$. P, Q and g can be sampled randomly such that they satisfy the conditions of the DDH-Assumption. Thus, picking a function f_{key} randomly $\in F_n$ is easy, if such a tuple of random values is available.
2. Polynomial Time Evaluation:
   The given construction of the functions in F can easily be transformed into a polynomial time construction-algorithm. Therefore for all possible keys $\langle P, Q, g, \vec{a} \rangle$ and inputs $x \in \{0,1\}^n$ the value $f_{P,Q,g,\vec{a}}(x)$ can be computed in polynomial time.

It remains to prove the third property of an efficiently computable pseudorandom function family, pseudorandomness. Here, we have very much the situation we had in the proof of the general construction of pseudorandom functions. But the major difference is, that in this case we are working with a generator that is not an efficiently computable pseudorandom generator, as we have discussed previously. Because we have to take that into account, it is not enough to apply the hybrid argument once. To prove that F really is pseudorandom, we need to apply the hybrid argument twice. The basic proof-principle is a familiar one. We assume, that the functions $f_{P,Q,g,\vec{a}}$ are not pseudorandom. Then we will lead that assumption to a contradiction.
Assume that the functions $f_{P,Q,g,\vec{a}}$ are not pseudorandom. Then there exists a distinguisher, that distinguishes them from random functions. Let $t(n)$ be the running time of such a distinguisher.
The core of the proof are hybrid distributions $I_i^{n,t(n)}$, $1 \leq i \leq t(n)$. For all $1 \leq i \leq t(n)$, let b_i and c_i be sampled randomly in $\mathbb{Z}_Q$. Then the hybrid distributions are defined as follows:

$$I_i^{n,t(n)} \;\; := \;\; \langle P, Q, g, g^a, (g^{b_1}, g^{a \cdot b_1}), \ldots, (g^{b_i}, g^{a \cdot b_i}), (g^{b_{i+1}}, g^{c_{i+1}}), \ldots, (g^{b_{t(n)}}, g^{c_{t(n)}}) \rangle$$

For $i = 1$ we get:

$$I_1^{n,t(n)} = \langle P, Q, g, g^a, (g^{b_1}, g^{c_1}), \ldots, (g^{b_{t(n)}}, g^{c_{t(n)}}) \rangle$$

For $i = t(n)$ we get:

$$I_{t(n)}^{n,t(n)} = \langle P, Q, g, g^a, (g^{b_1}, g^{a \cdot b_1}), \ldots, (g^{b_{t(n)}}, g^{a \cdot b_{t(n)}}) \rangle$$

The proof works in two steps. In the first step, we prove that $I_1^{n,t(n)}$ and $I_{t(n)}^{n,t(n)}$ are polynomially indistinguishable. We will show, that if there exists a distinguisher for the two extreme hybrid distributions $I_1^{n,t(n)}$ and $I_{t(n)}^{n,t(n)}$, then the DDH-Assumption does not hold. As that is a contradiction, there exists no such distinguisher. Therefore the two extreme hybrid distributions $I_1^{n,t(n)}$ and $I_{t(n)}^{n,t(n)}$ are polynomially indistinguishable.
In the second step, we show that if there exists a distinguisher for F and the set of all functions with the same domain and range, then there exists a distinguisher for the two extreme hybrid distributions $I_1^{n,t(n)}$ and $I_{t(n)}^{n,t(n)}$. But we know from the first step, that $I_1^{n,t(n)}$ and $I_{t(n)}^{n,t(n)}$ are polynomially indistinguishable, there exists no distinguisher for those distributions. And this contradiction implies, that there exists no distinguisher for F and the set of all functions with the same domain and range.
That completes the proof that F is an efficiently computable pseudorandom function family.

First Step:

Theorem 7.2.2
If the DDH-Assumption holds, then $I_1^{n,t(n)}$ and $I_{t(n)}^{n,t(n)}$ are polynomially indistinguishable.

Proof:

We want to show that $I_1^{n,t(n)}$ and $I_{t(n)}^{n,t(n)}$ are polynomially indistinguishable. Therefore, we want to show that for all probabilistic polynomial time algorithms D and for all $k \in \mathbb{N}$:

$$|\text{prob } [D(I_1^{n,t(n)}) = 1] - \text{prob } [D(I_{t(n)}^{n,t(n)}) = 1]| < \frac{1}{n^k}$$

As a prerequisite, we have the DDH-Assumption. The DDH-Assumption is, that for all P, Q, g, a, b, and c defined as above, there is no distinguisher for $g^{a \cdot b}$ and g^c, i.e., for all probabilistic polynomial time algorithms D', for all $k' \in \mathbb{N}$:

$$|\text{prob } [D'(P, Q, g, g^a, g^b, g^{a \cdot b}) = 1] - \text{prob } [D'(P, Q, g, g^a, g^b, g^c) = 1]| < \frac{1}{n^{k'}}$$

If we look back to Chapter 3, we can see that this is a straightforward application of the Hybrid Theorem 3.0.1 on page 13.
The probability distribution A^n in the Hybrid Theorem corresponds to the probability distribution that generates the $t(n)$ many $\langle g^{b_i}, g^{c_i}\rangle$-tuples here. Analogously B^n corresponds to the probability distribution that generates the $t(n)$ many $\langle g^{b_i}, g^{a \cdot b_i}\rangle$-tuples here. Then A is the probability distribution that generates values $g^{a \cdot b}$, where a and b are uniform in $\mathbb{Z}_Q$ and B is the probability distribution that generates values g^c, where c is uniform in $\mathbb{Z}_Q$. More formally we get the following correspondences between the Hybrid Theorem and the current situation:

$$\begin{array}{rcl} S_A & \cong & I_1^{n,t(n)} = \langle P, Q, g, g^a, (g^{b_1}, g^{c_1}), \ldots, (g^{b_{t(n)}}, g^{c_{t(n)}})\rangle, \\ S_B & \cong & I_{t(n)}^{n,t(n)} = \langle P, Q, g, g^a, (g^{b_1}, g^{a \cdot b_1}), \ldots, (g^{b_{t(n)}}, g^{a \cdot b_{t(n)}})\rangle, \\ a & \cong & \langle P, Q, g, g^a, g^b, g^c\rangle \qquad\qquad \text{and} \\ b & \cong & \langle P, Q, g, g^a, g^b, g^{a \cdot b}\rangle \end{array}$$

The DDH-Assumption states, that g^c and $g^{a \cdot b}$ are indistinguishable and therefore implies that $\langle P, Q, g, g^a, g^b, g^c\rangle$ and $\langle P, Q, g, g^a, g^b, g^{a \cdot b}\rangle$ are indistinguishable. That is the prerequisite we need for applying the Hybrid Theorem. That gives us the desired result. Now, we have that $S_A = I_1^{n,t(n)}$ and $S_B = I_{t(n)}^{n,t(n)}$ are polynomially indistinguishable. The proof of 7.2.2 is complete. We have shown, that $I_1^{n,t(n)}$ and $I_{t(n)}^{n,t(n)}$ are polynomially indistinguishable.

□

Second Step:

Theorem 7.2.3
If $I_1^{n,t(n)}$ and $I_{t(n)}^{n,t(n)}$ are polynomially indistinguishable, then F, defined as in the construction theorem, is an efficiently computable pseudorandom function family.

Proof:

We know, that $I_1^{n,t(n)}$ and $I_{t(n)}^{n,t(n)}$ are polynomially indistinguishable. Therefore, we know that for all probabilistic polynomial time algorithms D' and for all $k' \in \mathbb{N}$:

$$\text{prob}\ [D'(I_1^{n,t(n)}) = 1] - \text{prob}\ [D'(I_{t(n)}^{n,t(n)}) = 1] < \frac{1}{n^{k'}}$$

We want to show that F is pseudorandom. To do that, we need to show that there is no distinguisher D for a function $f_{P,Q,g,\vec{a}}$ sampled from F and a function f

sampled uniformly from all functions with the same domain and range. Therefore, we need to show that for all probabilistic polynomial time algorithms D and for all $k \in \mathbb{N}$:

$$\text{prob } [D(f_{P,Q,g,\vec{a}}) = 1] - \text{prob } [D(f) = 1] < \frac{1}{n^k}$$

This is similar to the situation we had in the general construction of pseudorandom functions. And again the proof works by contradiction. We assume that there exists a distinguisher D for $f_{P,Q,g,\vec{a}}$ and f with success probability $\delta_D(n) \geq \frac{1}{n^k}$ and running time $t(n)$. Then we will construct a distinguisher D' for $I_1^{n,t(n)}$ and $I_{t(n)}^{n,t(n)}$ with success probability $\delta_{D'}(n) \geq \frac{1}{n^{k'}}$.

Construction of D':

The distinguisher D' gets a string $I = \langle P, Q, g, g^a, (g^{b_1}, g^{\hat{c}_1}), \cdots, (g^{b_{t(n)}}, g^{\hat{c}_{t(n)}})\rangle$ as input, where each $\hat{c}_i$ is an element of $\mathbb{Z}_Q$ and either equal to $a \cdot b_i \bmod Q$ or sampled uniformly in $\mathbb{Z}_Q$. Then D' constructs a hybrid function h_J and runs D with input h_J. When D decides h_J to be pseudorandom, then D' decides I to be pseudorandom. When D decides h_J to be random, then D' decides I to be random. The hybrid function h_J is constructed similar to $f_{P,Q,g,\vec{a}}$. In the construction of $f_{P,Q,g,\vec{a}}(x)$ there are as many exponentiations as there are 1's in x. When constructing h_J, D' picks a random value $J \in \{1, \ldots, n\}$. This value J determines the number of exponentiations in dependence on x in the construction of $h_J(x)$. Informally spoken, J is the point where h_J starts to look at $x = (x_1 \ldots x_n)$. The bits x_1 to x_{J-1} are not being used in the construction. The point J is, where D' integrates its input I into h_J. And from $J+1$ onward, h_J is constructed like $f_{P,Q,g,\vec{a}}$. When D' runs $D(h_J)$, D makes a polynomial number of queries on h_J. Let $x^1, \ldots, x^{m(n)}$ be those queries where $x^i = (x_1^i \ldots x_n^i) \in \{0,1\}^n$.

Distinguisher D':

Input : $I = \langle P, Q, g, g^a, (g^{b_1}, g^{\hat{c}_1}), \cdots, (g^{b_{t(n)}}, g^{\hat{c}_{t(n)}})\rangle$
Output : $D'(I) \in \{0, 1\}$
Output 0 means, that D' decides I to be random.
Output 1 means, that D' decides I to be pseudorandom.

1. Pick a random value $J \in_U \{1, \ldots, n\}$
2. Pick $J - n$ random values $\langle a_{J+1}, \ldots a_n\rangle$ in $\mathbb{Z}_Q$

3. For $i = 1 \ldots m(n)$ define $l(i) := \min\limits_{1 \leq j \leq i} \{j | (x_1^j \ldots x_{J-1}^j) = (x_1^i \ldots x_{J-1}^i)\}$ and calculate $h_J(x^i)$ in the following way:

$$
h_J(x^i) := \begin{cases} \left(g^{\hat{c}_{l(i)}}\right)^{\prod_{\substack{x_k^i=1\\k>J}} a_k} & \text{if } x_J^i = 1 \\[2em] \left(g^{b_{l(i)}}\right)^{\prod_{\substack{x_k^i=1\\k>J}} a_k} & \text{if } x_J^i = 0 \end{cases}
$$

4. $D'(I) := D(h)$

The introduction of the index $l(i)$ in the third step of the algorithm is necessary to make sure that the function h_J is defined consistently. For each input of h_J, the distinguisher D' first checks whether this input or a prefix of this input has appeared before and if so takes that into account.
As $t(n)$ is the running time of D, there are at most $t(n)$ many queries on h_J. Therefore, $m(n) \leq t(n)$ and therefore h_J is well defined.
Now let us look at the hybrid functions $h_1, \ldots, h_J, \ldots, h_n$ that are determined by the choice of J and therefore are the hybrid distributions we need for the hybrid argument. The picture we get is somewhat different from the one we got in the general construction of pseudorandom function families on page 43.

$$
\begin{array}{rcl}
f_{P,Q,g,\vec{a}} \cong h_1 & : & \overbrace{\boxed{\text{1.query}}}^{n}_{\substack{\uparrow J=1\\(b_1,\hat{c}_1)}} \quad \ldots \quad \overbrace{\boxed{\text{m(n).query}}}^{n}_{\substack{\uparrow J=1\\(b_{m'(n)},\hat{c}_{m'(n)})}} \\
\vdots & & \\
h_J & : & \overbrace{\boxed{\text{1.query}}}^{n}_{\substack{\uparrow J\\(b_1,\hat{c}_1)}} \quad \ldots \quad \overbrace{\boxed{\text{m(n).query}}}^{n}_{\substack{\uparrow J\\(b_{m'(n)},\hat{c}_{m'(n)})}} \\
\vdots & & \\
f \cong h_n & : & \overbrace{\boxed{\text{1.query}}}^{n}_{\substack{J=n \uparrow\\(b_1,\hat{c}_1)}} \quad \ldots \quad \overbrace{\boxed{\text{m(n).query}}}^{n}_{\substack{J=n \uparrow\\(b_{m'(n)},\hat{c}_{m'(n)})}}
\end{array}
$$

As a query may appear more often than once, it is possible, that it is not the mth pair $(b, \hat{c})$ in I that is integrated into the $m(n)$th query, but a pair $(b_{m'(n)}, \hat{c}_{m'(n)})$ with $m'(n) \leq m(n)$.
Like in the general construction, the choice of J determines, at what point h_J starts to behave pseudorandom. But this time, not the whole input of D' is integrated into h_J at one point. Instead, for each query there is another pair $(b_i, \hat{c}_i)$ that is

integrated into h_J. That way, the input I of D' is integrated into h_J, not just in one point, but a part of I during each query. Now either $\hat{c}_i$ is uniform in $\mathbb{Z}_Q$ for all i, or it is pseudorandom for all i, i.e., equal to $b_i \cdot a$ for all i. If all $\hat{c}_i$ are pseudorandom and if the choice of D' is $J = 1$, then the hybrid function h_1 is a pseudorandom function. If all $\hat{c}_i$ are random and if the choice of D' is $J = n$, then the hybrid function h_n is a random function. Expressed in probabilities, that results in:

$$\begin{aligned}
\text{prob}\ [D'(I_t^{n,t}) = 1 | J = 1] &= \text{prob}\ [D(h_1) = 1] &= \text{prob}\ [D(f_{P,Q,g,\vec{a}}) = 1] \\
\text{prob}\ [D'(I_1^{n,t}) = 1 | J = n] &= \text{prob}\ [D(h_n) = 1] &= \text{prob}\ [D(f) = 1]
\end{aligned}$$

If all $\hat{c}_i$ are random, then all functions h_J start to behave pseudorandomly at the point $J + 1$. If all $\hat{c}_i$ are pseudorandom, then all functions h_J start to behave pseudorandomly at the point J.

$$\text{prob}\ [D'(I_1^{n,t}) = 1 | J = j] = \text{prob}\ [D'(I_t^{n,t}) = 1 | J = j + 1]$$

If $x_J^i = 0$, it does not matter, whether the $\hat{c}_i$ are random or pseudorandom, because then the corresponding a_i are left out in the computations anyway. So the overall effect is, that if the $\hat{c}_i$ are pseudorandom, then h behaves more pseudorandomly and if the $\hat{c}_i$ are random in $\mathbb{Z}_Q$, then h behaves more randomly.
These are the same dependencies we used in the proof of the Hybrid Theorem in Chapter 3. And as the prerequisites given here are analogous to the ones of the Hybrid Theorem, this is just another application of its arguments. The probability distribution A in the Hybrid Theorem here corresponds to the probability distribution induced by the strings $I = \langle P, Q, g, g^a, g^{b_1}, g^{\hat{c}_1}, \cdots, g^{b_t}, g^{\hat{c}_t} \rangle$ where all the $\hat{c}_i$ are sampled randomly. Analogously the probability distribution B in the Hybrid Theorem here corresponds to the probability distribution induced by the strings $I = \langle P, Q, g, g^a, g^{b_1}, g^{\hat{c}_1}, \cdots, g^{b_t}, g^{\hat{c}_t} \rangle$ where all the $\hat{c}_i$ equal $a \cdot b_i$.
The probability distribution B^n in the Hybrid Theorem here corresponds to the probability distribution induced by the above constructed function family F. The probability distribution A^n in the Hybrid Theorem here corresponds to the uniform distribution on the set of all functions with the same domain and range as the functions in F.

$$\begin{aligned}
S_A &\cong f \\
S_B &\cong f_{P,Q,g,\vec{a}} \\
a &\cong I_1^{n,t(n)} = \langle P, Q, g, g^a, (g^{b_1}, g^{\hat{c}_1}), \ldots, (g^{b_{t(n)}}, g^{\hat{c}_{t(n)}}) \rangle \qquad \text{and} \\
b &\cong I_{t(n)}^{n,t(n)} = \langle P, Q, g, g^a, (g^{b_1}, g^{a \cdot b_1}), \ldots, (g^{b_{t(n)}}, g^{a \cdot b_{t(n)}}) \rangle
\end{aligned}$$

Hence the success probability of D' is:

$$\begin{aligned}
\delta_{D'}(n) &= \text{prob}\ [D'(I_t^{n,t}) = 1] - \text{prob}\ [D'(I_1^{n,t}) = 1] \\
&= \frac{1}{n} \cdot \Big(\sum_{j=1}^{n} \text{prob}\ [D'(I_t^{n,t}) = 1 | J = j] - \sum_{j=1}^{n} \text{prob}\ [D'(I_1^{n,t}) = 1 | J = j] \Big) \\
&= \frac{1}{n} \cdot \big(\text{prob}\ [D'(I_t^{n,t}) = 1 | J = 1] - \text{prob}\ [D'(I_1^{n,t}) = 1 | J = n] \big) \\
&= \frac{1}{n} \cdot \big(\text{prob}\ [D(f_{P,Q,g,\vec{a}}) = 1] - \text{prob}\ [D(f) = 1] \big) \\
&> \frac{1}{n^{k+1}} = \frac{1}{n^{k'}}
\end{aligned}$$

Hence

$$\delta_{D'}(n) \quad > \quad \frac{1}{n^{k'}}$$

Therefore, F is an efficiently computable pseudorandom function family if and only if the DDH-Assumption holds.

□

7.2.3 Efficiency of the NR-Functions

Complexity Classes

To evaluate the efficiency of the construction of pseudorandom functions in Theorem 7.2.1 on page 50, we are first going to define some complexity classes:

NC^k = { f | f is computable by a $(\log n)^k$ depth treshold circuit of polynomially bounded size.}

TC^0 = { f | f is computable by a constant depth treshold circuit of polynomially bounded size.}

TC^0_k = { f | f is computable by a depth k treshold circuit of polynomially bounded size.}

Efficiency

Consider a function $f_{P,Q,g,\vec{a}}$ defined as in the Construction Theorem 7.2.1 on page 50. To evaluate this function at any given point $x \in \{0,1\}^n$ it is necessary to compute $f_{P,Q,g,\vec{a}}(x) = (g^{a_0})^{\prod\limits_{x_i=1} a_i}$. This term consists of one multiple product $y = a_0 \cdot \prod\limits_{x_i=1} a_i$ modulo Q and one modular exponentiation g^y.

To evaluate a function from the general construction at any given point, it is necessary to evaluate the pseudorandom generator from which the function is constructed once for each output bit. Therefore this pseudorandom functions are more efficiently computable then the generally constructed ones.

Furthermore it is possible to improve the efficiency in computing $f_{P,Q,g,\vec{a}}$ by preprocessing. The most obvious values to preprocess are, the values g^{2^i} for all $i \in \mathbb{N}$ up to the length of Q. If we have those values, computing $f_{P,Q,g,\vec{a}}(x)$ now consists of two multiple products modulo prime numbers. And in the case that Q is much smaller than P the first multiple product is much cheaper than the second one. In [8] Brickell et.al. considered further methods of preprocessing that improve the efficiency of calculating $f_{P,Q,g,\vec{a}}(x)$ by a factor of $O(\log(n))$. Naor and Reingold [20] also considered the parallel complexity of these pseudorandom functions. They used the results of Beame, Cook and Hoover [2] who showed that division and related operations are computable in NC^1 and the results of Reif and Tate [25, 26] who showed that these operations are also computable in TC^0. For us these results are interesting because they include multiple products. Details can be looked up in [29, 30]. There it was shown, that multiple sums are computable in TC^0, multiplications and divisions in TC^0_3 and multiple products in TC^0_4.

Now we have, that after preprocessing the g^{2^i}, it is possible to evaluate the pseudorandom function $f_{P,Q,g,\vec{a}}$ in TC^0. With further preprocessing instead of a straightforward application of [29, 30] the function $f_{P,Q,g,\vec{a}}$ can even be evaluated in TC^0_5. This further preprocessing is possible because of the special features of the function $f_{P,Q,g,\vec{a}}(x) = (g^{a_0})^{\prod\limits_{x_i=1} a_i}$. In both multiple products, we can assume any preprocessing of the values in the multiplications. That is so, because these values are all taken either from the sequence $\vec{a} =< a_0, \ldots, a_n >$ or from the g^{2^i}. Furthermore we do not need the actual value of the first product $y = a_0 \cdot \prod\limits_{x_i=1} a_i$.

And finally, the value P is known in advance as well.

Altogether that leads us to:

Remark 7.2.1
Let $F = \{F_n\}_{n\in\mathbb{N}}$ be as in Theorem 7.2.1 on page 50.
Then there exists a polynomial p such that for all functions $f_{P,Q,g,\vec{a}}$ in F there exists a depth 5 threshold circuit of size bounded by $p(n)$ that computes $f_{P,Q,g,\vec{a}}$.

8 Special Construction of Naor, Reingold and Rosen

In this chapter, we are going to describe and examine another construction of a pseudorandom function family. This construction and therefore the security of the resulting pseudorandom function family is based on the assumption that factoring Blum-Integers is hard.

8.1 Prerequisites

We start with giving a definition of Blum-Integers.

Definition 8.1.1 (Blum-Integers)
Let P and Q be two distinct primes both congruent to $3 \bmod 4$. Then $N = P \cdot Q$ is a Blum-Integer.

The reason why we are going to use Blum-Integers instead of normal integers is the fact that Blum, Blum and Shub proved in [4], that the squaring permutation restricted to Blum-Integers is a permutation when the domain is restricted to the subgroup of quadratic residues in $\mathbb{Z}_N^*$.
We will denote the subgroup of quadratic residues in $\mathbb{Z}_N^*$ by Q_N. It consists of all the elements $x \in \mathbb{Z}_N^*$ that have a square root $y \in \mathbb{Z}_N^*$ such that $y^2 = x \bmod N$. Every $x^2 \in Q_N$ has four distinct square roots $+x, -x, +y$ and $-y$ in $\mathbb{Z}_N^*$. Blum, Blum and Shub proved in [4] that in the case of N being a Blum-Integer, exactly one of those square roots is in Q_N again. Therefore squaring modulo a Blum-Integer is indeed a permutation.
The natural way to sample Blum-Integers is to sample them uniformly among all integers with the above constraints. It is possible to sample them under additional security constraints. But in order to keep the following construction and proof general, we assume Blum-Integers to be sampled uniformly among all Blum-Integers.
The following construction relies on the assumption that factoring these Blum-integers is hard. We say that factoring Blum-Integers is hard when there exists no probabilistic polynomial time algorithm A that can calculate P or Q with

non-negligible success probability. More formally we get:

Definition 8.1.2 (Factoring Blum-Integers)
Let N be an n-bit Blum-Integer.
If there is no probabilistic polynomial time algorithm A and no $k \in \mathbb{N}$ such that for all $n \in \mathbb{N}$ big enough

$$prob\ [A(N) \in \{P, Q\}] > \frac{1}{n^k}$$

Then factoring Blum-integers is hard.

8.2 Pseudorandom Functions from the BBS-Generator

8.2.1 The Generator

As in the previous constructions we are going to use a pseudorandom generator as a building block for a pseudorandom function family. The general construction needs a length doubling pseudorandom generator. This time we are going to use a modification of the Blum-Blum-Shub-Generator [4]. This generator is constructed exactly as it is proposed in the proof of the existence of pseudorandom generators in Chapter 5. The construction-scheme starts with the Blum-Micali-Paradigm:

Definition 8.2.1 (Blum-Micali-Paradigm)
Let $l(n)$ be an integer function. Let $x \in \{0,1\}^n$ be an n-bit seed. Let $f : \{0,1\}^n \to \{0,1\}^n$ be a one-way permutation and let $B : \{0,1\}^n \to \{0,1\}$ be a hard core predicate.
Then x can be stretched to a pseudorandom $l(n)$-bit string by using the following scheme:

$$B(x), B(f(x)), \ldots, B(f^{(k)}(x)), \ldots, B(f^{(l(n)-1)}(x))$$

The Blum-Blum-Shub-Generator is obtained by applying the squaring permutation modulo a Blum-Integer as the one-way-function and using the least significant bit as hard core predicate in the Blum-Micali-Paradigm. Another hard core predicate that can be applied to this scheme is the Goldreich-Levin hard core predicate [13].

Definition 8.2.2 (Goldreich-Levin Hard Core Predicate)
Let m and r be two bit strings of the same length.
Then the Goldreich-Levin hard core predicate $B_r(m)$ denotes the inner product $\langle m, r\rangle \bmod 2$.

The common property, that those two hard core predicates have, is that the Goldreich-Levin-Theorem can be applied to them to recover their respective arguments. The reconstruction algorithm of the Goldreich-Levin-Theorem fixes the argument of the hard core predicate and changes its parameter. That is possible with both predicates and therefore both hard core predicates can be used in the following construction. However, for the sake of simplicity, we will use the Goldreich-Levin hard core predicate instead of the least significant bit. Now we are going to apply the squaring permutation modulo a Blum-Integer and the Goldreich-Levin hard core predicate to the Blum-Micali-Paradigm. Then we get the pseudorandom generator we are going to use in the following construction of a pseudorandom function family.

Definition 8.2.3 (modified BBS-Generator)
Let $l(n)$ be an integer function. Let $B_r : \{0,1\}^n \to \{0,1\}$ be the Goldreich-Levin hard core predicate. Let $N = P \cdot Q$ be a Blum-Integer. Let $x \in \{0,1\}^n$ be an n-bit seed. Let all exponentiations be modulo N.
Then the modified BBS-Generator G stretches x to the pseudorandom $l(n)$-bit string

$$G(x) := B_r(x), B_r(x^2), \ldots, B_r(x^{2^k}), \ldots, B_r(x^{2^{l(n)-1}})$$

Now we have the pseudorandom generator we are going to use. What we still need is a function on which to apply this generator. Let g be a quadratic residue in $\mathbb{Z}_N^*$. Let $\vec{a} = \langle a_{1,0}, a_{1,1}, a_{2,0}, a_{2,1}, \ldots a_{n,0}, a_{n,1}\rangle$ be a sequence of $2n$ elements in $\{0, 1, \ldots, n-1\}$. Now let us look at the function:

$$h_{N,\vec{a},g}(x) := g^{\prod_{i=1}^{n} a_{i,x_i}}$$

The following figure 8.1 represents the construction of the unpredictable function. For any $x \in \{0,1\}^n$, the value $h_{N,\vec{a},g}(x)$ is stored in the leaf reachable from the root following the edge-path labeled with x.
This function is not pseudorandom itself, but it is unpredictable in some weak sense. Assuming that factoring Blum-Integers is hard, Naor and Reingold showed that $h_{N,\vec{a},g}$ is unpredictable against an adaptive sample and a random challenge [20, 21]. In other words, there is no probabilistic polynomial time algorithm that for a random challenge $x \in \{0,1\}^n$ can compute $h_{N,\vec{a},g}(x)$ after adaptively querying polynomially many values $h_{N,\vec{a},g}(y)$ with $y \neq x$.
Now the idea of the following construction is, to use the function $h_{N,\vec{a},g}$ as a seed to

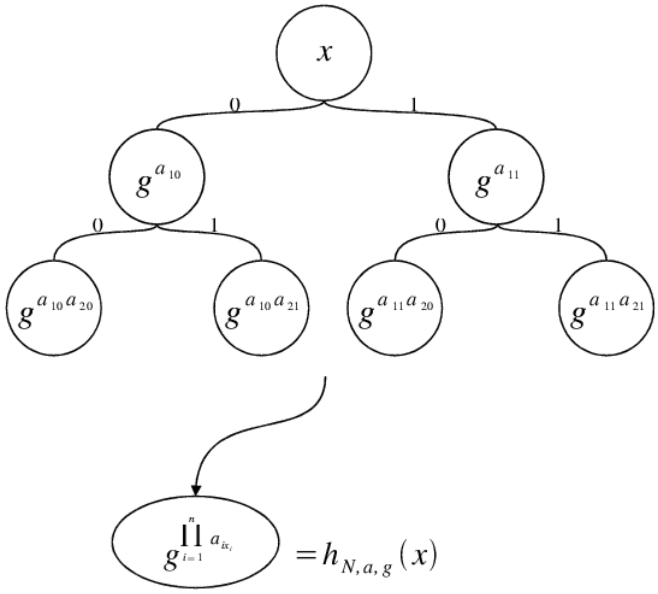


Abbildung 8.1: The Unpredictable Function

the modified BBS-Generator. What we get is the following function:

$$G(h(x)) = G_{N,r,l}(g^{\prod_{i=1}^{n} a_{i,x_i}}) = B_r(g^{\prod_{i=1}^{n} a_{i,x_i}}), \ldots, B_r(g^{2^k \prod_{i=1}^{n} a_{i,x_i}}), \ldots, B_r(g^{2^{l(n)-1} \prod_{i=1}^{n} a_{i,x_i}})$$

And this is a pseudorandom function as we will prove in the following. It is not obvious, why the construction we used results in a pseudorandom function. And it does not work in general. The reason, why this construction results in a pseudorandom function are the specific number theoretic features, which the function $h_{N,\vec{a},g}$ and the modified BBS-Generator have in common. But taking an unpredictable function as input of a pseudorandom generator does not generally result in a pseudorandom function. In fact, it is easy to find a counterexample that does not result in a pseudorandom function.

Counterexample:

1. Let $h : \{0,1\}^n \to \{0,1\}^{\frac{n}{2}}$ be an unpredictable function.
   For all $y_1, y_2 \in \{0,1\}^{\frac{n}{2}}$ define $h' : \{0,1\}^n \to \{0,1\}^n$ as $h'(y_1y_2) = \langle h(y_1), y_2 \rangle$.
   Clearly h' is unpredictable.

2. Let $G : \{0,1\}^{\frac{n}{2}} \to \{0,1\}^{l(n)}$ be a pseudorandom generator.
   For all $y_1, y_2 \in \{0,1\}^{\frac{n}{2}}$ define $G' : \{0,1\}^n \to \{0,1\}^{l(n)}$ as $G'(y_1y_2) := G(y_2)$.
   Clearly G' is a pseudorandom generator.

3. Define the function $f : \{0,1\}^n \rightarrow \{0,1\}^{l(n)}$ by applying G' on h':
   $f(y_1 y_2) := G'(h'(y_1 y_2)) = G'(h(y_1)y_2) = G(y_2)$

Then f is clearly not a pseudorandom function. Its output is independent from the unpredictable function. In fact, $f(y_1 y_2)$ is a constant function for all $y_1 \in \{0,1\}^{\frac{n}{2}}$. Therefore, the proof for our construction of a pseudorandom function family cannot hold in general but must use the specific number-theoretic features that the function $g^{\prod_{i=1}^{n} a_{i,x_i}}$ and the modified Blum-Blum-Shub-Generator have in common.

8.2.2 The NRR-Construction

We are now going to define a pseudorandom function family $F = \{F_n\}_{n \in \mathbb{N}}$ by using the modified Blum-Blum-Shub pseudorandom generator G defined above and the unpredictable function $h_{N,\vec{a},g}(x) = g^{\prod_{i=1}^{n} a_{i,x_i}}$.

Let P and Q be two n-bit primes such that $N = P \cdot Q$ is a Blum-Integer, let g be a quadratic residue in $\mathbb{Z}_N^*$, let $\vec{a} = \langle a_{1,0}, a_{1,1}, a_{2,0}, a_{2,1}, \ldots a_{n,0}, a_{n,1} \rangle$ be a sequence of $2n$ elements in $\{0, 1, \ldots, N-1\}$ and let r be an n-bit string.
Let B_r denote the inner product modulo 2, the Goldreich-Levin hard core predicate.
Then for all $n \in \mathbb{N}$, each function f_{key} in F_n has a key $key := \langle N, \vec{a}, g, r \rangle$.
For all possible inputs $x \in \{0,1\}^n$ and for all integer functions $l = l(n)$ define the function $f_{N,\vec{a},g,r} : \{0,1\}^n \rightarrow \{0,1\}^{l(n)}$ to be:

$$f_{N,\vec{a},g,r} := B_r(g^{\prod_{i=1}^{n} a_{i,x_i}}), B_r(g^{2\prod_{i=1}^{n} a_{i,x_i}}), \ldots, B_r(g^{2^k \prod_{i=1}^{n} a_{i,x_i}}), \ldots, B_r(g^{2^{l(n)-1} \prod_{i=1}^{n} a_{i,x_i}})$$

Theorem 8.2.1
Then $F = \{F_n\}_{n \in \mathbb{N}}$ is an efficiently computable pseudorandom function family.

Proof:

To prove that this construction results in an efficiently computable pseudorandom function family, we need to show the three properties of pseudorandom function families:

1. Indexing
2. Polynomial Time Evaluation
3. Pseudorandomness

The first two properties are easy to see. The main part of the proof will be to show, that the function family we have constructed above has the third property.

1. Indexing:
   Each function has a unique index associated with it. This index is the key that identifies the function and consists of a Blum-Integer N, a vector $\vec{a} \in \{0, 1, \ldots, n-1\}^{2n}$, a quadratic residue in $g \in \mathbb{Z}_N^*$ and an n-bit string r. Thus, picking randomly a function in F is easy, if the values for $N, \vec{a}, g$, and r are available

2. Polynomial Time Evaluation:
   The construction above can easily be transformed into a polynomial time construction-algorithm that upon input of a key $\langle N, \vec{a}, g, r \rangle$ and an argument $x \in \{0, 1\}^n$ can compute $f_{N,\vec{a},g,r}(x)$ in polynomial time.

That leaves the third property to be proven. We need to show that $F = \{F_n\}_{n \in \mathbb{N}}$ is a pseudorandom function family. To do that, we will show that if there exists a distinguisher for F and $F_{n,l(n)}$ then there exists a probabilistic polynomial time algorithm to factor Blum-Integers with non-negligible success.
This proof works in three steps. In the first step, we will construct a predictor for the Goldreich-Levin hard core predicate. Upon input N, $v^2 \bmod N$, and r this predictor computes $B_r(u)$, where u is the unique quadratic residue in $\mathbb{Z}_N^*$ which satisfies $u^2 = v^2 \bmod N$. In the second step, we to use the Goldreich-Levin Theorem [11] to retrieve u. With that knowledge, we can extract square roots in $\mathbb{Z}_N^*$. And that implies the ability to factor Blum-Integers as described in [24]. That is the third step and completes the proof, as the underlying assumption is that factoring Blum-Integers is hard. But the heart of the proof is the first step.

First Step:

The first step is the biggest part of the proof. Here we will use the hybrid argument again. It is also the part of the proof that works analogously to the proof of the general constructions of pseudorandom function families in Chapter 6. We will repeat the first part of the proof, because the construction of the hybrid function we are going to use here is slightly different from the hybrid function in the general proof. That difference is caused by the different construction of $F = \{F_n\}_{n \in \mathbb{N}}$ here. In the general construction, the pseudorandom functions were constructed by repeated applications of a pseudorandom generator. Here the pseudorandom functions are generated by one application of a pseudorandom generator on an unpredictable function. That results in a different construction of the hybrid functions needed in the proofs. Therefore we have to elaborate why the general proof is still working here and where it needs to be modified.
We will start with assuming that there is a distinguisher for $F = \{F_n\}_{n \in \mathbb{N}}$ and

$F_{n,l(n)}$ the set of all functions with the same domain and range.

Lemma 8.2.1 (simplified version)
Let D be a distinguisher for $F = \{F_n\}_{n\in\mathbb{N}}$ and $F_{n,l(n)}$ for any choice of $\vec{a}$. Let u be the unique quadratic residue in $\mathbb{Z}_N^*$ that satisfies $u^2 = v^2 \bmod N$. Then there exists a predictor P for $B_r(u)$ with success probability significantly greater than $\frac{1}{2}$.

In this lemma, we assume that D distinguishes F and $F_{n,l(n)}$ for any choice of $\vec{a}$. We will give this simplified version of the proof first and then show how to modify it so that it works for a randomly sampled vector $\vec{a}$.

Proof of Lemma 8.2.1(simplified version):

We assume, that there exists a distinguisher D for $F = \{F_n\}_{n\in\mathbb{N}}$ and $F_{n,l(n)}$. Then for any $f_{N,\vec{a},g,r} \in F_n$ and any function f sampled uniformly in the set of all functions with the same domain and range, there exists a $k \in \mathbb{N}$ such that for all n big enough:

$$\text{prob}\ [D(f_{N,\vec{a},g,r}) = 1] - \text{prob}\ [D(f) = 1] > \frac{1}{n^k} \tag{8.1}$$

The next step is to prove that there is a distinguisher D' for $B_r(u)$ and a random bit b. This distinguisher D' gets as input N, $v^2 \bmod N$, r and a bit b which either equals $B_r(u)$ or is sampled randomly in $\{0, 1\}$. If D' outputs 1 it means, that D' decided that the input-bit b is equal to $B_r(u)$. If D' outputs 0 it means, that D' decided that the input-bit b is random.

Construction of D':

The construction of D' with input b works by simulating D with a hybrid function h as input. Then the output of D' with input b is the output of D with input h, i.e., $D'(b) = D(h)$.
This hybrid function h is constructed similarly to the pseudorandom functions $f_{N,\vec{a},g,r}$. Where $f_{N,\vec{a},g,r}$ is constructed by using the Goldreich-Levin hard core predicate, h is constructed partially from the Goldreich-Levin hard core predicate and partly consists of random bits. The hybrid function h starts to behave like a random function and at a certain point changes to behave like $f_{N,\vec{a},g,r}$. The distinguisher D makes $m(n)$ queries on the function it gets as input. For each query, the function produces $l(n)$ output bits. As in the previous constructions in the previous proofs, D' stores all the queries and the outputs and on each new query first checks, whether the same query has already been made. If this is the case, D' answers the query with the previously stored output value. As the point at which h starts to behave like $f_{N,\vec{a},g,r}$ can be within a query and not only between

two queries, there are a total of $l(n) \cdot m(n)$ many possibilities for this point of change. First of all, D' randomly chooses this point of change. This point can be divided up into $I \cdot l(n) + j$, where $I \in \{0, \ldots, m(n) - 1\}$ and $j \in \{0, \ldots, l(n) - 1\}$. The first value I determines in which query the change takes places and the second value j determines the exact position within the $(I + 1)$th query.
Next D' starts the simulation of D with input h. The main difficulty here is to define $\vec{a}$ and g in such a way, that h really behaves like a pseudorandom function starting with the previously chosen point of change. If that point happens to be exactly between two queries, there is no difficulty in defining $\vec{a}$ and g. The problem only arises, when the point of change is within one query. That is the common case and the one we will have a closer look at.

Let N be a uniformly sampled Blum-Integer. Let v be sampled uniformly in Q_N. Let r be sampled uniformly in $\{0, 1\}^n$. Let u be the unique quadratic residue in $\mathbb{Z}_N^*$, which satisfies $u^2 = v^2 \bmod N$.

Distinguisher D':

Input	:	v^2, N, r, b
Output	:	$D'(v^2, N, r, b) \in \{0, 1\}$
		Output 0 means, that D' decides that b is random.
		Output 1 means, that D' decides that b is equal to $B_r(u)$.

1. Pick two random values $I \in_U \{0, \ldots, m(n) - 1\}$ and $j \in_U \{0, \ldots, l(n) - 1\}$
2. Run $D(h)$
3. Answer the first I queries with random bit strings.
4. Sample $b_0 \ldots b_{j-1} \in_U \{0, 1\}^j$
5. Answer the $(I + 1)$'th query with $b_0, \ldots, b_{j-1}, b, B_r(u^2), \ldots, B_r(u^{2^{l(n)-j}})$
6. Answer the remaining queries according to $f_{N,\vec{a},g,r}$
7. $D'(v^2, N, r, b) := D(h)$

Definition of g and $\vec{a}$:
It remains to define g and $\vec{a}$. The definition of g depends on the choice of j. The definition of $\vec{a}$ depends on the $(I + 1)$'th query made by D.
Let x be this $(I + 1)$'th query. Then we need that the $(j + 1)$th bit of $f_{N,\vec{a},g,r}(x)$ equals $B_r(u)$. That is the bit in question, that is answered with b by D'. We also require that D' will be able to compute the last $l(n) - j$ bits of the $(I + 1)$'th query and the answers to all the following queries efficiently.

Remark 8.2.1
It is worth noticing here why in this case the hybrid argument cannot be applied in a reverse order. If D' would answer D's queries first according to $f_{N,\vec{a},g,r}$ and then according to f, it would not be possible to assign values to $\vec{a}$ and g such that the $(j+1)$'th bit of $f_{N,\vec{a},g,r}(x)$ equals $B_r(u)$.

Definition of g:
Let $s := l(n) \cdot n - j$.
Define g to be $v^{2^{l(n)\cdot n-j}} \bmod N$.
Define u to be $g^{2^{-(l(n)\cdot n-j)}}$.

Then we need to show that $u^2 = v^2 \bmod N$.
Let γ be the order of g in $\mathbb{Z}_N^*$. As v^2 is uniformly distributed in the set of quadratic residues in $\mathbb{Z}_N^*$ and as squaring is a permutation over the set of quadratic residues in $\mathbb{Z}_N^*$, it follows immediately that g^{2^s} is a uniformly distributed quadratic residue in $\mathbb{Z}_N^*$. Now we need to show that this definition of g leads to the desired result. We will show:

1. γ is odd:
   Since the orders of all quadratic residues in $\mathbb{Z}_N^*$ divide the size of Q_N so does γ, the order of g in $\mathbb{Z}_N^*$. Therefore, it is sufficient to show that the size of Q_N is odd. Since N is a Blum-Integer, exactly one of the four square roots of a quadratic residue in $\mathbb{Z}_N^*$ is in Q_N. This means that the size of $\mathbb{Z}_N^*$ is four times the size of Q_N. And we know the size of $\mathbb{Z}_N^*$. It is $(P-1)\cdot(Q-1)$, where $P \cdot Q = N$ and $P \equiv Q \equiv 3 \bmod 4$. Therefore $\frac{P-1}{2}$ and $\frac{Q-1}{2}$ are odd. And therefore $\frac{P-1}{2} \cdot \frac{Q-1}{2} = \frac{(P-1)(Q-1)}{4} = |Q_N|$ is odd as well. And as we said before, this implies that γ is odd as well.

   $\square$

2. for all $0 < i < s$, $g^{2^{-i} \bmod \gamma} = v^{2^{s-i}} \bmod N$:
   We know that γ is odd. Therefore, there exists $2 \in \mathbb{Z}_\gamma^*$. Then its inverse $2^{-1} \bmod \gamma$ exists as well. $2^{-1} \bmod \gamma$ simply is $\frac{\gamma+1}{2}$. Let $g^{2^{-1}}$ denote $g^{2^{-1} \bmod \gamma} \bmod N$ for simplicity of exposition. Now, whenever 2^{-1} appears in an exponent, it denotes $2^{-1} \bmod \gamma$. Analogously, whenever 2^{-i} appears in an exponent for any natural number $i > 1$, it denotes $2^{-i} \bmod \gamma$ which is $(\frac{\gamma+1}{2})^i$. Since g is a quadratic residue, this implies that for all $0 < i < s$ the exponentiation $g^{2^{-i}} \bmod N$ is a quadratic residue. Thus, for $i = 1$ we have that both $g^{2^{-1}}$ and $v^{2^{s-1}}$ are square roots of g and quadratic residues. And since squaring over the set of quadratic residues in $\mathbb{Z}_N^*$ is a permutation, $g^{2^{-1}}$ and $v^{2^{s-1}}$ must be equal. By induction over i, we get that $g^{2^{-i}} = v^{2^{s-i}} \bmod N$ for $0 < i < s$.

   $\square$

What we still need is that $u^2 \equiv v^2 \bmod N$. To this end, we define $u := g^{2^{-s}}$. We will prove later that this definition of u will yield the desired result.

Definition of $\vec{a}$:
The definition of $\vec{a}$ depends on the $(I+1)$'th query D makes. Therefore it is not necessary to give an explicit definition of $\vec{a}$ for the first I queries. Those queries are answered with random bits. Hence, we only need to define $\vec{a}$ from the $(I+1)$'th query onward.
Let $\vec{a} = \langle a_{1,0}, a_{1,1}, a_{2,0}, a_{2,1}, \ldots, a_{n,0}, a_{n,1} \rangle$. Then for all $1 \le i \le n$ define a_{i,x_i} to be $2^{-l(n)} \bmod \gamma$. Choose all $a_{i,\overline{x_i}}$ uniformly in $\{0, \ldots, N-1\}$.

Now, we have defined $\vec{a}$, g and u. We want to have:

$$B_r(g^{2^j \prod_{i=1}^{n} a_{i,x_i}}) = B_r(u)$$

And with the notations above we get:

$$g^{2^j \prod_{i=1}^{n} a_{i,x_i}} = g^{2^{j-l(n)n}} = g^{2^{-s}} = u \bmod N$$

That is what we wanted. Thus, we have a construction, where D' can answer all the remaining queries starting with the $(j+1)$'th bit of the $(I+1)$'th query in a way which is consistent with the definitions of $\vec{a}$ and g. And D can do that without actually knowing the values of the a_{i,x_i}.
Since $v^2 = u^2 \bmod N$ and since v^2 and r are part of the input of D', D' can answer D's $(I+1)$'th query with

$$b_0, b_1, \ldots, b_{j-1}, b, B_r(v^2), \ldots, B_r(v^{2^{l(n)-j}})$$

Now we still need to see that it is possible for D' to answer the remaining queries. In other words, we need to see, that D' can compute $f_{N,g,\vec{a},r}(y)$ for any $y \neq x$. To answer any query y with $f_{N,g,\vec{a},r}(y)$, the distinguisher D' needs to compute the value $g^{\prod_{i=1}^{n} a_{i,y_i}}$. Let us have a closer look at this:

$$g^{\prod_{i=1}^{n} a_{i,y_i}} = g^{(\prod_{y_i = x_i} a_{i,y_i})(\prod_{y_i \neq x_i} a_{i,y_i})}$$

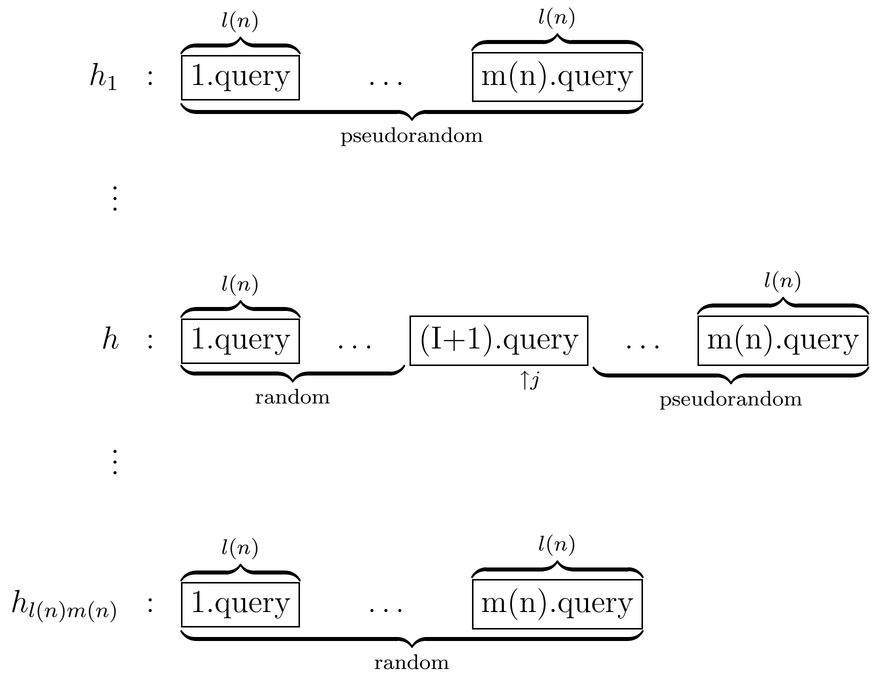
Let k be the number of locations for which y_i equals x_i. Then we get:

$$\begin{aligned} &= g^{(2^{-l(n)\cdot k})(\prod_{y_i \neq x_i} a_{i,y_i})} \\ &= v^{(2^{s-l(n)\cdot k})(\prod_{y_i \neq x_i} a_{i,y_i})} \bmod N \end{aligned}$$

As we are looking at values $y \neq x$, we know that $k < n$. We also know that $j < l(n)$ since j was chosen randomly in $\{0, \ldots, l(n) - 1\}$. Now if we look at

$$\begin{aligned} s - l(n) \cdot k &= l(n) \cdot n - j - l(n) \cdot k \\ &= (n - k) \cdot l(n) - j \end{aligned}$$

we get that this must be an integer greater than or equal to 1. Therefore, D' can compute $v^{(2^{s-l(n)\cdot k})(\prod_{y_i \neq x_i} a_{i,y_i})} \bmod N$. The distinguisher D' has all the necessary values as it also knows all the a_{i,y_i} for all $y_i \neq x_i$. That completes the construction of a distinguisher for $B_r(u)$ and a a random bit b.
Now we still need to prove that this distinguisher has non-negligible success probability. But if we look at what we already have, we will find that we get the same picture then in the proof of the general construction of pseudorandom function families on page 43.



If the input to D' is a random bit, then the constructed hybrid function h starts to behave pseudorandomly at the point $I \cdot l(n) + (j + 1)$. Therefore, h then is the function $h_{I \cdot l(n)+(j+1)}$. If the input to D' is a Goldreich-Levin hard core predicate, h starts to behave pseudorandomly at the point $I \cdot l(n) + j$ and h then is the function $h_{I \cdot l(n)+j}$.
The first hybrid function h_1 we get when $I = 0$ and $j = 0$. And it is a pseudorandom function, when the input to D' is a Goldreich-Levin hard core predicate.
The last hybrid function $h_{l(n)m(n)}$ we get when $I = m(n) - 1$ and $j = l(n) - 1$. And it is a random function when the input to D' is a random bit.
Therefore, we now have the exact same situation here as in the proof of the general

construction of pseudorandom function families and do not need to repeat the rest of the proof, which is an application of the Hybrid Theorem on page 13.
We assume that there exists a distinguisher D satisfying Equation 8.1 on page 65 with success probability $\delta_D(n) \geq \frac{1}{n^k}$. Using that distinguisher, we have constructed a distinguisher D' for the Goldreich-Levin hard core predicate. And the hybrid argument now tells us that this distinguisher D' has success probability at least $\delta_{D'}(n) \geq \frac{1}{m(n) \cdot l(n)} \cdot \delta_D(n) =: \frac{1}{n^{k'}}$. Hence, if D has a non-negligible success probability then D' has a non-negligible success probability as well.
The problem with D' is that the probabilities are taken over a relatively large set of variables. In order to construct a predictor for $B_r(u)$ we therefore need to take that into consideration.

The Averaging Argument

So far, we have a distinguisher D' for $B_r(u)$ and b with success probability $\delta_{D'}(n) \geq \frac{1}{n^{k'}}$. But this distinguisher works for fixed v^2, N and r. Therefore it needs to be shown, that on an at least an $\frac{1}{2 \cdot n^{k'}}$ -fraction of the possible choices for N and v^2 the distinguisher D' can tell apart $B_r(u)$ from a random bit b with success probability at least $\frac{1}{2 \cdot n^{k'}}$ over the choices of r. This distinguisher D' can be used to construct a predictor P for $B_r(u)$ with success probability at least $\frac{1}{2} + \frac{1}{2 \cdot n^{2 \cdot k'}}$. This will be done by using a common averaging argument.
To show that D' has success probability at least $\frac{1}{2 \cdot n^{k'}}$ for at least an $\frac{1}{2 \cdot n^{k'}}$ -fraction of the possible choices for N and v^2, let us look at the following sets K_1 and K_2.
Let K_1 be the set of (N, v^2)-pairs for which D' has success probability at least $\frac{1}{2 \cdot n^{k'}}$.
Let K_2 be the set of (N, v^2)-pairs for which D has an success probability of less then $\frac{1}{2 \cdot n^{k'}}$.

$$
\begin{aligned}
K_1 &:= \{(N, v^2) | D' \text{ has success probability } \geq \frac{1}{2 \cdot n^k}\} \\
K_2 &:= \{(N, v^2) | D' \text{ has success probability } < \frac{1}{2 \cdot n^k}\}
\end{aligned}
$$

Then the success probability $\delta_{D'}(n) = \frac{1}{n^{k'}}$ of D' is equal to the sum of the probability of a pair (N, v^2) being in K_1 times the success probability D' has for (N, v^2)-pairs in K_1 plus the probability of a pair (N, v^2) being in K_2 times the success probability D' has for (N, v^2)-pairs in K_2. The maximum success probability of D' for (N, v^2)-pairs in K_1 is at least $\frac{1}{2 \cdot n^{k'}}$. In this case D' can have a maximum success probability of 1. For (N, v^2)-pairs in K_2, D' can have a maximum success probability D' of $\frac{1}{2 \cdot n^{k'}}$.

Therefore

$$\frac{1}{n^{k'}} \leq \delta_{D'}(n) \leq \underbrace{\text{prob } [(N,v^2) \in K_1]}_{:=p} \cdot 1 + \underbrace{\text{prob } [(N,v^2) \in K_1]}_{1-p} \cdot \frac{1}{2 \cdot n^{k'}}$$

$$= p + (1-p) \cdot \frac{1}{2 \cdot n^{k'}}$$

$$\Leftrightarrow \qquad \frac{1}{n^{k'}} \leq p + \frac{1}{2 \cdot n^{k'}} - p \cdot \frac{1}{2 \cdot n^{k'}}$$

$$\Leftrightarrow \qquad \frac{1}{2n^{k'}} \leq p \cdot (1 - \frac{1}{2 \cdot n^{k'}})$$

And as $\frac{1}{2 \cdot n^{k'}}$ is positive valued, we get:

$$\frac{1}{2n^{k'}} \leq p \cdot (1 - \frac{1}{2 \cdot n^{k'}}) \leq p$$

So now we have:

$$p \geq \frac{1}{2 \cdot n^{k'}}$$

And as p is the probability for a (N, v^2)-pair to be in K_1, we have

$$\text{prob } [(N,v^2) \in K_1] \geq \frac{1}{2 \cdot n^{k'}}$$

and as therefore the probability for a (N, v^2)-pair to be in K_2 is $(1-p)$ we have

$$\text{prob } [(N,v^2) \in K_1] \leq (1 - \frac{1}{2 \cdot n^{k'}}).$$

This implies that the normalized size of K_1 is at least $\frac{1}{2 \cdot n^{k'}}$ and that of K_2 therefore is at most $1 - \frac{1}{2 \cdot n^{k'}}$. The sizes of K_1 and K_2 are normalized such that they sum up to 1.

$$|K_1| \geq \frac{1}{2 \cdot n^{k'}}$$

$$|K_2| \leq 1 - \frac{1}{2 \cdot n^{k'}}$$

So now we have that D' has success probability at least $\frac{1}{2 \cdot n^{k'}}$ for at least an $\frac{1}{2 \cdot n^{k'}}$ fraction of the possible choices for N and v^2.
The next step is to construct a predictor P for $B_r(u)$. We have shown in Chapter 4 that such a predictor exists. There we have proven, that there exists an i, such that there is a predictor for the i'th bit under B. And as B is a probability distribution on $\{0,1\}$, there is only one possible i. For this $i = 1$, we now give a predictor.

Predictor P:

Input: v^2, N, r
Output: $P(v^2, N, r)$

- Choose $b \in_U \{0,1\}$
- Run $D'(v^2, N, r, b)$
- $P(v^2, N, r) := D'(v^2, N, r, b) \oplus b \oplus 1$

Now let us look at the success rate $\sigma_P(B)$ of P. The success rate of P is directly correlated to the success probability of D'. The success probability of D' can be divided up into the cases of the (N, v^2)-pairs that are in K_1 and K_2 respectively. Thus can the success rate of P. If the (N, v^2)-pair is in K_1, D' has success probability at least $\frac{1}{2 \cdot n^{k'}}$. That implies, that in that case P has a success rate of $\frac{1}{2} + \frac{1}{2 \cdot n^{k'}}$. In the second case, where the (N, v^2)-pair is in K_2, the success probability of D' is less than $\frac{1}{2 \cdot n^{k'}}$. We only know that the success rate of P is less than $\frac{1}{2} + \frac{1}{2 \cdot n^{k'}}$ in this case. But we know that the success rate of a predictor is always at least $\frac{1}{2}$. Therefore, that is all we know about this special case.
So the worst possible case for the success rate of P is, when K_1 is as small as possible and for the (N, v^2)-pairs in K_1 the success rate is the lowest possible, which is $\frac{1}{2} + \frac{1}{2 \cdot n^{k'}}$. Then K_2 is as big as possible, which is $1 - \frac{1}{2 \cdot n^{k'}}$ and P has a success rate of $\frac{1}{2}$. As all (N, v^2)-pairs are either in K_1 or K_2 the total success probability $\delta(n)$ of P adds up to:

$$
\begin{aligned}
\sigma_P(B) &\geq |K_1| \cdot \left(\frac{1}{2} + \frac{1}{2 \cdot n^{k'}}\right) + |K_2| \cdot \frac{1}{2} \\
&\geq \frac{1}{2 \cdot n^{k'}} \cdot \left(\frac{1}{2} + \frac{1}{2 \cdot n^{k'}}\right) + \left(1 - \frac{1}{2 \cdot n^{k'}}\right) \cdot \frac{1}{2} \\
&= \frac{1}{4 \cdot n^{k'}} + \left(\frac{1}{2 \cdot n^{k'}}\right)^2 + \frac{1}{2} - \frac{1}{4 \cdot n^{k'}} \\
&= \frac{1}{2} + \left(\frac{1}{2 \cdot n^{k'}}\right)^2
\end{aligned}
$$

That completes the proof of Lemma 8.2.1 on page 65 and finishes the simplified proof of the first step of the proof of the NRR-Construction.
To complete the proof, we still need to show that Lemma 8.2.1 on page 65 also works for a randomly chosen vector $\vec{a}$. Therefore, we are now giving the general version of this lemma:

Lemma 8.2.2 (general version)
Let D be a distinguisher for $F = \{F_n\}_{n\in\mathbb{N}}$ and the set of all functions with the same range and domain for a randomly chosen vector $\vec{a}$. *Then there exists a predictor $\overline{P}$ for $B_r(u)$, where u is the unique quadratic residue in $\mathbb{Z}_N^*$ that satisfies $u^2 = v^2 \bmod N$.*

Proof of Lemma 8.2.2 (general version):

This proof works exactly as the simplified version. We first construct a distinguisher $\overline{D'}$ for U_1 and B, where B is the probability distribution on $\{0,1\}$ generated by the Goldreich-Levin hard core predicate. Then we use this distinguisher $\overline{D'}$ to construct a predictor $\overline{P}$ for $B_r(uw)$, where w is a variable, that we will define later. The construction of $\overline{D'}$ is identical to the previous one except for adding additional "randomization" elements on the values in $\vec{a}$. All the a_{i,x_i} we defined to be $2^{-l(n)} \bmod \gamma$. Next we add a random value r_i on each a_{i,x_i} and get a composed vector $\vec{a'}$ with $a'_{i,x_i} = r_i + 2^{-l(n)} \bmod \gamma$ for all $1 \le i \le n$. That causes a change in the $(j+1)$'th bit of the answer to the $(I+1)$'th query. Instead of

$$B_r(g^{2^j \prod_{i=1}^{n} a_{i,x_i}}) = B_r(g^{2^j \prod_{i=1}^{n} 2^{-l} \bmod \gamma})$$

we get

$$B_r(g^{2^j \prod_{i=1}^{n} a'_{i,x_i}}) = B_r(g^{2^j \prod_{i=1}^{n} (r_i + 2^{-l}) \bmod \gamma}) =: B_r(u \cdot w)$$

where w is completely determined by the r_i and by v^2. But we did not say yet when the r_i are being sampled. The first idea that comes to ones mind is to let $\overline{D'}$ sample those values. But then those values and therefore the value of w would change with each new call of $\overline{D'}$. That would hinder the reconstruction algorithm for the argument of the Goldreich-Levin hard core predicate. The solution to that problem is to pick the r_i in advance and then start $\overline{D'}$.
We are going to show now, that w is efficiently computable given the values of the r_i and v^2.
Define the polynomial p to be:

$$p(x) = \prod_{i=1}^{n} (r_i + x) = x^n + \sum_{i=0}^{n-1} \beta_i x^i$$

The coefficients β_i can be efficiently calculated from the r_i either recursively or by interpolation. Then we get:

$$\begin{aligned}
g^{2^j \prod_{i=1}^{n} a'_{i,x_i}} &= g^{2^j \prod_{i=1}^{n} (r_i + 2^{-l})} \\
&= g^{2^j p(2^{-l})} \\
&= g^{2^{j-l(n)\cdot n} + \sum_{i=0}^{n-1} \beta_i 2^{-(l(n)\cdot i - j)}} \\
&= g^{2^{-s}} \cdot v^{\sum_{i=0}^{n-1} \beta_i 2^{s-(l(n)\cdot i - j)}} \\
&= u \cdot \underbrace{v^{\sum_{i=0}^{n-1} \beta_i 2^{l(n)\cdot(n-i)}}}_{=w} \qquad = u \cdot w \bmod N
\end{aligned}$$

Now, we know that $\overline{D'}$ can calculate w. As $\overline{D'}$ also knows v^2, it has all the necessary information to calculate the remaining bits of the answer to the $(I+1)$'th query. For example the $(j+1)$'th bit of the answer to the $(I+1)$'th query is $B_r((uw)^2) = B_r(v^2 w^2)$ and that can be calculated by $\overline{D'}$ as we have shown.
We will now show that $\overline{D'}$ is also able to efficiently compute $f_{N,\vec{a'},g,r}(y)$ for any $y \neq x$ and therefore answer the remaining queries. To show that, it is enough to prove that $\overline{D'}$ can efficiently compute $g^{\prod_{i=1}^{n} a'_{i,x_i}}$.
The distinguisher $\overline{D'}$ starts with computing the coefficients δ_i of the polynomial $q(x) = \prod_{y_i = x_i} (r_i + x) = \sum_{i=0}^{k} \delta_i x^i$ where k is the number of locations where $x_i = y_i$. Then we have that $\prod_{x_i = y_i} a'_{i,y_i} = q(2^{-l(n)})$. Thus, we can calculate:

$$\begin{aligned}
g^{\prod_{i=1}^{n} a'_{i,y_i}} &= g^{(\prod_{y_i = x_i} a'_{i,y_i})\cdot(\prod_{y_i \neq x_i} a'_{i,y_i})} \\
&= g^{(\sum_{i=0}^{k} \delta_i 2^{-l(n)i})\cdot(\prod_{y_i \neq x_i} a'_{i,y_i})} \\
&= \prod_{i=0}^{k} g^{(\delta_i 2^{-l(n)i})\cdot(\prod_{y_i \neq x_i} a'_{i,y_i})} \\
&= \prod_{i=0}^{k} v^{(2^{s-l(n)i})\cdot(\delta_i \prod_{y_i \neq x_i} a'_{i,y_i})} \bmod N
\end{aligned}$$

Since $i \leq k < n$ as $y \neq x$ and $j < l(n)$ we have that $s - l(n)i = (n-i)l - j \geq 1$. And hence $\overline{D'}$ is able to compute all the required values by exponentiations of v^2. So now we have shown, that the proof works with a composed vector $\vec{a'}$. What still needs to be proven is that such a composed vector $\vec{a'}$ is close enough to a uniformly sampled vector $\vec{a}$. So let us now look at the statistical distance between a uniformly chosen vector $\vec{a}$ and the composed vector $\vec{a'}$.

Corollary 8.2.1
Let r_i and a_{i,x_i} be sampled uniformly in $\{0,\ldots,N-1\}$. Let $a'_{i,x_i} = r_i + 2^{-l} \bmod \gamma$. Then the statistical distance between a'_{i,x_i} and a_{i,x_i} is negligible, i.e. there is no $k \in \mathbb{N}$ such that the statistical distance between a'_{i,x_i} and a_{i,x_i} is at least as big as $\frac{1}{n^k}$ for n big enough.

We know that γ divides $(P-1)(Q-1)$. Let a be a probability variable in $\{0,\ldots,N-1\}$. Then we have that

$$\begin{aligned} & \text{prob}\left[a = a'_{i,x_i} \bmod \gamma | r_i \in \{0,\ldots,(P-1)(Q-1)-1\}\right] \\ = \; & \text{prob}\left[a = a_{i,x_i} \bmod \gamma | a_{i,x_i} \in \{0,\ldots,(P-1)(Q-1)-1\}\right] \end{aligned}$$

And that means that the distribution of the a'_{i,x_i} conditioned on the event that $r_i \in \{0,\ldots,(P-1)(Q-1)-1\}$ is the same as the distribution of the a_{i,x_i} conditioned on the event that $a_{i,x_i} \in \{0,\ldots,(P-1)(Q-1)-1\}$. And that is just the uniform distribution over $\{0,\ldots,\gamma-1\}$. What we still need now is the probability that $r_i \in \{0,\ldots,(P-1)(Q-1)-1\}$.

$$\begin{aligned} & \text{prob}\left[r_i \in \{0,\ldots,(P-1)(Q-1)-1\}\right] \\ = \; & \text{prob}\left[a_{i,x_i} \in \{0,\ldots,(P-1)(Q-1)-1\}\right] \\ = \; & \frac{(P-1)(Q-1)}{N} \\ = \; & \frac{PQ-P-Q+1}{N} \\ = \; & 1 - \frac{P+Q}{N} + \frac{1}{N} \end{aligned}$$

which for some fixed $l \in \mathbb{N}$ is

$$\geq \; 1 - \frac{1}{n^l}$$

as $N = P \cdot Q$ is an n-bit Blum-Integer.
And therefore we have that the statistical distance between a_{i,x_i} and a'_{i,x_i} is negligible.

□

So now let us look at the functions $f_{N,\vec{a'},g,r}$ generated by variables $\vec{a'}$ and g as we defined them above in comparison to the functions $f_{N,\vec{a},g,r}$ generated by randomly chosen $\vec{a}$ and g according to the construction in Theorem 8.2.1 on page 63.

Corollary 8.2.2
The statistical distance between $f_{N,\vec{a'},g,r}$ and $f_{N,\vec{a},g,r}$ is negligible.

We just have proven that for all $1 \leq i \leq n$ we have a negligible statistical distance between a_{i,x_i} and a'_{i,x_i}. By the triangle inequality the statistical distance between $\vec{a}$ and $\vec{a'}$ is at most n times as big. We get a maximum distance of $n \cdot \frac{1}{n^l}$ between those two vectors. It follows immediately that that is the statistical distance of $f_{N,\vec{a'},g,r}$ and $f_{N,\vec{a},g,r}$ as well.

□

Now as the statistical distance between two functions $f_{N,\vec{a'},g,r}$ and $f_{N,\vec{a},g,r}$ is at most $n \cdot \frac{1}{n^l}$, any distinguisher M between those functions has that same success probability. This is the case, since applying any function, even a randomized one, does not change the statistical distance between $f_{N,\vec{a'},g,r}$ and $f_{N,\vec{a},g,r}$. Therefore we have that:

$$\text{prob}\,[M(f_{N,\vec{a},g,r}) = 1] - \text{prob}\,[M(f_{N,\vec{a'},g,r}) = 1] \leq n \cdot \frac{1}{n^l}$$

In the construction of the distinguisher D' in the proof of Lemma 8.2.1 on page 65 we use an adapted vector $\vec{a}$ and get the success probability $\delta_{D'}(n) \geq \frac{1}{n^{k'}}$.
As we are using a function $f_{N,\vec{a'},g,r}$ in the construction of the distinguisher $\overline{D'}$ in the proof of Lemma 8.2.2 on page 73, the success probability $\delta_{\overline{D'}}(n)$ of $\overline{D'}$ differs from $\delta_{D'}$ by the amount of the statistical distance between $f_{N,\vec{a'},g,r}$ and $f_{N,\vec{a},g,r}$. Now, we have a distinguisher $\overline{D'}$ for the Goldreich-Levi hard core predicate with success probability at least $\frac{1}{n^{k'}} \pm n \cdot \frac{1}{n^l}$.
Instead of a distinguisher for $B_r(u)$ and a random bit b we have a distinguisher for $B_r(u \cdot w)$ and a random bit b.

Lemma 8.2.3
Assume that there exists a distinguisher for $F = \{F_n\}_{n \in \mathbb{N}}$ and $F_{n,l(n)}$.
Then there exists a predictor $\overline{P}$ for $B_r(uw)$.

Proof:

We get this predictor by applying the averaging argument the way we did on page 70. The predictor gets N, $v^2 \bmod N$, r and $\vec{r}$ as input. Here N is a randomly chosen Blum-Integer, v is randomly chosen in $\mathbb{Z}_N^*$, r is randomly chosen in $\{0,1\}^n$, $\vec{r}$ is a random vector of $2n$ elements in $\{0,\dots,N-1\}$, $w = w(\vec{r}, v^2)$ is a quadratic residue in $\mathbb{Z}_N^*$ and uw is the unique quadratic residue in $\mathbb{Z}_N^*$ which satisfies $(uw)^2 = v^2 \cdot w^2 \bmod N$. The predictor $\overline{P}$ we get is identical to the one on page 72 except that this predictor calls the modified distinguisher $\overline{D'}$ instead of D'.

Predictor $\overline{P}$:

Input: v^2, N, r
Output: $\overline{P}(v^2, N, r)$

- Choose $b \in_U \{0,1\}$
- Run $\overline{D'}(v^2, N, r, b)$
- $\overline{P}(v^2, N, r) := \overline{D'}(v^2, N, r, b) \oplus b \oplus 1$

By the same argument as in the case of the modified distinguisher $\overline{D'}$ the success rate of $\overline{P}$ differs from that of P only by the statistical distance between $f_{N,\vec{a'},g,r}$ and $f_{N,\vec{a},g,r}$. Therefore this predictor $\overline{P}$ has success rate

$$\begin{aligned}\sigma_{\overline{P}}(n) &\geq \sigma_P(n) \pm \frac{1}{n^{k'-1}}\\ &\geq \frac{1}{2} + (\frac{1}{2 \cdot n^{k'}})^2 \pm \frac{1}{n^{k'-1}}\end{aligned}$$

The predictor $\overline{P}$ has a polynomial advantage. That completes the proof of Lemma 8.2.2 on page 73.

□

The first step of the proof of Theorem 8.2.1 on page 63 is finished.

□

Second Step:

In the second step of the proof of Theorem 8.2.1 on page 63, we will use the Goldreich-Levin hard core predicate, to recover the value uw from $B_r(uw)$.

Theorem 8.2.2 (Goldreich-Levin-Theorem)
Let z and r be n-bit strings. Assume there exists a predictor for $B_r(z)$ with success probability $\frac{1}{n^k}$ and running time $t(n)$ where the probabilities are taken over the choices of r. Then there exists a probabilistic polynomial time algorithm R that retrieves z with success probability $\Omega(\frac{1}{n^k})$ and running time $O(\frac{n^2 \cdot t(n)}{\frac{1}{n^k}^2})$.

We are not going to give the proof of this theorem here. It is somewhat technical and can be looked up in [13] as well as in [15] and [11].
But in order to use this theorem on $B_r(uw)$ we need to modify the predictor P from above for $B_r(uw)$ in such a way, that the value of w does not change every time the predictor is invoked. As

$$w = v^{\sum_{i=0}^{n-1} \beta_i 2^{l(n)\cdot(n-i)}}$$

where the β_i are the coefficients of the polynomial

$$p(x) = \prod_{i=1}^{n} (r_i + x) = x^n + \sum_{i=0}^{n-1} \beta_i x^i$$

we need to fix the values N, $v^2 \bmod N$ and $\vec{r}$ in advance. That way the predictor can be invoked with different values r and fixed w.

Predictor P':

Input: $v^2, N, \vec{r}, r$
Output: $P'(v^2, N, \vec{r}, r)$

- Choose $b \in_U \{0, 1\}$
- Run $D'(v^2, N, \vec{r}, r, b)$
- $P'(v^2, N, \vec{r}, r) := D'(v^2, N, \vec{r}, r, b) \oplus b \oplus 1$

Now the value of w does not change every time P is invoked, as the vector $\vec{r}$ as well as N and $v^2 \bmod N$ are fixed in advance. This predictor P' can now be used as an oracle to reconstruct uw.
We get the desired result. Assume that there exists a distinguisher for $F = \{F_n\}_{n\in\mathbb{N}}$.

Let the predictor P' be defined as above. Then there exists a probabilistic polynomial time algorithm R that retrieves z with success probability $\Omega(\frac{1}{n^k})$ and running time $O(\frac{n^2 \cdot t(n)}{(\frac{1}{n^k})^2})$.

□

Third Step:

In the third part of the proof of Theorem 8.2.1 on page 63, we show that we now can extract square roots in $\mathbb{Z}_N^*$ and subsequently factor Blum-Integers.

Lemma 8.2.4
Assume there exists a distinguisher D for $F = \{F_n\}_{n \in \mathbb{N}}$ and the set of all functions with the same domain and range. Then there exists a probabilistic polynomial time algorithm A that factors Blum-Integers with non-negligible success probability.

Proof:
To prove this lemma, we will give the algorithm A that factors Blum-Integers with non-negligible success probability.

Factor-Algorithm A:

Input: N

1. Choose $v \in_U \mathbb{Z}_N^*$
2. Compute $v^2 \bmod N$
3. Choose $\vec{r} = \langle r_1 \dots r_n, r'_1 \dots r'_n \rangle \in_U \{0, \dots, N-1\}^{2n}$
4. Compute $w := v^{\sum_{i=0}^{n-1} \beta_i 2^{l(n) \cdot (n-i)}} \bmod N$ where the β_i are the coefficients of the polynomial $p(x) = \prod_{i=1}^{n} (r_i + x) = x^n + \sum_{i=0}^{n-1} \beta_i x^i$.
5. Invoke the Goldreich-Levin reconstruction algorithm R:
   R invokes $P'(B_r(uw))$ for different values r.
   Denote the output of R with z.
6. Compute $u = z \cdot w^{-1} \bmod N$.
   If $z = u \cdot w$ (R has the correct output) then $u^2 = v^2 \bmod N$.

   - If $u \neq \pm v \bmod N$, output $gcd(u - v, N)$ $(\in \{P, Q\})$
   - If $u = \pm v \bmod N$, output "failed"

Running time of A:
This algorithm clearly runs in polynomial time. The steps 1, 2, 3, 4, and 6 can be carried out efficiently and are independent of the predictor P'. In step 5 we use the Goldreich-Levin Theorem. This Theorem guarantees that R is a polynomial time algorithm as we have shown that P' has a non-negligible success rate.
Therefore A is a probabilistic polynomial time algorithm.

Success-Probability of A:
We have shown that with probability $\frac{1}{2n^{k'}}$ the predictor P' predicts $B_r(uw)$ with advantage $(\frac{1}{2n^{k'}})^2$ over $\frac{1}{2}$. From the Goldreich-Levin Theorem on page 78 we know that R retrieves uw with success probability at least $\Omega(\frac{1}{n^k})$ over the choices of N, $v^2 \bmod N$ and $\vec{r}$. The variables uw and w are both quadratic residues in $\mathbb{Z}_N^*$. Hence, u is also a quadratic residue in $\mathbb{Z}_N^*$. And as every element in $\mathbb{Z}_N^*$ has exactly four distinct square roots, it follows that u equals $\pm v \bmod N$ with probability exactly $\frac{1}{2}$. Therefore, we can now conclude that A has success probability $\Omega((\frac{1}{n^k})^2)$.

That finishes the proof of Theorem 8.2.1 on page 63.

□

8.2.3 Efficiency of the NRR-Functions

Consider a function $f_{N,g,\vec{a},r}$ defined as in the Construction Theorem 8.2.1 on page 63. We will see that each evaluation requires only a constant number of modular multiplications per output bit. The whole output can be computed at the cost of fewer than three modular exponentiations.
The analysis of this function is very similar to the previous one on page 57 as this construction resembles the one from Theorem 7.2.1 on page 50. This construction is an improvement of the one in the last chapter in the way that the number of output bits is much more easily expandable.
To compute $f_{N,g,\vec{a},r}$ for any possible input $x \in \{0,1\}^n$, it is necessary to compute $f_{N,g,\vec{a},r}(x) = B_r(g^{\prod_{i=1}^{n} a_{i,x_i}}), \ldots, B_r(g^{2^{l(n)-1} \prod_{i=1}^{n} a_{i,x_i}})$. That involves one multiple product $y = \prod_{i=1}^{n} a_{i,x_i} \bmod \varphi(N)$, one modular exponentiation $z = g^y \bmod N$ and $l(n) - 2$ successive modular squarings. What remains to be done, is to calculate the Goldreich-Levin hard core predicate $l(n)$ times. But that is a cheap operation compared to modular multiplication. The modular exponentiation can be counted as n modular multiplications and the modular squarings require less than one modular multiplication each. As was discussed in the previous chapter, the efficiency can be improved by preprocessing techniques. The most obvious preprocessing is to

calculate all the g^{2^i} for all $i \in \mathbb{N}$ up to the length of $\varphi(N) = (P-1)(Q-1)$. Given that, there remain $2n + l(n)$ modular multiplications per evaluation of $f_{N,g,\vec{a},r}$. This result is very similar to the one from the previous chapter. As the same possibilities for preprocessing are given here, we get similar results. The two computations $y = \prod_{i=1}^{n} a_{i,x_i} \bmod \varphi(N)$ and $z = g^y \bmod N$ can be done in TC_5^0. The only difference here is, that the modular multiplications per output bit need to be done after $z = g^y \bmod N$ was computed. Modular multiplications can be done in TC_3^0, as was shown in [29, 30]. Therefore we get the result, that the functions $f_{N,g,\vec{a},r}$ are computable in TC_8^0 :

Remark 8.2.2
Let $F = \{F_n\}_{n\in\mathbb{N}}$ be as in Theorem 8.2.1 on page 63.
Then there exists a polynomial p such that for all functions $f_{N,g,\vec{a},r}$ in F there exists a depth 8 threshold circuit of size bounded by $p(n)$ that computes $f_{N,g,\vec{a},r}$.

Therefore this construction is not more efficient than the previous one. But this construction holds another feature, that is interesting to note here.
The construction we examine here allows for a variable output length. There is a natural way of expanding the output length of $f_{N,g,\vec{a},r}$ by simply computing more bits of the form $B_r(g^{2^j})$.
While expanding the output is also possible with the general construction of pseudorandom function families and the special construction we examined in the last chapter, here we get it "almost for free". Each additional output bit costs only one additional modular multiplication.

Literaturverzeichnis

[1] Adams, D. *The Hitch Hiker's Guide to the Galaxy.* Pan Books Ltd., London, 1980.

[2] Beame, P., Cook, S., and Hoover, H. Log depth circuits for division and related problems. *SIAM J. Comput. 15* (1986), 994–1003.

[3] Biham, E., Boneh, D., and Reingold, O. Breaking generalized Diffie-Hellmann modulo a composite is no easier than factoring. Theory of Cryptography Library. Record 97-14 at: http://theory.lcs.mit.edu/ tcryptol/homepage.html.

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

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

[6] Boneh, D., and Lipton, R. Algorithms for black-box fields and their application to cryptography. In *Advances in Cryptology - CRYPTO'96, Lecture Notes in Computer Science* (1996), vol. 1109, Springer, pp. 283–297.

[7] Boneh, D., and Venkatesan, R. Hardness of computing most significant bits in secret keys in Diffie-Hellman and related schemes. In *Advances in Cryptology - CRYPTO '96, Lecture Notes in Computer Science* (1996), vol. 1109, Springer, pp. 129–142.

[8] Brickell, E., Gordon, D., McCurley, K., and Wilson, D. Fast exponentiation with precomputation. In *Proc. Advances in Cryptology - EUROCRYPT'92, Lecture Notes in Computer Science* (1992), Springer, pp. 200–207.

[9] Canetti, R., Friedlander, J., and Shparlinski, I. On certain exponential sums and the distribution of Diffie-Hellman triples, 1997. preprint.

[10] Diffie, W., and Hellman, M. New directions in cryptography. In *IEEE Trans. Inform. Theory* (1976), vol. 22(6), pp. 644–654.

[11] Goldreich, O. *Foundations of Cryptography (Fragments of a Book).* Electronic Publication, 1995. http://www.eccc.uni-trier.de/eccc/info/ECCC-Books/eccc-books.html (Electronic Colloquium on Computational Complexity).

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

[13] Goldreich, O., and Levin, L. A hard-core predicate for all one-way functions. In *Proc. 21st Ann. ACM Symp. on Theory of Computing* (1989), pp. 25–32.

[14] Goldwasser, S., and Bellare, M. Lecture notes on cryptography, 1999.

[15] Luby, M. *Pseudo-randomness and applications.* Princton University Press, 1996.

[16] Maurer, U. Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete logarithms. In *Advances in Cryptology - CRYPTO 94, Lecture Notes in Computer Science* (1994), vol. 740, Springer, pp. 271–281.

[17] McCurley, K. A key distribution system equivalent to factoring. *J. of Cryptology 1* (1988), 95–105.

[18] McCurley, K. The discrete logarithm problem. In *Cryptography and Computational Number Theory Proc. Symp. Appl. Math., AMS Lecture Notes* (1990), vol. 42, pp. 49–74.

[19] Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. *Handbook of Applied Cryptography.* CRC Press Inc., Boca Raton FL., 1997. http://www.cacr.math.uwaterloo.ca/hac.

[20] Naor, M., and Reingold, O. Number-theoretic constructions of efficient peudo-random functions. In *Proc. 38th IEEE Symposium on Foundations of Computer Science* (1997), pp. 458–467.

[21] Naor, M., and Reingold, O. From unpredictability to indistinguishability: A simple construction of pseudo-random functions from MAC's. In *Advances in Cryptology - CRYPTO'98, Lecture Notes in Computer Science* (1998), vol. 1462, Springer, pp. 267–282.

[22] Naor, M., Reingold, O., and Rosen, A. Pseudo-random functions and factoring. In *Electronic Colloquium on Complexity* (2001), vol. Report No.64.

[23] Odlyzko, A. Discrete logarithms and smooth polynomials. *Contemporary Mathematics,AMS* (1993).

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

[25] Reif, J. On treshold circuits and polynomial computation. In *Proc. of the 2nd Conference on Structure in Complexity Theory* (1987), pp. 118–123.

[26] Reif, J., and Tate, S. On treshold circuits and polynomial computation. *SIAM J. Comput. 5* (1992), 896–908.

[27] Shmuely, Z. *Composite Diffie-Hellman public-key generating systems are hard to break.* Technical Report 356, Computer Science Department, Technion, Israel, 1985.

[28] Shoup, V. Lower bounds for discrete logarithms and relatede problems. In *Proc. Advances in Cryptology - EUROCRYPT '97* (1997).

[29] Siu, K.-Y., Bruck, J., Kailath, T., and Hofmeister, T. Depth efficient neural network for division and related problems. *IEEE Trans. Inform. Theory 39* (1993), 946–956.

[30] Siu, K.-Y., and Roychowdhury, V. On optimal depth treshold circuits for multiplication and related problems. *SIAM J. Disc. Math. 7(2)* (1994), 284–292.

[31] Stinson, D. R. *Cryptography, Theory and Practice.* CRC Press Inc., Boca Raton FL., 1995. stinson@bibd.unl.edu.

[32] Yao, A. C. Theory and applications of trapdoor functions. In *23rd Annual IEEE Symposium on Foundations of Computer Science* (Chicago IL, 1982), IEEE Computer Society Press, pp. 80–91.