Lecture Notes on Cryptography

Shafi Goldwasser[1]   Mihir Bellare[2]

August 1999

[1] MIT Laboratory of Computer Science, 545 Technology Square, Cambridge, MA 02139, USA. E-mail: `shafi@theory.lcs.mit.edu` ; Web page: `http://theory.lcs.mit.edu/ shafi`

[2] Department of Computer Science and Engineering, Mail Code 0114, University of California at San Diego, 9500 Gilman Drive, La Jolla, CA 92093, USA. E-mail: `mihir@cs.ucsd.edu` ; Web page: `http://www-cse.ucsd.edu/users/mihir`

Foreword

This is a set of lecture notes on cryptography compiled for 6.87s, a one week long course on cryptography taught at MIT by Shafi Goldwasser and Mihir Bellare in the summers of 1996, 1997, 1998 and 1999. Some of the materiel is based on courses we have taught at our respective institutions in the past; some is inspired from survey papers in the field; the rest is newly written.

Cryptography is of course a vast subject. The thread followed by these notes is to develop and explain the notion of provable security and its usage for the design of secure protocols.

The Teaching Assistant for the course in '96 was Rosario Gennaro. He contributed Section 9.6, Section 11.4, Section 11.5, and Appendix D to the notes, and also compiled, from various sources, some of the problems in Appendix E. Much of the material in Chapters 2, 3 and 7 is a result of scribe notes, originally taken by MIT graduate students who attended Professor Goldwasser's *Cryptography and Cryptanalysis* course over the years, and later edited by Frank D'Ippolito who was a teaching assistant for the course in 1991. Frank also contributed much of the advanced number theoretic material in the Appendix. Some of the material in Chapter 3 is from the chapter on Cryptography, by R. Rivest, in the Handbook of Theoretical Computer Science. Chapters 4, 5, 6, 8 and 10, and Sections 3.5, 9.5 and 7.4.6, are based on Bellare's course notes for the cryptography and network security course (CSE207) he teaches at UCSD.

Shafi Goldwasser and Mihir Bellare — Cambridge, Massachusetts, August 1999.

Table of Contents

Chapter 1

Introduction to Modern Cryptography

Cryptography is about communication in the presence of an adversary. It encompasses many problems (encryption, authentication, key distribution to name a few). The field of modern cryptography provides a theoretical foundation based on which we may understand what exactly these problems are, how to evaluate protocols that purport to solve them, and how to build protocols in whose security we can have confidence. We introduce the basic issues by discussing the problem of encryption.

1.1 Encryption: Historical Glance

The most ancient and basic problem of cryptography is secure communication over an insecure channel. Party A wants to send to party B a secret message over a communication line which may be tapped by an adversary.

The traditional solution to this problem is called *private key encryption*. In private key encryption A and B hold a meeting before the remote transmission takes place and agree on a pair of encryption and decryption algorithms $\mathcal{E}$ and $\mathcal{D}$, and an additional piece of information S to be kept secret. We shall refer to S as the *common secret key*. The adversary may know the encryption and decryption algorithms $\mathcal{E}$ and $\mathcal{D}$ which are being used, but does not know S.

After the initial meeting when A wants to send B the *cleartext* or *plaintext* message m over the insecure communication line, A *encrypts* m by computing the *ciphertext* $c = \mathcal{E}(S, m)$ and sends c to B. Upon receipt, B *decrypts* c by computing $m = \mathcal{D}(S, c)$. The line-tapper (or adversary), who does not know S, should not be able to compute m from c.

Let us illustrate this general and informal setup with an example familiar to most of us from childhood, the *substitution cipher*. In this method A and B meet and agree on some secret permutation $f\colon \Sigma \to \Sigma$ (where Σ is the alphabet of the messages to be sent). To encrypt message $m = m_1 \ldots m_n$ where $m_i \in \Sigma$, A computes $\mathcal{E}(f, m) = f(m_1) \ldots f(m_n)$. To decrypt $c = c_1 \ldots c_n$ where $c_i \in \Sigma$, B computes $\mathcal{D}(f, c) = f^{-1}(c_1) \ldots f^{-1}(c_n) = m_1 \ldots m_n = m$. In this example the common secret key is the permutation f. The encryption and decryption algorithms $\mathcal{E}$ and $\mathcal{D}$ are as specified, and are known to the adversary. We note that the substitution cipher is easy to break by an adversary who sees a moderate (as a function of the size of the alphabet Σ) number of ciphertexts.

A rigorous theory of perfect secrecy based on information theory was developed by Shannon [180] in 1943. [1]. In this theory, the adversary is assumed to have unlimited computational resources. Shannon showed

[1] Shannon's famous work on information theory was an outgrowth of his work on security ([181]).

that secure (properly defined) encryption system can exist only if the size of the secret information S that A and B agree on prior to remote transmission is as large as the number of secret bits to be ever exchanged remotely using the encryption system.

An example of a private key encryption method which is secure even in presence of a computationally unbounded adversary is the *one time pad*. A and B agree on a secret bit string $pad = b_1 b_2 \ldots b_n$, where $b_i \in_R \{0,1\}$ (i.e pad is chosen in $\{0,1\}^n$ with uniform probability). This is the common secret key. To encrypt a message $m = m_1 m_2 \ldots m_n$ where $m_i \in \{0,1\}$, A computes $\mathcal{E}(pad, m) = m \oplus pad$ (bitwise exclusive or). To decrypt ciphertext $c \in \{0,1\}^n$, B computes $\mathcal{D}(pad, c) = pad \oplus c = pad \oplus (m \oplus pad) = m$. It is easy to verify that $\forall m, c$ the $\mathbf{P}_{pad}\left[\mathcal{E}(pad, m) = c\right] = \frac{1}{2^n}$. From this, it can be argued that seeing c gives "no information" about what has been sent. (In the sense that the adversary's a posteriori probability of predicting m given c is no better than her a priori probability of predicting m without being given c.)

Now, suppose A wants to send B an additional message m'. If A were to simply send $c = \mathcal{E}(pad, m')$, then the sum of the lengths of messages m and m' will exceed the length of the secret key pad, and thus by Shannon's theory the system cannot be secure. Indeed, the adversary can compute $\mathcal{E}(pad, m) \oplus \mathcal{E}(pad, m') = m \oplus m'$ which gives information about m and m' (e.g. can tell which bits of m and m' are equal and which are different). To fix this, the length of the pad agreed upon a-priori should be the sum total of the length of all messages ever to be exchanged over the insecure communication line.

1.2 Modern Encryption: A Computational Complexity Based Theory

Modern cryptography abandons the assumption that the Adversary has available infinite computing resources, and assumes instead that the adversary's computation is resource bounded in some reasonable way. In particular, in these notes we will assume that the adversary is a probabilistic algorithm who runs in polynomial time. Similarly, the encryption and decryption algorithms designed are probabilistic and run in polynomial time.

The running time of the encryption, decryption, and the adversary algorithms are all measured as a function of a *security parameter* k which is a parameter which is fixed at the time the cryptosystem is setup. Thus, when we say that the adversary algorithm runs in polynomial time, we mean time bounded by some polynomial function in k.

Accordingly, in modern cryptography, we speak of the *infeasibility* of breaking the encryption system and computing information about exchanged messages where as historically one spoke of the *impossibility* of breaking the encryption system and finding information about exchanged messages. We note that the encryption systems which we will describe and claim "secure" with respect to the new adversary are not "secure" with respect to a computationally unbounded adversary in the way that the one-time pad system was secure against an unbounded adversary. But, on the other hand, it is no longer necessarily true that the size of the secret key that A and B meet and agree on before remote transmission must be as long as the total number of secret bits ever to be exchanged securely remotely. In fact, at the time of the initial meeting, A and B do not need to know in advance how many secret bits they intend to send in the future.

We will show how to construct such encryption systems, for which the number of messages to be exchanged securely can be a polynomial in the length of the common secret key. How we construct them brings us to anther fundamental issue, namely that of cryptographic, or complexity, assumptions.

As modern cryptography is based on a gap between efficient algorithms for encryption for the legitimate users versus the computational infeasibility of decryption for the adversary, it requires that one have available *primitives* with certain special kinds of computational hardness properties. Of these, perhaps the most basic is a *one-way function*. Informally, a function is one-way if it is easy to compute but hard to invert. Other primitives include *pseudo-random number generators*, and *pseudorandom function families*, which we will define and discuss later. From such primitives, it is possible to build secure encryption schemes.

Thus, a central issue is where these primitives come from. Although one-way functions are widely believed to exist, and there are several conjectured candidate one-way functions which are widely used, we currently do not know how to mathematically prove that they actually exist. We shall thus design cryptographic schemes assuming we are given a one-way function. We will use the conjectured candidate one-way functions for our working examples, throughout our notes. We will be explicit about what exactly can and cannot be proved and is thus assumed, attempting to keep the latter to a bare minimum.

We shall elaborate on various constructions of private-key encryption algorithms later in the course.

The development of *public key cryptography* in the seventies enables one to drop the requirement that A and B must share a key in order to encrypt. The receiver B can publish authenticated[2] information (called the *public-key*) for anyone including the adversary, the sender A, and any other sender to read at their convenience (e.g in a phone book). We will show encryption algorithms in which whoever can read the public key can send encrypted messages to B without ever having met B in person. The encryption system is no longer intended to be used by a pair of prespecified users, but by many senders wishing to send secret messages to a single recipient. The receiver keeps secret (to himself alone!) information (called the receiver's *private key*) about the public-key, which enables him to decrypt the cyphertexts he receives. We call such an encryption method *public key encryption*.

We will show that secure public key encryption is possible given a *trapdoor function*. Informally, a trapdoor function is a one-way function for which there exists some *trapdoor* information known to the receiver alone, with which the receiver can invert the function. The idea of public-key cryptosystems and trapdoor functions was introduced in the seminal work of Diffie and Hellman in 1976 [65, 66]. Soon after the first implementations of their idea were proposed in [164], [157], [132].

A simple construction of public key encryption from trapdoor functions goes as follows. Recipient B can choose at random a trapdoor function f and its associated trapdoor information t, and set its public key to be a description of f and its private key to be t. If A wants to send message m to B, A computes $\mathcal{E}(f,m) = f(m)$. To decrypt $c = f(m)$, B computes $f^{-1}(c) = f^{-1}(f(m)) = m$. We will show that this construction is not secure enough in general, but construct probabilistic variants of it which are secure.

1.3 A Short List of Candidate One Way Functions

As we said above, the most basic primitive for cryptographic applications is a one-way function which is "easy" to compute but "hard" to invert. (For public key encryption, it must also have a trapdoor.) By "easy", we mean that the function can be computed by a probabilistic polynomial time algorithm, and by "hard" that any probabilistic polynomial time (PPT) algorithm attempting to invert it will succeed with "small" probability (where the probability ranges over the elements in the domain of the function.) Thus, to qualify as a potential candidate for a one-way function, the hardness of inverting the function should not hold only on rare inputs to the function but with high probability over the inputs.

Several candidates which seem to posses the above properties have been proposed.

1. *Factoring.* The function $f : (x, y) \mapsto xy$ is conjectured to be a one way function. The asymptotically proven fastest factoring algorithms to date are variations on Dixon's random squares algorithm [122]. It is a randomized algorithm with running time $L(n)^{\sqrt{2}}$ where $L(n) = e^{\sqrt{\log n \log\log n}}$. The number field sieve by Lenstra, Lenstra, Manasee, and Pollard with modifications by Adlemann and Pomerance is a factoring algorithm proved under a certain set of assumptions to factor integers in expected time
$$e^{((c+o(1))(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}})}$$
[124, 3].
2. *The discrete log problem.* Let p be a prime. The multiplicative group $\mathcal{Z}_p^* = (\{x < p | (x, p) = 1\}, \cdot \bmod p)$ is cyclic, so that $\mathcal{Z}_p^* = \{g^i \bmod p | 1 \le i \le p-1\}$ for some generator $g \in \mathcal{Z}_p^*$. The function $f : (p, g, x) \mapsto$

[2] Saying that the information is "authenticated" means that the sender is given a guarantee that the information was published by the legal receiver. How this can be done is discussed in a later chapter.

$(g^x \bmod p, p, g)$ where p is a prime and g is a generator for Z_p^* is conjectured to be a one-way function. Computing $f(p, g, x)$ can be done in polynomial time using repeated squaring. However, The fastest known proved solution for its inverse, called the discrete log problem is the index-calculus algorithm, with expected running time $L(p)^{\sqrt{2}}$ (see [122]). An interesting problem is to find an algorithm which will generate a prime p and a generator g for Z_p^*. It is not known how to find generators in polynomial time. However, in [8], E. Bach shows how to generate random factored integers (in a given range $\frac{N}{2} \ldots N$). Coupled with a fast primality tester (as found in [122], for example), this can be used to efficiently generate random tuples $(p-1, q_1, \ldots, q_k)$ with p prime. Then picking $g \in \mathcal{Z}_p^*$ at random, it can be checked if $(g, p-1) = 1$, $\forall q_i$, $g^{\frac{p-1}{q_i}} \bmod p \neq 1$, and $g^{p-1} \bmod p = 1$, in which case $order(g) = p-1$ ($order(g) = |\{g^i \bmod p | 1 \leq i \leq p-1\}|$). It can be shown that the density of $\mathcal{Z}_p^*$ generators is high so that few guesses are required. The problem of efficiently finding a generator for a specific $\mathcal{Z}_p^*$ is an intriguing open research problem.

3. *Subset sum.* Let $a_i \in \{0,1\}^n, \vec{a} = (a_1, \ldots, a_n), s_i \in \{0,1\}, \vec{s} = (s_1, \ldots, s_n)$, and let $f : (\vec{a}, \vec{s}) \mapsto (\vec{a}, \sum_{i=1}^n s_i a_i)$. An inverse of $(\vec{a}, \sum_{i=1}^n s_i a_i)$ under f is any $(\vec{a}, \vec{s_i}')$ so that $\sum_{i=1}^n s_i a_i = \sum_{i=1}^n s_i' a_i$. This function f is a candidate for a one way function. The associated decision problem (given $(\vec{a}, y)$, does there exists $\vec{s}$ so that $\sum_{i=1}^n s_i a_i = y$?) is NP-complete. Of course, the fact that the subset-sum problem is NP-complete cannot serve as evidence to the one-wayness of f_{ss}. On the other hand, the fact that the subset-sum problem is easy for special cases (such as "hidden structure" and low density) can not serve as evidence for the weakness of this proposal. The conjecture that f is one-way is based on the failure of known algorithm to handle random high density instances. Yet, one has to admit that the evidence in favor of this candidate is much weaker than the evidence in favor of the two previous ones.

4. *DES with fixed message.* Fix a 64 bit message M and define the function $f(K) = \mathrm{DES}_K(M)$ which takes a 56 bit key K to a 64 bit output $f(K)$. This appears to be a one-way function. Indeed, this construction can even be proven to be one-way assuming DES is a family of pseudorandom functions, as shown by Luby and Rackoff [129].

5. *RSA.* This is a candidate one-way trapdoor function. Let $N = pq$ be a product of two primes. It is believed that such an N is hard to factor. The function is $f(x) = x^e \bmod N$ where e is relatively prime to $(p-1)(q-1)$. The trapdoor is the primes p, q, knowledge of which allows one to invert f efficiently. The function f seems to be one-way. To date the best attack is to try to factor N, which seems computationally infeasible.

In Chapter 2 we discuss formal definitions of one-way functions and are more precise about the above constructions.

1.4 Security Definitions

So far we have used the terms "secure" and "break the system" quite loosely. What do we really mean? It is clear that a minimal requirement of security would be that: any adversary who can see the ciphertext and knows which encryption and decryption algorithms are being used, can not recover the entire cleartext. But, many more properties may be desirable. To name a few:

1. It should be hard to recover the messages from the ciphertext when the messages are drawn from arbitrary probability distributions defined on the set of all strings (i.e arbitrary *message spaces*). A few examples of message spaces are: the English language, the set $\{0,1\}$). We must assume that the message space is known to the adversary.

2. It should be hard to compute partial information about messages from the ciphertext.

3. It should be hard to detect simple but useful facts about traffic of messages, such as when the same message is sent twice.

4. The above properties should hold with high probability.

In short, it would be desirable for the encryption scheme to be the mathematical analogy of opaque envelopes containing a piece of paper on which the message is written. The envelopes should be such that all legal senders can fill it, but only the legal recipient can open it.

We must answer a few questions:

- How can "opaque envelopes" be captured in a precise mathematical definition? Much of Chapters 6 and 7 is dedicated to discussing the precise definition of security in presence of a computationally bounded adversary.
- Are "opaque envelopes" achievable mathematically? The answer is positive . We will describe the the proposals of private (and public) encryption schemes which we prove secure under various assumptions.

We note that the simple example of a public-key encryptions system based on trapdoor function, described in the previous section, does not satisfy the above properties. We will show later, however, probabilistic variants of the simple system which do satisfy the new security requirements under the assumption that trapdoor functions exist. More specifically, we will show probabilistic variants of RSA which satisfy the new security requirement under, the assumption that the original RSA function is a trapdoor function, and are similar in efficiency to the original RSA public-key encryption proposal.

1.5 The Model of Adversary

The entire discussion so far has essentially assumed that the adversary can listen to cyphertexts being exchanged over the insecure channel, read the public-file (in the case of public-key cryptography), generate encryptions of any message on his own (for the case of public-key encryption), and perform probabilistic polynomial time computation. This is called a *passive adversary*.

One may imagine a more powerful adversary who can intercept messages being transmitted from sender to receiver and either stop their delivery all together or alter them in some way. Even worse, suppose the adversary can request a polynomial number of cyphertexts to be decrypted for him. We can still ask whether there exists encryption schemes (public or secret) which are secure against such more powerful adversaries.

Indeed, such adversaries have been considered and encryption schemes which are secure against them designed. The definition of security against such adversaries is more elaborate than for passive adversaries.

In Chapters 6 and 7 we consider a passive adversary who knows the probability distribution over the message space. We will also discuss more powerful adversaries and appropriate definitions of security.

1.6 Road map to Encryption

To summarize the introduction, our challenge is to design both secure private-key and public-key encryption systems which provably meet our definition of security and in which the operations of encryption and decryption are as fast as possible for the sender and receiver.

Chapters 6 and 7 embark on an in depth investigation of the topic of encryption, consisting of the following parts. For both private-key and public-key encryption, we will:

- Discuss formally how to define security in presence of a bounded adversary.
- Discuss current proposals of encryption systems and evaluate them respect to the security definition chosen.
- Describe how to design encryption systems which we can prove secure under explicit assumptions such as the existence of one-way functions, trapdoor functions, or pseudo random functions.

- Discuss efficiency aspects of encryption proposals, pointing out to possible ways to improve efficiency by performing some computations off-line, in batch mode, or in a incremental fashion.

We will also overview some advanced topics connected to encryption such chosen-ciphertext security, non-malleability, key-escrow proposals, and the idea of shared decryption among many users of a network.

Chapter 2

One-way and trapdoor functions

One Way functions, namely functions that are "easy" to compute and "hard" to invert, are an extremely important cryptographic primitive. Probably the best known and simplest use of one-way functions, is for passwords. Namely, in a time-shared computer system, instead of storing a table of login passwords, one can store, for each password w, the value $f(w)$. Passwords can easily be checked for correctness at login, but even the system administrator can not deduce any user's password by examining the stored table.

In Section 1.3 we had provided a short list of some candidate one-way functions. We now develop a theoretical treatment of the subject of one-way and trapdoor functions, and carefully examine the candidate one-way functions proposed in the literature. We will occasionaly refer to facts about number theory discussed in Chapter C.

We begin by explaining why one-way functions are of fundamental importance to cryptography.

2.1 One-Way Functions: Motivation

In this section, we provide motivation to the definition of one-way functions. We argue that the existence of one-way functions is a necessary condition to the existence of most known cryptographic primitives (including secure encryption and digital signatures). As the current state of knowledge in complexity theory does not allow to prove the existence of one-way function, even using more traditional assumptions as $\mathcal{P} \neq \mathcal{NP}$, we will have to assume the existence of one-way functions. We will later try to provide evidence to the plausibility of this assumption.

As stated in the introduction chapter, modern cryptography is based on a gap between efficient algorithms guaranteed for the legitimate user versus the unfeasibility of retrieving protected information for an adversary. To make the following discussion more clear, let us concentrate on the cryptographic task of secure data communication, namely encryption schemes.

In secure encryption schemes, the legitimate user is able to decipher the messages (using some private information available to him), yet for an adversary (not having this private information) the task of decrypting the ciphertext (i.e., "breaking" the encryption) should be infeasible. Clearly, the breaking task can be performed by a non-deterministic polynomial-time machine. Yet, the security requirement states that breaking should not be feasible, namely could not be performed by a probabilistic polynomial-time machine. Hence, the existence of secure encryption schemes implies that there are tasks performed by non-deterministic polynomial-time machines yet cannot be performed by deterministic (or even randomized) polynomial-time machines. In other words, a necessary condition for the existence of secure encryption schemes is that $\mathcal{NP}$ is not contained in $\mathcal{BPP}$ (and hence that $\mathcal{P} \neq \mathcal{NP}$).

However, the above mentioned necessary condition (e.g., $\mathcal{P} \neq \mathcal{NP}$) is not a sufficient one. $\mathcal{P} \neq \mathcal{NP}$ only implies that the encryption scheme is hard to break in the worst case. It does not rule-out the possibility that the encryption scheme is easy to break in almost all cases. In fact, one can easily construct "encryption schemes" for which the breaking problem is NP-complete and yet there exist an efficient breaking algorithm that succeeds on 99% of the cases. Hence, worst-case hardness is a poor measure of security. Security requires hardness on most cases or at least average-case hardness. Hence, a necessary condition for the existence of secure encryption schemes is the existence of languages in $\mathcal{NP}$ which are hard on the average. Furthermore, $\mathcal{P} \neq \mathcal{NP}$ is not known to imply the existence of languages in $\mathcal{NP}$ which are hard on the average.

The mere existence of problems (in NP) which are hard on the average does not suffice. In order to be able to use such problems we must be able to generate such hard instances together with auxiliary information which enable to solve these instances fast. Otherwise, the hard instances will be hard also for the legitimate users and they gain no computational advantage over the adversary. Hence, the existence of secure encryption schemes implies the existence of an efficient way (i.e. probabilistic polynomial-time algorithm) of generating instances with corresponding auxiliary input so that

(1) it is easy to solve these instances given the auxiliary input; and

(2) it is hard on the average to solve these instances (when not given the auxiliary input).

We avoid formulating the above "definition". We only remark that the coin tosses used in order to generate the instance provide sufficient information to allow to efficiently solve the instance (as in item (1) above). Hence, without loss of generality one can replace condition (2) by requiring that these coin tosses are hard to retrieve from the instance. The last simplification of the above conditions essentially leads to the definition of a one-way function.

2.2 One-Way Functions: Definitions

In this section, we present several definitions of one-way functions. The first version, hereafter referred to as strong one-way function (or just one-way function), is the most convenient one. We also present weak one-way functions which may be easier to find and yet can be used to construct strong one way functios, and non-uniform one-way functions.

2.2.1 (Strong) One Way Functions

The most basic primitive for cryptographic applications is a one-way function. Informally, this is a function which is "easy" to compute but "hard" to invert. Namely, any probabilistic polynomial time (PPT) algorithm attempting to invert the one-way function on a element in its range, will succeed with no more than "negligible" probability, where the probability is taken over the elements in the domain of the function and the coin tosses of the PPT attempting the inversion.

This informal definition introduces a couple of measures that are prevalent in complexity theoretic cryptography. An easy computation is one which can be carried out by a PPT algorithm; and a function $\nu\colon \mathsf{N} \to \mathbf{R}$ is negligible if it vanishes faster than the inverse of any polynomial. More formally,

Definition 2.2.1 ν is negligible if for every constant $c \geq 0$ there exists an integer k_c such that $\nu(k) < k^{-c}$ for all $k \geq k_c$.

Another way to think of it is $\nu(k) = k^{-\omega(1)}$.

A few words, concerning the notion of negligible probability, are in place. The above definition and discussion considers the success probability of an algorithm to be *negligible* if as a function of the input length the success probability is bounded by any polynomial fraction. It follows that repeating the algorithm polynomially (in the input length) many times yields a new algorithm that also has a negligible success probability. In other words, events which occur with negligible (in n) probability remain negligible even if the experiment is repeated for polynomially (in k) many times. Hence, defining negligible success as "occurring with probability smaller than any polynomial fraction" is naturally coupled with defining feasible as "computed within polynomial time". A "strong negation" of the notion of a negligible fraction/probability is the notion of a non-negligible fraction/probability. we say that a function ν is *non-negligible* if there exists a polynomial p such that for all sufficiently large k's it holds that $\nu(k) > \frac{1}{p(k)}$. Note that functions may be neither negligible nor non-negligible.

Definition 2.2.2 A function f: $\{0,1\}^* \rightarrow \{0,1\}^*$ is *one-way* if:

(1) there exists a PPT that on input x output $f(x)$;

(2) For every PPT algorithm A there is a negligible function ν_A such that for sufficiently large k,

$$\mathbf{P}\left[\, f(z) = y \;:\; x \stackrel{R}{\leftarrow} \{0,1\}^k \;;\; y \leftarrow f(x) \;;\; z \leftarrow A(1^k, y) \,\right] \;\leq\; \nu_A(k)$$

Remark 2.2.3 The guarantee is probabilistic. The adversary is not unable to invert the function, but has a low probability of doing so where the probability distribution is taken over the input x to the one-way function where x if of length k, and the possible coin tosses of the adversary. Namely, x is chosen at random and y is set to $f(x)$.

Remark 2.2.4 The advsersary is not asked to find x; that would be pretty near impossible. It is asked to find some inverse of y. Naturally, if the function is 1-1 then the only inverse is x.

Remark 2.2.5 Note that the adversary algorithm takes as input $f(x)$ and the security parameter 1^k (expressed in unary notatin) which corresponds to the binary length of x. This represents the fact the adversary can work in time polynomial in $|x|$, even if $f(x)$ happends to be much shorter. This rules out the possibility that a function is considered one-way merely because the inverting algorithm does not have enough time to print the output. Consider for example the function defined as $f(x) = y$ where y is the $\log k$ least significant bits of x where $|x| = k$. Since the $|f(x)| = \log |x|$ no algorithm can invert f in time polynomial in $|f(x)|$, yet there exists an obvious algorithm which finds an inverse of $f(x)$ in time polynomial in $|x|$. Note that in the special case of length preserving functions f (i.e., $|f(x)| = |x|$ for all x's), the auxiliary input is redundant.

Remark 2.2.6 By this definition it trivially follows that the size of the output of f is bounded by a polynomial in k, since $f(x)$ is a poly-time computable.

Remark 2.2.7 The definition which is typical to definitions from computational complexity theory, works with asymptotic complexity—what happens as the size of the problem becomes large. Security is only asked to hold for large enough input lengths, namely as k goes to infinity. Per this definition, it may be entirely feasible to invert f on, say, 512 bit inputs. Thus such definitions are less directly relevant to practice, but useful for studying things on a basic level. To apply this definition to practice in cryptography we must typically envisage not a single one-way function but a family of them, parameterized by a *security parameter* k. That is, for each value of the security parameter k there is be a specific function $f : \{0,1\}^k \rightarrow \{0,1\}^*$. Or, there may be a family of functions (or cryptosystems) for each value of k. We shall define such familes in subsequent section.

The next two sections discuss variants of the strong one-way function definition. The first time reader is encouraged to directly go to Section 2.2.4.

2.2.2 Weak One-Way Functions

One way functions come in two flavors: strong and weak. The definition we gave above, refers to a strong way function. We could weaken it by replacing the second requirement in the definition of the function by a weaker requirement as follows.

Definition 2.2.8 A function $f\colon \{0,1\}^* \to \{0,1\}^*$ is *weak one-way* if:

(1) there exists a PPT that on input x output $f(x)$;

(2) There is a polynomial functions Q such that for every PPT algorithm A, and for sufficiently large k,

$$\mathbf{P}\left[\, f(z) \neq y \;:\; x \stackrel{R}{\leftarrow} \{0,1\}^k \;;\; y \leftarrow f(x) \;;\; z \leftarrow A(1^k, y) \,\right] \geq \frac{1}{Q(k)}$$

The difference between the two definitions is that whereas we only require some non-negligible fraction of the inputs on which it is hard to invert a weak one-way function, a strong one-way function must be hard to invert on all but a negligible fraction of the inputs. Clearly, the latter is preferable, but what if only weak one-way functions exist ? Our first theorem is that the existence of a weak one way function implies the existence of a strong one way function. Moreover, we show how to construct a strong one-way function from a weak one. This is important in practice as illustarted by the following example.

Example 2.2.9 Consider for example the function $f : \mathbf{Z} \times \mathbf{Z} \mapsto \mathbf{Z}$ where $f(x,y) = x \cdot y$. This function can be easily inverted on at least half of its outputs (namely, on the even integers) and thus is not a strong one way function. Still, we said in the first lecture that f is hard to invert when x and y are primes of roughly the same length which is the case for a polynomial fraction of the k-bit composite integers. This motivated the definition of a weak one way function. Since the probability that an k-bit integer x is prime is approximately $1/k$, we get the probability that both x and y such that $|x| = |y| = k$ are prime is approximately $1/k^2$. Thus, for all k, about $1 - \frac{1}{k^2}$ of the inputs to f of length $2k$ are prime pairs of equal length. It is believed that no adversary can invert f when x and y are primes of the same length with non-negligible success probability, and under this belief, f is a weak one way function (as condition 2 in the above definition is satisfied for $Q(k) = O(k^2)$).

Theorem 2.2.10 *Weak one way functions exist if and only if strong one way functions exist.*

Proof Sketch: By definition, a strong one way function is a weak one way function. Now assume that f is a weak one way function such that Q is the polynomial in condition 2 in the definition of a weak one way function. Define the function

$$f_1(x_1 \ldots x_N) = f(x_1) \ldots f(x_N)$$

where $N = 2kQ(k)$ and each x_i is of length k.

We claim that f_1 is a strong one way function. Since f_1 is a concatenation of N copies of the function f, to correctly invert f_1, we need to invert $f(x_i)$ correctly for each i. We know that every adversary has a probability of at least $\frac{1}{Q(k)}$ to fail to invert $f(x)$ (where the probability is taken over $x \in \{0,1\}^k$ and the coin tosses of the adversary), and so intuitively, to invert f_1 we need to invert $O(kQ(k))$ instances of f. The probability that the adversary will fail for at least one of these instances is extremely high.

The formal proof (which is omitted here and will be given in appendix) will take the form of a reduction; that is, we will assume for contradiction that f_1 is not a strong one way function and that there exists some adversary A_1 that violates condition 2 in the definition of a strong one way function. We will then show that A_1 can be used as a subroutine by a new adversary A that will be able to invert the original function f with probability better than $1 - \frac{1}{Q(|x|)}$ (where the probability is taken over the inputs $x \in \{0,1\}^k$ and the coin tosses of A). But this will mean that f is not a weak one way function and we have derived a contradiction. ∎

This proof technique is quite typical of proofs presented in this course. Whenever such a proof is presented it is important to examine the cost of the reduction. For example, the construction we have just outlined is not length preserving, but expands the size of the input to the function quadratically.

2.2.3 Non-Uniform One-Way Functions

In the above two definitions of one-way functions the inverting algorithm is probabilistic polynomial-time. Stronger versions of both definitions require that the functions cannot be inverted even by non-uniform families of polynomial size algorithm We stress that the "easy to compute" condition is still stated in terms of uniform algorithms. For example, following is a non-uniform version of the definition of (strong) one-way functions.

Definition 2.2.11 A function f is called *non-uniformly strong one-way* if the following two conditions hold

(1) *easy to compute:* as before There exists a PPT algorithm to compute for f.

(2) *hard to invert:* For every (even non-uniform) family of polynomial-size algorithms $A = \{M_k\}_{k\in\mathsf{N}}$, there exists a negligible ν_A such that for all sufficiently large k's

$$\mathbf{P}\left[\; f(z) \neq y \;:\; x \stackrel{R}{\leftarrow} \{0,1\}^k \;;\; y \leftarrow f(x) \;;\; z \leftarrow M_k(y) \right] \;\leq\; \nu_A(k)$$

Note that it is redundent to give 1^k as an auxiliary input to M_k.

It can be shown that if f is non-uniformly one-way then it is (strongly) one-way (i.e., in the uniform sense). The proof follows by converting any (uniform) probabilistic polynomial-time inverting algorithm into a non-uniform family of polynomial-size algorithm, without decreasing the success probability. Details follow. Let A' be a probabilistic polynomial-time (inverting) algorithm. Let r_k denote a sequence of coin tosses for A' maximizing the success probability of A'. The desired algorithm M_k incorporates the code of algorithm A' and the sequence r_k (which is of length polynomial in k).

It is possible, yet not very plausible, that strongly one-way functions exist and but there are no non-uniformly one-way functions.

2.2.4 Collections Of One Way Functions

Instead of talking about a single function $f : \{0,1\}^* \to \{0,1\}^*$, it is often convenient to talk about collections of functions, each defined over some finite domain and finite ranges. We remark, however, that the single function format makes it easier to prove properties about one way functions.

Definition 2.2.12 Let I be a set of indices and for $i \in I$ let D_i and R_i be finite. A collection of strong one way functions is a set $F = \{f_i : D_i \to R_i\}_{i\in I}$ satisfying the following conditions.

(1) There exists a PPT S_1 which on input 1^k outputs an $i \in \{0,1\}^k \cap I$

(2) There exists a PPT S_2 which on input $i \in I$ outputs $x \in D_i$

(3) There exists a PPT A_1 such that for $i \in I$ and $x \in D_i$, $A_1(i,x) = f_i(x)$.

(4) For every PPT A there exists a negligible ν_A such that $\forall$ k large enough

$$\mathbf{P}\left[\; f_i(z) \neq y \;:\; i \stackrel{R}{\leftarrow} I \;;\; x \stackrel{R}{\leftarrow} D_i \;;\; y \leftarrow f_i(x) \;;\; z \leftarrow A(i,y) \right] \;\leq\; \nu_A(k)$$

(here the probability is taken over choices of i and x, and the coin tosses of A).

In general, we can show that the existence of a single one way function is equivalent to the existence of a collection of one way functions. We prove this next.

Theorem 2.2.13 *A collection of one way functions exists if and only if one way functions exist.*

Proof: Suppose that f is a one way function.

Set $F = \{f_i : D_i \to R_i\}_{i \in I}$ where $I = \{0,1\}^*$ and for $i \in I$, take $D_i = R_i = \{0,1\}^{|i|}$ and $f_i(x) = f(x)$. Furthermore, S_1 uniformly chooses on input 1^k, $i \in \{0,1\}^k$, S_2 uniformly chooses on input i, $x \in D_i = \{0,1\}^{|i|}$ and $A_1(i,x) = f_i(x) = f(x)$. (Note that f is polynomial time computable.) Condition 4 in the definition of a collection of one way functions clearly follows from the similar condition for f to be a one way function.

Now suppose that $F = \{f_i : D_i \to R_i\}_{i \in I}$ is a collection of one way functions. Define $f_F(1^k, r_1, r_2) = A_1(S_1(1^k, r_1), S_2(S_1(1^k, r_1), r_2))$ where A_1, S_1, and S_2 are the functions associated with F as defined in Definition 2.2.12. In other words, f_F takes as input a string $1^k \circ r_1 \circ r_2$ where r_1 and r_2 will be the coin tosses of S_1 and S_2, respectively, and then

- Runs S_1 on input 1^k using the coin tosses r_1 to get the index $i = S_1(1^k, r_1)$ of a function $f_i \in F$.
- Runs S_2 on the output i of S_1 using the coin tosses r_2 to find an input $x = S_2(i, r_2)$.
- Runs A_1 on i and x to compute $f_F(1^k, r_1, r_2) = A_1(i, x) = f_i(x)$.

Note that randomization has been restricted to the input of f_F and since A_1 is computable in polynomial time, the conditions of a one way function are clearly met. ∎

A possible example is the following, treated thoroughly in Section 2.3.

Example 2.2.14 The hardness of computing discrete logarithms yields the following collection of functions. Define $EXP = \{EXP_{p,g}(i) = g^i \bmod p,\ EXPp, g : Z_p \to Z_p^*\}_{<p,g> \in I}$ for $I = \{< p, g > p$ prime, g generator for $Z_p^*\}$.

2.2.5 Trapdoor Functions and Collections

Infromally, a *trapdoor function* f is a one-way function with an extra property. There also exists a secret inverse function (the*trapdoor*) that allows its possessor to efficiently invert f at any point in the domain of his choosing. It should be easy to compute f on any point, but infeasible to invert f on any point without knowledge of the inverse function . Moreover, it should be easy to generate matched pairs of f's and corresponding trapdoor. Once a matched pair is generated, the publication of f should not reveal anything about how to compute its inverse on any point.

Definition 2.2.15 A *trapdoor* function is a one-way function $f : \{0,1\}^* \to \{0,1\}^*$ such that there exists a polynomial p and a probabilistic polynomial time algorithm I such that for every k there exists an $t_k \in \{0,1\}^*$ such that $|t_k| \leq p(k)$ and for all $x \in \{0,1\}^*$, $I(f(x), t_k) = y$ such that $f(y) = f(x)$.

An example of a function which may be trapdoor if factoring integers is hard was proposed by Rabin[157]. Let $f(x,n) = x^2 \bmod n$ where $n = pq$ a product of two primes and $x \in Z_n^*$. Rabin[157] has shown that inverting f is easy iff factoring composite numbers product of two primes is easy. The most famous candidate trapdoor function is the RSA[164] function $f(x,n,l) = x^l \bmod n$ where $(l, \phi(n)) = 1$.

Again it will be more convenient to speak of families of trapdoor functions parameterized by security parameter k.

Definition 2.2.16 Let I be a set of indices and for $i \in I$ let D_i be finite. A collection of strong one way trapdoor functions is a set $F = \{f_i : D_i \to D_i\}_{i \in I}$ satisfying the following conditions.

(1) There exists a polynomial p and a PTM S_1 which on input 1^k outputs pairs (i, t_i) where $i \in I \cap \{0,1\}^k$ and $|t_i| < p(k)$ The information t_i is referred to as the trapdoor of i.

(2) There exists a PTM S_2 which on input $i \in I$ outputs $x \in D_i$

(3) There exists a PTM A_1 such that for $i \in I$, $x \in D_i$ $A_1(i,x) = f_i(x)$.

(4) There exists a PTM A_2 such that $A_2(i, t_i, f_i(x)) = x$ for all $x \in D_i$ and for all $i \in I$ (that is, f_i is easy to invert when t_i is known).

(5) For every PPT A there exists a negligble ν_A such that $\forall$ k large enough

$$\mathbf{P}\left[\, f_i(z) \neq y \;:\; i \stackrel{R}{\leftarrow} I \;;\; x \stackrel{R}{\leftarrow} D_i \;;\; y \leftarrow f_i(x) \;;\; z \leftarrow A(i,y) \,\right] \;\leq\; \nu_A(k)$$

A possible example is the following treated in in detail in the next sections.

Example 2.2.17 [The RSA collections of possible trapdoor functions] Let p, q denote primes, $n = pq$, $Z_n^* = \{1 \leq x \leq n, (x,n) = 1\}$ the multiplicative group whose cardinality is $\varphi(n) = (p-1)(q-1)$, and $e \in Z_{p-1}$ relatively prime to $\varphi(n)$. Our set of indices will be $I = \{< n, e > \text{ such that } n = pq \; |p| = |q|\}$ and the trapdoor associated with the particular index $< n, e >$ be d such that $ed = 1 \bmod \phi(n)$. Let $RSA = \{RSA_{<n,e>} : Z_n^* \to Z_n^*\}_{<n,e> \in I}$ where $RSA_{<n,e>}(x) = x^e \bmod n$

2.3 In Search of Examples

Number theory provides a source of candidates for one way and trapdoor functions. Let us start our search for examples by a digression into number theorey. See also the mini-course on number theory in Appendix C.

Calculating Inverses in Z_p^*

Consider the set $\mathbf{Z}_p^* = \{x : 1 \leq x < p \text{ and } \gcd(x,p) = 1\}$ where p is prime. $\mathbf{Z}_p^*$ is a group under multiplicaton modulo p. Note that to find the inverse of $x \in \mathbf{Z}_p^*$; that is, an element $y \in \mathbf{Z}_p^*$ such that $yx \equiv 1 \bmod p$, we can use the Euclidean algorithm to find integers y and z such that $yx + zp = 1 = \gcd(x,p)$. Then, it follows that $yx \equiv 1 \bmod p$ and so $y \bmod p$ is the desired inverse.

The Euler Totient Function $\varphi(n)$

Euler's Totient Function φ is defined by $\varphi(n) = |\{x : 1 \leq x < p \text{ and } \gcd(x,n) = 1\}$. The following are facts about φ.

(1) For p a prime and $\alpha \geq 1$, $\varphi(p^\alpha) = p^{\alpha-1}(p-1)$.

(2) For integers m, n with $\gcd(m,n) = 1$, $\varphi(mn) = \varphi(m)\varphi(n)$.

Using the rules above, we can find φ for any n because, in general,

$$\begin{aligned} \varphi(n) &= \varphi(\prod_{i=1}^{k} {p_i}^{\alpha_i}) \\ &= \prod_{i=1}^{k} \varphi({p_i}^{\alpha_i}) \end{aligned}$$

$$= \prod_{i=1}^{k} {p_i}^{\alpha_i - 1}(p_i - 1)$$

Z_p^* Is Cyclic

A group G is cyclic if and only if there is an element $g \in G$ such that for every $a \in G$, there is an integer i such that $g^i = a$. We call g a *generator* of the group G and we denote the index i by $ind_g(a)$.

Theorem 2.3.1 (Gauss) *If p is prime then $\mathbf{Z}_p^*$ is a cyclic group of order $p-1$. That is, there is an element $g \in \mathbf{Z}_p^*$ such that $g^{p-1} \equiv 1 \bmod p$ and $g^i \not\equiv 1 \bmod p$ for $i < p-1$.*

¿From Theorem 2.3.1 the following fact is immediate.

Fact 2.3.2 Given a prime p, a generator g for $\mathbf{Z}_p^*$, and an element $a \in \mathbf{Z}_p^*$, there is a unique $1 \le i \le p-1$ such that $a = g^i$.

The Legendre Symbol

Fact 2.3.3 If p is a prime and g is a generator of $\mathbf{Z}_p^*$, then

$$g^c = g^a g^b \bmod p \Leftrightarrow c = a + b \bmod p - 1$$

¿From this fact it follows that there is an homomorphism $f : \mathbf{Z}_p^* \to \mathbf{Z}_{p-1}$ such that $f(ab) = f(a) + f(b)$. As a result we can work with $\mathbf{Z}_{p-1}$ rather than $\mathbf{Z}_p^*$ which sometimes simplifies matters. For example, suppose we wish to determine how many elements in $\mathbf{Z}_p^*$ are perfect squares (these elements will be referred to as *quadratic residues* modulo p). The following lemma tells us that the number of quadratic residues modulo p is $\frac{1}{2}|\mathbf{Z}_p^*|$.

Lemma 2.3.4 *$a \in \mathbf{Z}_p^*$ is a quadratic residue modulo p if and only if $a = g^x \bmod p$ where x satisfies $1 \le x \le p-1$ and is even.*

Proof: Let g be a generator in $\mathbf{Z}_p^*$.
($\Leftarrow$) Suppose an element $a = g^{2x}$ for some x. Then $a = s^2$ where $s = g^x$.
($\Rightarrow$) Consider the square of an element $b = g^y$. $b^2 = g^{2y} \equiv g^e \bmod p$ where e is even since $2y$ is reduced modulo $p-1$ which is even. Therefore, only those elements which can be expressed as g^e, for e an even integer, are squares. ▮

Consequently, the number of quadratic residues modulo p is the number of elements in $\mathbf{Z}_p^*$ which are an even power of some given generator g. This number is clearly $\frac{1}{2}|\mathbf{Z}_p^*|$.

The *Legendre Symbol* $\mathbf{J}_p(x)$ specifies whether x is a perfect square in $\mathbf{Z}_p^*$ where p is a prime.

$$\mathbf{J}_p(x) = \begin{cases} 1 & \text{if } x \text{ is a square in } \mathbf{Z}_p^* \\ 0 & \text{if } \gcd(x,p) \neq 1 \\ -1 & \text{if } x \text{ is not a square in } \mathbf{Z}_p^* \end{cases}$$

The Legendre Symbol can be calculated in polynomial time due to the following theorem.

Theorem 2.3.5 *[Euler's Criterion]* $\mathbf{J}_p(x) \equiv x^{\frac{p-1}{2}} \bmod p$.

Using repeated doubling to compute exponentials, one can calculate $x^{\frac{p-1}{2}}$ in $O(|p|^3)$ steps. Though this $\mathbf{J}_p(x)$ can be calculated when p is a prime, it is not known how to determine for general x and n, whether x is a square in $\mathbf{Z}_n^*$.

2.3.1 The Discrete Logarithm Function

Let EXP be the function defined by $\mathrm{EXP}(p, g, x) = (p, g, g^x \bmod p)$. We are particularly interested in the case when p is a prime and g is a generator of Z_p^*. Deine an index set $I = \{(p, g) : p$ is prime and g is a generator of $Z_p^*\}$. For $(p, g) \in I$, it follows by Fact 2.3.2 that $\mathrm{EXP}(p, g, x)$ has a unique inverse and this allows us to define for $y \in \mathbf{Z}_p^*$ the discrete logarithm function DL by $\mathrm{DL}(p, g, y) = (p, g, x)$ where $x \in \mathbf{Z}_{p-1}$ and $g^x \equiv y \bmod p$. Given p and g, $\mathrm{EXP}(p, g, x)$ can easily be computed in polynomial time. However, it is unknown whether or not its inverse DL can be computed in polynomial time unless $p-1$ has very small factors (see [151]). Pohlig and Hellman [151] present effective techniques for this problem when $p-1$ has only small prime factors.

The best fully proved up-to-date algorithm for computing discrete logs is the Index-calculus algorithm. The expected running time of such algorithm is polynomial in $e^{\sqrt{\log k \log\log k}}$ where k is the size of the modulos p. There is a recent variant of the number field sieve algorithm for discrete logarithm which seems to work in faster running time of $e^{(\log k \log\log k)^{\frac{1}{3}}}$. It interesting to note that working over the finite field $GF(2^k)$ rather than working modulo p seems to make the problem substantially easier (see Coppersmith [56] and Odlyzko [145]). Curiously, computing discrete logarithms and factoring integers seem to have essentially the same difficulty at least as indicated by the current state of the art algorithms.

With all this in mind, we consider EXP a good candidate for a one way function. We make the following explicit assumption in this direction. The assumption basically says that there exists no polynomial time algorithm that can solvethe discrete log problem with prime modulos.

Strong Discrete Logarithm Assumption (DLA):[1] For every polynomial Q and every PPT A, for all sufficiently large k,

$$\Pr[A(p, g, y) = x \text{ such that } y \equiv g^x \bmod p \text{ where } 1 \leq x \leq p-1] < \frac{1}{Q(k)}$$

(where the probability is taken over all primes p such that $|p| \leq k$, the generators g of $\mathbf{Z}_p^*$, $x \in \mathbf{Z}_p^*$ and the coin tosses of A).

An immediate consequence of this assumption we get

Theorem 2.3.6 *Under the strong discrete logarithm assumption there exists a strong one way function; namely, exponentiation modulo a prime p.*

Some useful properties of EXP and DL follow.

Remark 2.3.7 If $\mathrm{DL}(p, g_1, y)$ is easy to calculate for some generator $g_1 \in \mathbf{Z}_p^*$ then it is also easy to calculate $\mathrm{DL}(p, g_2, y)$ for any other generator $g_2 \in \mathbf{Z}_p^*$. (The group $\mathbf{Z}_p^*$ has $\varphi(p-1)$ generators.) To see this suppose that $x_1 = \mathrm{DL}(p, g_1, y)$ and $x_2 = \mathrm{DL}(p, g_2, y)$. If $g_2 \equiv {g_1}^z \bmod p$ where $\gcd(z, p-1)$ then $y \equiv {g_1}^{x_2 z} \bmod p$ and consequently, $x_2 \equiv z^{-1} x_1 \bmod p-1$.

The following result shows that to efficiently calculate $\mathrm{DL}(p, g, y)$ for $(p, g) \in I$ it will suffice to find a polynomial time algorithm which can calculate $\mathrm{DL}(p, g, y)$ on at least a $\frac{1}{Q(|p|)}$ fraction of the possible inputs $y \in \mathbf{Z}_p^*$ for some polynomial Q.

[1] We note that a weaker assumption can be made concerning the discrete logarithm problem, and by the standard construction one can still construct a strong one-way function. We will assume for the purpose of the course the first stronger assumption. **Weak Discrete Logarithm Assumption:** There is a polynomial Q such that for every PTM A there exists an integer k_0 such that $\forall k > k_0$ $\Pr[A(p, g, y) = x \text{ such that } y \equiv g^x \bmod p \text{ where } 1 \leq x \leq p-1] < 1 - \frac{1}{Q(k)}$ (where the probability is taken over all primes p such that $|p| \leq k$, the generators g of $\mathbf{Z}_p^*$, $x \in \mathbf{Z}_p^*$ and the coin tosses of A).

Proposition 2.3.8 *Let ϵ, $\delta \in (0,1)$ and let S be a subset of the prime integers. Suppose there is a probabilistic algorithm A such that for all primes $p \in S$ and for all generators g of $\mathbf{Z}_p^*$*

$$\Pr[A(p,g,y) = x \textit{ such that } g^x \equiv y \bmod p] > \epsilon$$

(where the probability is taken over $y \in \mathbf{Z}_p^*$ and the coin tosses of A) and A runs in time polynomial in $|p|$. Then there is a probabilistic algorithm A' running in time polynomial in $\epsilon^{-1}, \delta^{-1}$, and $|p|$ such that for all primes $p \in S$, generators g of $\mathbf{Z}_p^*$, and $y \in \mathbf{Z}_p^*$

$$\Pr[A'(p,g,y) = x \textit{ such that } g^x \equiv y \bmod p] > 1 - \delta$$

(where the probability is taken over the coin tosses of A').

Proof: Choose the smallest integer N for which $\frac{1}{e^N} < \delta$.

Consider the algorithm A$'$ running as follows on inputs $p \in S$, g a generator of $\mathbf{Z}_p^*$ and $y \in \mathbf{Z}_p^*$.

Repeat $\epsilon^{-1}N$ times.
  Randomly choose z such that $1 \le z \le p-1$.
  Let $w = A(p,g,g^z y)$
  If A succeeds then $g^w = g^z y = g^{z+x} \bmod p$ where $x = \mathrm{DL}_{p,g}(y)$
    and therefore $\mathrm{DL}_{p,g}(y) = w - z \bmod p-1$.
  Otherwise, continue to next iteration.
End loop

We can estimate the probability that A$'$ fails:

$$\begin{aligned}
\Pr[A'(p,g,y) \text{ fails}] &= \Pr[\text{A single iteration of the loop of A}' \text{ fails}]^{\epsilon^{-1}N} \\
&< (1-\epsilon)^{\epsilon^{-1}N} \\
&< (e^{-N}) \\
&< \delta
\end{aligned}$$

Note that since $N = O(\log(\delta^{-1})) = O(\delta^{-1})$, A$'$ is a probabilistic algorithm which runs in time polynomial in ϵ^{-1}, δ^{-1}, and $|p|$. ∎

The discrete logarithm problem also yields the following collection of functions.

Let $I = \{(p,g) : p$ is prime and g is a generator of $\mathbf{Z}_p^*\}$ and define

$$\mathrm{EXP} = \{\mathrm{EXP}_{p,g} : \mathbf{Z}_{p-1} \to \mathbf{Z}_p^* \text{ where } \mathrm{EXP}_{p,g}(x) = g^x \bmod p\}_{(p,g)\in I}.$$

Then, under the strong discrete logarithm assumption, EXP is a collection of strong one way functions. This claim will be shown to be true next.

Theorem 2.3.9 *Under the strong discrete logarithm assumption there exists a collection of strong one way functions.*

Proof: We shall show that under the DLA EXP is indeed a collection of one way functions. For this we must show that it satisfies each of the conditions in the definition of a collection of one way functions.

For condition 1, define S_1 to run as follows on input 1^k.

(1) Run Bach's algorithm (given in [8]) to get a random integer n such that $|n| = k$ along with its factorization.

(2) Test whether $n+1$ is prime. See primality testing in section C.9.

(3) If so, let $p = n+1$. Given the prime factorization of $p-1$ we look for generators g of $\mathbf{Z}_p^*$ as follows.

  (1) Choose $g \in \mathbf{Z}_p^*$ at random.

  (2) If $p - 1 = \prod_i {q_i}^{\alpha_i}$ is the prime factorization of $p-1$ then for each q_i check that $g^{\frac{p-1}{q_i}} \not\equiv 1 \bmod p$.

  If so, then g is a generator of $\mathbf{Z}_p^*$. Output p and g.

Otherwise, repeat from step 1.

Claim 2.3.10 g is a generator of $\mathbf{Z}_p^*$ if for each prime divisor q of $p-1$, $g^{\frac{p-1}{q}} \not\equiv 1 \bmod p$.

Proof: The element g is a generator of $\mathbf{Z}_p^*$ if $g^{p-1} \equiv 1 \bmod p$ and $g^j \not\equiv 1 \bmod p$ for all j such that $1 \leq j < p-1$; that is, g has order $p-1$ in $\mathbf{Z}_p^*$.

Now, suppose that g satisfies the condition of Claim 2.3.10 and let m be the order of g in $\mathbf{Z}_p^*$. Then $m \mid p-1$. If $m < p-1$ then there exists a prime q such that $m \mid \frac{p-1}{q}$; that is, there is an integer d such that $md = \frac{p-1}{q}$. Therefore $g^{\frac{p-1}{q}} = (g^m)^d \equiv 1 \bmod n$ contradicting the hypothesis. Hence, $m = p-1$ and g is a generator of $\mathbf{Z}_p^*$. ∎

Also, note that the number of generators in $\mathbf{Z}_p^*$ is $\varphi(p-1)$ and in [166] it is shown that

$$\varphi(k) > \frac{k}{6 \log \log k}.$$

Thus we expect to have to choose $O(\log \log p)$ candidates for g before we obtain a generator. Hence, S_1 runs in expected polynomial time.

For condition 2 in the definition of a collection of one way functions, we can define S_2 to simply output $x \in Z_{p-1}$ at random given $i = (p, g)$.

Condition 3 is true since the computation of $g^x \bmod p$ can be performed in polynomial time and condition 4 follows from the strong discrete logarithm assumption. ∎

2.3.2 The RSA function

In 1977 Rivest, Shamir, and Adleman [164] proposed trapdoor function candidate motivated by finding a public-key cryptosystem satisfying the requirements proposed by Diffie and Hellman. The trapdoor function proposed is $RSA(n, e, x) = x^e \bmod n$ where the case of interest is that n is the product of two large primes p and q and $\gcd(e, \phi(n)) = 1$. The corresponding trapdoor information is d such that $d \cdot e \equiv 1 \bmod \phi(n)$.

Viewd as a collection, let RSA $= \{\mathrm{RSA}_{n,e} : \mathbf{Z}_n^* \to \mathbf{Z}_n^*$ where $\mathrm{RSA}_{n,e}(x) = x^e \bmod n\}_{(n,e) \in I}$. for $I = \{< n, e >$ s.t. $n = pq$ $|p| = |q|, (e, \phi(n)) = 1\}$.

RSA is easy to compute. How hard is it to invert? We know that if we can factor n we can invert RSA via the Chinese Remainder Theorem, however we don't know if the converse is true. Thus far, the best way known to invert RSA is to first factor n. There are a variety of algorithms for this task. The best running time for a fully proved algorithm is Dixon's random squares algorithms which runs in time $O(e^{\sqrt{\log n \log \log n}})$.

In practice we may consider others. Let $\ell = |p|$ where p is the smallest prime divisor of n. The Elliptic Curve algorithm takes expected time $O(e^{\sqrt{2\log\ell\log\log\ell}})$. The Quadratic Sieve algorithm runs in expected $O(e^{\sqrt{\ln n \ln\ln n}})$. Notice the difference in the argument of the superpolynomial component of the running time. This means that when we suspect that one prime factor is substantially smaller than the other, we should use the Elliptic Curve method, otherwise one should use the Quadratic sieve. The new number field sieve algorithm seems to achieve a $O(e^{1.9(\ln n)^{1/3}(\ln\ln n)^{2/3}})$ running time which is a substantial improvement asymptotically although in practice it still does not seem to run faster than the Quadratic Sieve algorithm for the size of integers which people currently attempt to factor. The recommended size for n these days is 1024 bits.

With all this in mind, we make an explicit assumption under which one can prove that RSA provides a collection of trapdoor functions.

Strong RSA Assumption:[2] Let $H_k = \{n = pq : p \neq q$ are primes and $|p| = |q| = k\}$. Then for every polynomial Q and every PTM A, there exists an integer k_0 such that $\forall k > k_0$

$$\Pr[A(n, e, \mathrm{RSA}_{n,e}(x)) = x] < \frac{1}{Q(k)}$$

(where the probability is taken over all $n \in H_k$, e such that $\gcd(e, \varphi(n)) = 1$, $x \in \mathbf{Z}_n^*$, and the coin tosses of A).

We need to prove some auxilary claims.

Claim 2.3.11 For $(n, e) \in I$, $\mathrm{RSA}_{n,e}$ is a permutation over $\mathbf{Z}_n^*$.

Proof: Since $\gcd(e, \varphi(n)) = 1$ there exists an integer d such that $ed \equiv 1 \bmod \varphi(n)$. Given $x \in \mathbf{Z}_n^*$, consider the element $x^d \in \mathbf{Z}_n^*$. Then $\mathrm{RSA}_{n,e}(x^d) \equiv (x^d)^e \equiv x^{ed} \equiv x \bmod n$. Thus, the function $\mathrm{RSA}_{n,e} : \mathbf{Z}_n^* \longrightarrow \mathbf{Z}_n^*$ is onto and since $|\mathbf{Z}_n^*|$ is finite it follows that $\mathrm{RSA}_{n,e}$ is a permutation over $\mathbf{Z}_n^*$. ▮

Remark 2.3.12 Note that the above is a constructive proof that RSA has an unique inverse. Since $\gcd(e, \varphi(n)) = 1$ if we run the extended Euclidean algorithm we can find $d \in \mathbf{Z}_n^*$ such that

$$RSA_{n,e}^{-1}(x) = (x^e \bmod n)^d \bmod n = x^{ed} \bmod n = x \bmod n$$

. Note that once we found a d such that $ed \equiv 1 \bmod \varphi(n)$ then we can invert $\mathrm{RSA}_{n,e}$ efficiently because then $\mathrm{RSA}_{n,e}(x)^d \equiv x^{ed} \equiv x \bmod \varphi(n)$.

Theorem 2.3.13 *Under the strong RSA assumption, RSA is a collection of strong one way trapdoor permutations.*

Proof: First note that by Claim 2.3.11, $\mathrm{RSA}_{n,e}$ is a permutation of $\mathbf{Z}_n^*$. We must also show that RSA satisfies each of the conditions in Definition 2.2.16. For condition 1, define S_1 to compute, on input 1^k,

a pair $(n, e) \in I \cap \{0,1\}^k$ and corresponding d such that $ed \equiv 1 \bmod \varphi(n)$. The algorithm picks two random primes of equal size by choosing random numbers and testing them for primality and setting n to be their procuct, then $e \in Z_{\phi(n)}$ is chosen at random, and finally d is computed in polynomial time by first computing $\varphi(n) = (p-1)(q-1)$ and then using the extended Euclidean algorithm. For condition 2, define

[2] A weaker assumption can be made which under standard constructions is equivalent to the stronger one which is made in this class. **Weak RSA Assumption:** Let $H_k = \{n = pq : p \neq q$ are prime and $|p| = |q| = k\}$. There is a polynomial Q such that for every PTM A, there exists an integer k_0 such that $\forall k > k_0$ $\Pr[A(n, e, \mathrm{RSA}_{n,e}(x)) = x] < 1 - \frac{1}{Q(k)}$ (where the probability is taken over all $n \in H_k$, e such that $\gcd(e, \varphi(n)) = 1$, $x \in \mathbf{Z}_n^*$, and the coin tosses of A).

S_2 to randomly generate $x \in \mathbf{Z}_n^*$ on input (n, e). Let $A_1((n, e), x) = \mathrm{RSA}_{n,e}(x)$. Note that exponentiation modulo n is a polynomial time computation and therefore condition 3 holds. Condition 4 follows from the Strong RSA assumption. For condition 5, let $A_2((n, e), d, \mathrm{RSA}_{n,e}(x)) \equiv \mathrm{RSA}_{n,e}(x)^d \equiv x^{ed} \equiv x \bmod n$ and this is a polynomial time computation. ▮

One of the properties of the RSA function is that if we have a polynomial time algorithm that inverts $RSA_{n,e}$ on at least a polynomial proportion of the possible inputs $x \in Z_n^*$ then a subsequent probabilistic expected polynomial time algorithm can be found which inverts $\mathrm{RSA}_{n,e}$ on almost all inputs $x \in Z_n^*$. This can be taken to mean that for a given n, e if the function is hard to invert then it is almost everywhere hard to invert.

Proposition 2.3.14 *Let ϵ, $\delta \in (0, 1)$ and let $S \subseteq I$. Suppose there is a probabilistic algorithm A such that for all $(n, e) \in S$*

$$\Pr[A(n, e, RSA_{n,e}(x)) = x] > \epsilon$$

(where the probability is taken over $x \in \mathbf{Z}_n^*$ and the coin tosses of A) and A runs in time polynomial in $|n|$. Then there is a probabilistic algorithm A' running in time polynomial in $\epsilon^{-1}, \delta^{-1}$, and $|n|$ such that for all $(n, e) \in S$, and $x \in \mathbf{Z}_n^*$

$$\Pr[A'(n, e, RSA_{n,e}(x)) = x] > 1 - \delta$$

(where the probability is taken over the coin tosses of A').

Proof: Choose the smallest integer N for which $\frac{1}{e^N} < \delta$.

Consider the algorithm A$'$ running as follows on inputs $(n, e) \in S$ and $\mathrm{RSA}_{n,e}(x)$.

Repeat $\epsilon^{-1}N$ times.
    Randomly choose $z \in \mathbf{Z}_n^*$.
    Let $y = A(n, e, \mathrm{RSA}_{n,e}(x) \cdot \mathrm{RSA}_{n,e}(z)) = A(n, e, \mathrm{RSA}_{n,e}(xz))$.
    If A succeeds then $y = xz$ and therefore $x = yz^{-1} \bmod n$. Output x.
    Otherwise, continue to the next iteration.
End loop

We can estimate the probability that A$'$ fails:

$$\begin{aligned}\Pr[A'(n, e, RSA_{n,e}(x)) \neq x] &= \Pr[\text{A single iteration of the loop of A}' \text{ fails}]^{\epsilon^{-1}N} \\ &< (1-\epsilon)^{\epsilon^{-1}N} \\ &< (e^{-N}) \\ &< \delta\end{aligned}$$

Note that since $N = O(\log(\delta^{-1})) = O(\delta^{-1})$, A$'$ is a probabilistic algorithm which runs in time polynomial in ϵ^{-1}, δ^{-1}, and $|n|$. ▮

Open Problem 2.3.15 It remains to determine whether a similar result holds if the probability is also taken over the indices $(n, e) \in I$. Specifically, if ϵ, $\delta \in (0, 1)$ and A is a PTM such that

$$\Pr[A(n, e, RSA_{n,e}(x)) = x] > \epsilon$$

(where the probability is taken over $(n, e) \in I$, $x \in \mathbf{Z}_n^*$ and the coin tosses of A), does there exist a PTM A' running in time polynomial in ϵ^{-1} and δ^{-1} such that

$$\Pr[A'(n, e, RSA_{n,e}(x)) = x] > 1 - \delta$$

(where the probability is taken over $(n, e) \in I$ and the coin tosses of A$'$)?

2.3.3 Connection Between The Factorization Problem And Inverting RSA

Fact 2.3.16 If some PPT algorithm A can factor n then there exists a PPT A' that can invert $RSA_{\langle n,e\rangle}$.

The proof is obvious as $\phi(n) = (p-1)(q-1)$. The trapdoor information d can be found by using the extended Euclidean algorithm because $d = e^{-1} \bmod \phi(n)$.

Fact 2.3.17 If there exists a PTM B which on input $\langle n, e\rangle$ finds d such that $ed \equiv 1 \bmod \phi(n)$ then there exists a PTM, B$'$ that can factor n.

Open Problem 2.3.18 It remains to determine whether inverting RSA and factoring are equivalent. Namely, if there is a PTM C which, on input $\langle n, e\rangle$, can invert $RSA_{\langle n,e\rangle}$, does there exist a PTM C' that can factor n? The answer to this question is unknown. Note that Fact 2.3.17 does not imply that the answer is yes, as there may be other methods to invert RSA which do not necessarily find d.

2.3.4 The Squaring Trapdoor Function Candidate by Rabin

Rabin in [157] introduced a candidate trapdoor function which we call the squaring function. The squaring function resemble the RSA function except that Rabin was able to actually *prove* that inverting the squaring function is as hard as factoring integers. Thus, inverting the squaring function is a computation which is at least as hard as inverting the RSA function and possibly harder.

Definition 2.3.19 Let $I = \{n = pq : p \text{ and } q \text{ are distinct odd primes.}\}$. For $n \in$ I, the squaring function $SQUARE_n : \mathbf{Z}_n^* \longrightarrow \mathbf{Z}_n^*$ is defined by $SQUARE_n(x) \equiv x^2 \bmod n$. The trapdoor information of $n = pq \in I$ is $t_n = (p, q)$. We will denote the entire collection of Rabin's functions by RABIN $= \{SQUARE_n : \mathbf{Z}_n^* \longrightarrow \mathbf{Z}_n^*\}_{n\in I}$.

Remark 2.3.20 Observe that while Rabin's function squares its input, the RSA function uses a varying exponent; namely, e where $\gcd(e, \phi(n)) = 1$. The requirement that $\gcd(e, \phi(n)) = 1$ guarentees that the RSA function is a permutation. On the other hand, Rabin's function is 1 to 4 and thus it does not have a uniquely defined inverse. Specifically, let $n = pq \in I$ and let $a \in \mathbf{Z}_p^*$. As discussed in section C.4, if $a \equiv x^2 \bmod p$ then x and $-x$ are the distinct square roots of a modulo p and if $a \equiv y^2 \bmod q$ then y and $-y$ are the distinct square roots of a modulo q. Then, there are four solutions to the congruence $a \equiv z^2 \bmod n$, constructed as follows. Let $c, d \in \mathbf{Z}_n$ be the Chinese Remainder Theorem coefficients as discussed in Appendix C.4. Then

$$c = \begin{cases} 1 \bmod p \\ 0 \bmod q \end{cases}$$

and

$$d = \begin{cases} 0 \bmod p \\ 1 \bmod q \end{cases}$$

and the four solutions are $cx + dy$, $cx - dy$, $-cx + dy$, and $-cx - dy$.

The main result is that RABIN is a collection of strong one way trapdoor functions and the proof relies on an assumption concerning the difficulty of factoring. We state this assumption now.

Factoring Assumption: Let $H_k = \{pq : p \text{ and } q \text{ are prime and } |p| = |q| = k\}$. Then for every polynomial Q and every PTM A, $\exists k_0$ such that $\forall k > k_0$

$$\Pr[A(n) = p : p \mid n \text{ and } p \neq 1, n] < \frac{1}{Q(k)}$$

(where the probability is taken over all $n \in H_k$ and the coin tosses of A).

Our ultimate goal is to prove the following result.

Theorem 2.3.21 *Under the factoring assumption, RABIN is a collection of one way trapdoor functions.*

Before proving this, we consider two auxiliary lemmas. Lemma 2.3.22 constructs a polynomial-time machine A which computes square roots modulo a prime. Lemma 2.3.25 constructs another polynomial-time machine, SQRT, that inverts Rabin's function using the trapdoor information; specifically, it computes a square root modulo composites given the factorization. SQRT makes calls to A.

Lemma 2.3.22 *Let p be an odd prime and let a be a square modulo p. There exists a probabilistic algorithm A running in expected polynomial time such that $A(p, a) = x$ where $x^2 \equiv a \bmod p$.*

Proof: Let p be an odd prime and let a be a quadratic residue in $\mathbf{Z}_p^*$. There are two cases to consider; $p \equiv 1 \bmod 4$ and $p \equiv 3 \bmod 4$.

Case 1 $p \equiv 3 \bmod 4$; that is, $p = 4m + 3$ for some integer m.

Since a is a square we have $1 = \mathbf{J}_p(a) \equiv a^{\frac{p-1}{2}} \bmod p \Longrightarrow a^{2m+1} \equiv 1 \bmod p$
$$\Longrightarrow a^{2m+2} \equiv a \bmod p$$

Therefore, a^{m+1} is a square root of a modulo p.

Case 2 $p \equiv 1 \bmod 4$; that is, $p = 4m + 1$ for some integer m.

As in Case 1, we will attempt to find an odd exponent e such that $a^e \equiv 1 \bmod p$.

Again, a is a square and thus $1 = \mathbf{J}_p(a) \equiv a^{\frac{p-1}{2}} \bmod p \Longrightarrow a^{2m} \equiv 1 \bmod p$.

However, at this point we are not done as in Case 1 because the exponent on a in the above congruence is even. But notice that $a^{2m} \equiv 1 \bmod p \Longrightarrow a^m \equiv \pm 1 \bmod p$. If $a^m \equiv 1 \bmod p$ with m odd, then we proceed as in Case 1.

This suggests that we write $2m = 2^l r$ where r is an odd integer and compute $a^{2^{l-i}r} \bmod p$ for $i = 1, \ldots, l$ with the intention of reaching the congruence $a^r \equiv 1 \bmod p$ and then proceeding as in Case 1. However, this is not guarenteed as there may exist an integer l' satisfying $0 \leq l' < l$ such that $a^{2^{l'}r} \equiv -1 \bmod p$. If this congruence is encountered, we can recover as follows. Choose a quadratic nonresidue $b \in \mathbf{Z}_p^*$. Then $-1 = \mathbf{J}_p(b) \equiv b^{\frac{p-1}{2}} \bmod p$ and therefore $a^{2^{l'}r} \cdot b^{2^l r} = a^{2^{l'}r} \cdot b^{\frac{p-1}{2}} \equiv 1 \bmod p$. Thus, by multiplying by $b^{2^l r} \equiv -1 \bmod p$, we obtain a new congruence $\left(a^r b^{2^{l-l'}r}\right)^{2^{l'}} \equiv 1 \bmod p$. We proceed by taking square roots in this congruence. Since $l' < l$, we will, after l steps, arrive at $a^r b^{2s} \equiv 1 \bmod p$ where s is integral. At this point we have $a^{r+1} b^{2s} \equiv a \bmod p \Longrightarrow a^{\frac{r+1}{2}} b^s$ is a square root of $a \bmod p$.

¿From the above discussion (Cases 1 and 2) we obtain a probabilistic algorithm A for taking square roots. The algorithm A runs as follows on input a, p where $\mathbf{J}_p(a) = 1$.

(1) If $p = 4m + 3$ for some integer m then output a^{m+1} as a square root of $a \bmod p$.

(2) If $p = 4m + 1$ for some integer m then randomly choose $b \in \mathbf{Z}_p^*$ until a value is found satisfying $\mathbf{J}_p(b) = -1$.

(1) Initialize $i = 2m$ and $j = 0$.

(2) Repeat until i is odd.

$i \leftarrow \frac{i}{2}$ and $j \leftarrow \frac{j}{2}$.

If $a^i b^j = -1$ then $j \leftarrow j + 2m$.

Output $a^{\frac{i+1}{2}} b^{\frac{j}{2}}$ as a square root of $a \bmod p$.

This algorithm terminates after $O(l)$ iterations because in step 2 (ii) the exponent on a is divided by 2. Note also, that since exactly half of the elements in $\mathbf{Z}_p^*$ are quadratic nonresidues, it is expected that 2 iterations will be required to find an appropriate value for b at the beginning of step 2. Thus, A runs in expected polynomial time and this completes the proof of Lemma 2.3.22. ∎

Remark 2.3.23 There is a deterministic algorithm due to René Schoof (see [173]) which computes the square root of a quadratic residue a modulo a prime p in time polynomial in $|p|$ and a (specifically, the algorithm requires $O((a^{\frac{1}{2}+\epsilon} \log p)^9)$ elementary operations for any $\epsilon > 0$). However, it is unknown whether there exists a deterministic algorithm running in time polynomial in $|p|$.

Open Problem 2.3.24 Does there exist a deterministic algorithm that computes square roots modulo a prime p in time polynomial in $|p|$?

The next result requires knowledge of the Chinese Remainder Theorem. The statement of this theorem as well as a constructive proof is given in Appendix C.4. In addition, a more general form of the Chinese Remainder Theorem is presented there.

Lemma 2.3.25 *Let p and q be primes, $n = pq$ and a a square modulo p. There exists a probabilistic algorithm SQRT running in expected polynomial time such that $SQRT(p, q, n, a) = x$ where $x^2 \equiv a \bmod n$.*

Proof: The algorithm SQRT will first make calls to A, the algorithm of Lemma 2.3.22, to obtain square roots of a modulo each of the primes p and q. It then combines these square roots, using the Chinese Remainder Theorem, to obtain the required square root.

The algorithm SQRT runs as follows.

(1) Let $A(p, a) = x_1$ and $A(q, a) = x_2$.

(2) Use the Chinese Remainder Theorem to find (in polynomial time) $y \in \mathbf{Z}_n$ such that $y \equiv x_1 \bmod p$ and $y \equiv x_2 \bmod q$ and output y.

Algorithm SQRT runs correctly because $y^2 \equiv \begin{cases} x_1^2 \equiv a \bmod p \\ x_2^2 \equiv a \bmod q \end{cases} \implies y^2 \equiv a \bmod n$. ∎

On the other hand, if the factors of n are unknown then the computation of square roots modulo n is as hard as factoring n. We prove this result next.

Lemma 2.3.26 *Computing square roots modulo $n \in H_k$ is as hard as factoring n.*

Proof: Suppose that I is an algorithm which on input $n \in H_k$ and a a square modulo n outputs y such that $a \equiv y^2 \bmod n$ and consider the following algorithm B which on input n outputs a nontrivial factor of n.

(1) Randomly choose $x \in \mathbf{Z}_n^*$.

(2) Set $y = I(n, x^2 \bmod n)$.

(3) Check if $x \equiv \pm y \bmod n$. If not then $\gcd(x-y, n)$ is a nontrivial divisor of n. Otherwise, repeat from 1.

Algorithm B runs correctly because $x^2 \equiv y^2 \bmod n \Longrightarrow (x+y)(x-y) \equiv 0 \bmod n$ and so $n|[(x+y)(x-y)]$. But $n \not| (x-y)$ because $x \not\equiv y \bmod n$ and $n \not| (x+y)$ because $x \not\equiv -y \bmod n$. Therefore, $\gcd(x-y, n)$ is a nontrivial divisor of n. Note also that the congruence $a \equiv x^2 \bmod n$ has either 0 or 4 solutions (a proof of this result is presented in Appendix C.4). Therefore, if $I(n, x^2) = y$ then $x \equiv \pm y \bmod n$ with probability $\frac{1}{2}$ and hence the above algorithm is expected to terminate in 2 iterations. ∎

We are now in a position to prove the main result, Theorem 2.3.21.

Proof: For condition 1, define S_1 to find on input 1^k an integer $n = pq$ where p and q are primes of equal length and $|n| = k$. The trapdoor information is the pair of factors (p, q).

For condition 2 in the definition of a collection of one way trapdoor functions, define S_2 to simply output $x \in Z_n^*$ at random given n.

Condition 3 is true since the computation of $x^2 \bmod n$ can be performed in polynomial time and condition 4 follows from the factoring assumption and Lemma 2.3.26.

Condition 5 follows by applying the algorithm $SQRT$ from Lemma 2.3.25. ∎

Lemma 2.3.26 can even be made stronger as we can also prove that if the algorithm I in the proof of Lemma 2.3.26 works only on a small portion of its inputs then we are still able to factor in polynomial time.

Proposition 2.3.27 *Let $\epsilon, \delta \in (0,1)$ and let $S \subseteq H_k$. Suppose there is a probabilistic algorithm I such that for all $n \in S$*

$$\Pr[I(n,a) = x \textit{ such that } a \equiv x^2 \bmod n] > \epsilon$$

(where the probability is taken over $n \in S$, $a \in {\mathbf{Z}_n^*}^2$, and the coin tosses of I). Then there exists a probabilistic algorithm FACTOR running in time polynomial in ϵ^{-1}, δ^{-1}, and $|n|$ such that for all $n \in S$,

$$\Pr[\textit{FACTOR}(n) = d \textit{ such that } d \mid n \textit{ and } d \neq 1, n] > 1 - \delta$$

(where the probability is taken over n and over the coins tosses of FACTOR).

Proof: Choose the smallest integer N such that $\frac{1}{e^N} < \delta$.

Consider the algorithm FACTOR running as follows on inputs $n \in S$.

Repeat $2\epsilon^{-1}N$ times.
  Randomly choose $x \in \mathbf{Z}_n^*$.
  Set $y = I(n, x^2 \bmod n)$.
  Check if $x \equiv \pm y \bmod n$. If not then $\gcd(x-y, n)$ is a nontrivial divisor of n.
  Otherwise, continue to the next iteration.
End loop

We can estimate the probability that FACTOR fails. Note that even when $I(n, x^2 \bmod n)$ produces a square root of $x^2 \bmod n$, FACTOR(n) will be successful exactly half of the time.

$$\begin{aligned}\Pr[\text{FACTOR}(n) \text{ fails to factor } n] &= \Pr[\text{A single iteration of the loop of FACTOR fails}]^{\epsilon^{-1}N} \\ &< (1-\tfrac{1}{2}\epsilon)^{2\epsilon^{-1}N} \\ &< (e^{-N}) \\ &< \delta\end{aligned}$$

Since $N = O(\log(\delta^{-1})) = O(\delta^{-1})$, FACTOR is a probabilistic algorithm which runs in time polynomial in ϵ^{-1}, δ^{-1}, and $|n|$. ▮

2.3.5 A Squaring Permutation as Hard to Invert as Factoring

We remarked earlier that Rabin's function is not a permutation. If $n = pq$ where p and q are primes and $p \equiv q \equiv 3 \bmod 4$ then we can reduce the Rabin's function $SQUARE_n$ to a permutation g_n by restricting its domain to the quadratic residues in $\mathbf{Z}_n^*$, denoted by Q_n. This will yield a collection of one way permutations as we will see in Theorem 2.3.5. This suggestion is due to Blum and Williams.

Definition 2.3.28 Let $J = \{pq : p \neq q \text{ are odd primes}, |p| = |q|, \text{ and } p \equiv q \equiv 3 \bmod 4\}$. For $n \in$ J let the function $g_n : Q_n \longrightarrow Q_n$ be defined by $g_n(x) \equiv x^2 \bmod n$ and let BLUM-WILLIAMS = $\{g_n\}_{n \in J}$.

We will first prove the following result.

Lemma 2.3.29 *Each function $g_n \in$ BLUM-WILLIAMS is a permutation. That is, for every element $y \in Q_n$ there is a unique element $x \in Q_n$ such that $x^2 = y \bmod n$.*

Proof: Let $n = p_1 p_2 \in$ J. Note that by the Chinese Remainder Theorem, $y \in Q_n$ if and only if $y \in Q_n$ and $y \in Q_{p_1}$ and $y \in Q_{p_2}$. Let a_i and $-a_i$ be the square roots of $y \bmod p_i$ for $i = 1, 2$. Then, as is done in the proof of the Chinese Remainder Theorem, we can construct Chinese Remainder Theorem coefficients c_1, c_2 such that $c_1 = \begin{cases} 1 \bmod p_1 \\ 0 \bmod p_2 \end{cases}$ and $c_2 = \begin{cases} 0 \bmod p_1 \\ 1 \bmod p_2 \end{cases}$ and consequently, the four square

roots of $y \bmod n$ are
$$\begin{aligned} w_1 &= c_1a_1 + c_2a_2, \\ w_2 &= c_1a_1 - c_2a_2, \\ w_3 &= -c_1a_1 - c_2a_2 = -(c_1a_1 + c_2a_2) = -w_1, \\ \text{and } w_4 &= -c_1a_1 + c_2a_2 = -(c_1a_1 - c_2a_2) = -w_2. \end{aligned}$$

Since $p_1 \equiv p_2 \equiv 3 \bmod 4$, there are integers m_1 and m_2 such that $p_1 = 4m_1 + 3$ and $p_2 = 4m_2 + 3$. Thus, $\mathbf{J}_{p_1}(w_3) = \mathbf{J}_{p_1}(-w_1) = \mathbf{J}_{p_1}(-1)\mathbf{J}_{p_1}(w_1) = (-1)^{\frac{p_1-1}{2}}\mathbf{J}_{p_1}(w_1) = -\mathbf{J}_{p_1}(w_1)$ because $\frac{p_1-1}{2}$ is odd and similarly, $\mathbf{J}_{p_1}(w_4) = -\mathbf{J}_{p_1}(w_2)$, $\mathbf{J}_{p_2}(w_3) = -\mathbf{J}_{p_2}(w_1)$, and $\mathbf{J}_{p_2}(w_4) = -\mathbf{J}_{p_2}(w_2)$. Therefore, without loss of generality, we can assume that $\mathbf{J}_{p_1}(w_1) = \mathbf{J}_{p_1}(w_2) = 1$ (and so $\mathbf{J}_{p_1}(w_3) = \mathbf{J}_{p_1}(w_4) = -1$).

Since only w_1 and w_2 are squares modulo p_1 it remains to show that only one of w_1 and w_2 is a square modulo n or equivalently modulo p_2.

First observe that $\mathbf{J}_{p_2}(w_1) \equiv (w_1)^{\frac{p_2-1}{2}} \equiv (c_1a_1 + c_2a_2)^{2m_2+1} \equiv (a_2)^{2m_2+1} \bmod p_2$ and that $\mathbf{J}_{p_2}(w_2) \equiv (w_2)^{\frac{p_2-1}{2}} \equiv (c_1a_1 - c_2a_2)^{2m_2+1} \equiv (-a_2)^{2m_2+1} \bmod p_2$ (because $c_1 \equiv 0 \bmod p_2$ and $c_2 \equiv 1 \bmod p_2$). Therefore, $\mathbf{J}_{p_2}(w_2) = -\mathbf{J}_{p_2}(w_1)$. Again, without loss of generality, we can assume that $\mathbf{J}_{p_2}(w_1) = 1$ and $\mathbf{J}_{p_2}(w_2) = -1$ and hence, w_1 is the only square root of y that is a square modulo both p_1 and p_2. Therefore, w_1 is the only square root of y in Q_n. ▮

Theorem 2.3.30 [Williams, Blum] *BLUM-Williams is a collection of one-way trapdoor permutations.*

Proof: This follows immediately from Lemma 2.3.29 because each function $g_n \in$ J is a permutation. The trapdoor information of $n = pq$ is $t_n = (p, q)$. ▮

2.4 Hard-core Predicate of a One Way Function

Recall that $f(x)$ does not necessarily hide everything about x even if f is a one-way function. E.g. if f is the RSA function then it preserves the Jacobi symbol of x, and if f is the discrete logarithm function EXP then it is easy to compute the least significant bit of x from $f(x)$ by a simple Legendre symbol calculation. Yet, it seems likely that there is at least one bit about x which is hard to "guess" from $f(x)$, given that x in its entirety is hard to compute. The question is: can we point to specific bits of x which are hard to compute, and how hard to compute are they. The answer is encouraging. A number of results are known which give a particular bit of x which is hard to guess given $f(x)$ for some particular f's such as RSA and the discrete logarithm function. We will survey these results in subsequent sections.

More generally, we call a predicate about x which is impossible to compute from $f(x)$ better than guessing it at random a *hard-core predicate* for f.

We first look at a general result by Goldreich and Levin [90] which gives for any one-way function f a predicate B such that it is as hard to guess $B(x)$ from $f(x)$ as it is to invert f.

Historical Note: The idea of a hard-core predicate for one-way functions was introduced by Blum, Goldwasser and Micali. It first appears in a paper by Blum and Micali [39] on pseduo random number generation. They showed that a if the EXP function ($f_{p,g}(x) = g^x \pmod p$) is hard to invert then it is hard to even guess better than guessing at random the most significant bit of x. Under the assumption that quadratic residues are hard to distinguish from quadratic non-residues modulo composite moduli, Goldwasser and Micali in [94] showed that the squaring function has a hard core perdicate as well. Subsequently, Yao [194] showed a general result that given any one way function, there is a predicate $B(x)$ which is as hard to guess from $f(x)$ as to invert f for any function f. Goldreich and Levin's result is a significantly simpler construction than Yao's earlier construction.

2.4.1 Hard Core Predicates for General One-Way Functions

We now introduce the concept of a *hard-core predicate* of a function and show by explicit construction that any strong one way function can be modified to have a hard-core predicate.

Note: Unless otherwise mentioned, the probabilities during this section are calculated uniformly over all coin tosses made by the algorithm in question.

Definition 2.4.1 A hard-core predicate of a function $f : \{0,1\}^* \to \{0,1\}^*$ is a boolean predicate $B : \{0,1\}^* \to \{0,1\}$, such that

(1) $\exists PPT\, A$, such that $\forall x\, A(x) = B(x)$

(2) $\forall\, \mathrm{PPT}\, G,\ \forall\, \mathrm{constants}\, c,\ \exists\, k_0,\ \mathrm{s.t.} \forall_{k>k_0}$

$$\Pr[G(f(x)) = B(x)] < \frac{1}{2} + \frac{1}{k^c}.$$

The probability is taken over the random coin tosses of G, and random choices of x of length k.

Intuitively, the definition guarantees that given x, $B(x)$ is efficiently computable, but given only $f(x)$, it is hard to even "guess" $B(x)$; that is, to guess $B(x)$ with a probability significantly better than $\frac{1}{2}$.

Yao, in [194], showed that the existence of any trapdoor length-preserving permutation implies the existence of a trapdoor predicate. Goldreich and Levin greatly simplified Yao's construction and show that any one-way function can be modified to have a trapdoor predicate as follows (we state a simple version of their general result).

Theorem 2.4.2 [90] *Let f be a (strong) length preserving one-way function. Define $f'(x \circ r) = f(x) \circ r$, where $|x| = |r| = k$, and $\circ$ is the concatenation function. Then*

$$B(x \circ r) = \Sigma_{i=1}^{k} x_i r_i (mod\ 2).$$

is a hard-core predicate for f'.

Note: $v \circ w$ denotes concatenation of strings v and w. Computing B from f' is trivial as $f(x)$ and r are easily recoveravle from $f'(x,r)$. Finaly notice that if f is one-way then so is f'.

For a full proof of the theorem we refer the reader to [90].

It is trivial to extend the definition of a hard-core predicate for a one way function, to a collection of hard core predicates for a collection of one-way functions.

Definition 2.4.3 A hard-core predicate of a one-way function collection $F = \{f_i : D_i \rightarrow R_i\}_{i \in I}$ is a collection of boolean predicates $B = \{B_i : D_i \rightarrow R_i\}_{i \in I}$ such that

(1) $\exists PPT\, A$, such that $\forall i, x\; A(i,x) = B_i(x)$

(2) $\forall\, \mathrm{PPT}\, G,\ \forall\, \mathrm{constants}\, c,\ \exists\ , k_0,\ \mathrm{s.t.}\, \forall_{k > k_0}$

$$\Pr[G(i, f_i(x)) = B_i(x)] < \frac{1}{2} + \frac{1}{k^c}.$$

The probability is taken over the random coin tosses of G, random choices of $i \in I \cap \{0,1\}^k$ and random $x \in D_i$.

2.4.2 Bit Security Of The Discrete Logarithm Function

Let us examine the bit security of the EXP collection of functions directly rather than through the Goldreich Levin general construction.

We will be interested in the most significant bit of the discrete logarithm x of y modulo p.

For $(p, g) \in I$ and $y \in \mathbf{Z}_p^*$, let $B_{p,g}(y) = \begin{cases} 0 & \text{if } y = g^x \bmod p \\ & \quad \text{where } 0 \le x < \frac{p-1}{2} \\ 1 & \text{if } y = g^x \bmod p \\ & \quad \text{where } \frac{p-1}{2} \le x < p-1 \end{cases}$.

We want to show that if for p a prime and g a generator of $\mathbf{Z}_p^*$, $\mathrm{EXP}_{p,g}(x) \equiv g^x \bmod p$ is hard to invert, then given $y = \mathrm{EXP}_{p,g}(x)$, $B_{p,g}(y)$ is hard to compute in a very strong sense; that is, in attempting to compute $B_{p,g}(y)$ we can do no better than essentially guessing its value randomly. The proof will be by way of a reduction. It will show that if we can compute $B_{p,g}(y)$ in polynomial time with probability greater than $\frac{1}{2} + \epsilon$ for some non-negligible $\epsilon > 0$ then we can invert $\mathrm{EXP}_{p,g}(x)$ in time polynomial in $|p|$, $|g|$, and ϵ^{-1}. The following is a formal statement of this fact.

Theorem 2.4.4 *Let S be a subset of the prime integers. Suppose there is a polynomial Q and a PTM G such that for all primes $p \in S$ and for all generators g of $\mathbf{Z}_p^*$*

$$\Pr[G(p, g, y) = B_{p,g}(y)] > \frac{1}{2} + \frac{1}{Q(|p|)}$$

(where the probability is taken over $y \in \mathbf{Z}_p^*$ and the coin tosses of G). Then for every polynomial P, there is a PTM I such that for all primes $p \in S$, generators g of $\mathbf{Z}_p^*$, and $y \in \mathbf{Z}_p^*$

$$\Pr[I(p,g,y) = x \text{ such that } y \equiv g^x \bmod p] > 1 - \frac{1}{P(|p|)}$$

(where the probability is taken over the coin tosses of I).

We point to [39] for a proof of the above theorem.

As a corollary we immediately get the following.

Definition 2.4.5 Define $MSB_{p,g}(x) = 0$ if $1 \leq x < \frac{p-1}{2}$ and 1 otherwise for $x \in Z_{p-1}$, and $MSB = \{MSB_{p,g}(x) : Z_{p-1} \to \{0,1\}\}_{(p,g) \in I}$. for $I = \{(p,g) : p$ is prime and g is a generator of $\mathbf{Z}_p^*\}$.

Corollary 2.4.6 *Under the strong DLA, MSB is a collection of hard-core predicates for EXP.*

It can be shown that actually $O(\log \log p)$ of the most significant bits of $x \in \mathbf{Z}_{p-1}$ are hidden by the function $EXP_{p,g}(x)$. We state this result here without proof.

Theorem 2.4.7 *For a PTM A, let*

$$\alpha = Pr[A(p,g,g^x, x_{\log\log p} x_{\log\log p - 1} \ldots x_0) = 0 \mid x = x_{|p|} \ldots x_0]$$

(where the probability is taken over $x \in \mathbf{Z}_n^*$ and the coin tosses of A) and let

$$\beta = Pr[A(p,g,g^x, r_{\log\log p} r_{\log\log p - 1} \ldots r_0) = 0 \mid r_i \in_R \{0,1\}]$$

(where the probability is taken over $x \in \mathbf{Z}_n^*$, the coin tosses of A, and the bits r_i). Then under the Discrete Logarithm Assumption, we have that for every polynomial Q and every PTM A, $\exists k_0$ such that $\forall k > k_0$, $|\alpha - \beta| < \frac{1}{Q(k)}$.

Corollary 2.4.8 *Under the Discrete Logarithm Assumption we have that for every polynomial Q and every PTM A, $\exists k_0$ such that $\forall k > k_0$ and $\forall k_p < \log \log p$*

$$\Pr[A(p,g,g^x, x_{k_p} \ldots x_0) = x_{k_p+1}] < \frac{1}{2} + \frac{1}{Q(k)}$$

(where the probability is taken over the primes p such that $|p| = k$, the generators g of $\mathbf{Z}_p^*$, $x \in \mathbf{Z}_p^*$, and the coin tosses of A).

For further information on the simultaneous or individual security of the bits associated with the discrete logarithm see [127, 104].

2.4.3 Bit Security of RSA and SQUARING functions

Let $I = \{<n,e> — n = pq \ |p| = |q|, (e, \phi(n)) = 1\}$, and $RSA = \{RSA_{<n,e>} : Z_n^* \to Z_n^*\}_{<n,e> \in I}$ be the collection of functions as defined in 2.2.17.

Alexi, Chor, Goldreich and Schnoor [6] showed that guessing the least significant bit of x from $RSA_{<n,e>}(x)$ better than at random is as hard as inverting RSA.

Theorem 2.4.9 *[6] Let $S \subset I$. Let $c > 0$. If there exists a probabilistic polynomial-time algorithm O such that for $(n, e) \in S$,*

$$prob(O(n, e, x^e \bmod n) = \text{ least significant bit of } x \bmod n) \geq \frac{1}{2} + \frac{1}{k^c}$$

(taken over coin tosses of O and random choices of $x \in Z_n^*$) Then there exists a probabilistic expected polynomial time algorithm A such that for all $n, e \in S$, for all $x \in Z_n^*$, $A(n, e, x^e \bmod n) = x \bmod n$.

Now define $LSB = \{LSB_{<n,e>} : Z_n^* \to Z_n^*\}_{<n,e> \in I}$ where $LSB_{<n,e>}(x)$ =least significant bit of x.

A direct corollary to the above theorem is.

Corollary 2.4.10 *Under the (strong) RSA assumption, LSB is a collection of hard core predicates for RSA.*

A similar result can be shown for the most signifant bit of x and in fact for the $\log \log n$ least (and most) significant bits of x simultaneously. Moreover, similar results can be shown for the RABIN and BLUM-WILLIAMS collections. We refer to [6], [192] for the detailed results and proofs. Also see [78] for reductions of improved security.

2.5 One-Way and Trapdoor Predicates

A *one-way predicate*, first introduced in [93, 94] is a notion which preceeds hard core predicates for one-way functions and is strongly related to it. It will be very useful for both design of secure encryption and protocol design.

A *one-way predicate* is a boolean function $B : \{0,1\}^* \to \{0,1\}$ for which

(1) *Sampling is possible*: There exists a PPT algorithm that on input $v \in \{0,1\}$ and 1^k, selects a random x such that $B(x) = v$ and $|x| \leq k$.

(2) *Guessing is hard*: For all $c > 0$, for all k sufficiently large, no PPT algorithm given $x \in \{0,1\}^k$ can compute $B(x)$ with probability greater than $\frac{1}{2} + \frac{1}{k^c}$. (The probability is taken over the random choices made by the adversary and x such that $|x| \leq k$.)

A *trapdoor predicate* is a one-way predicate for which there exists, for every k, trapdoor information t_k whose size is bounded by a polynomial in k and whose knowledge enables the polynomial-time computation of $B(x)$, for all x such that $|x| \leq k$.

Restating as a collection of one-way and trapdoor predicates is easy.

Definition 2.5.1 Let I be a set of indices and for $i \in I$ let D_i be finite. A collection of one-way predicates is a set $B = \{B_i : D_i \to \{0,1\}\}_{i \in I}$ satisfying the following conditions. Let $D_i^v = \{x \in D_i, B_i(x) = v$.

(1) There exists a polynomial p and a PTM S_1 which on input 1^k finds $i \in I \cap \{0,1\}^k$.

(2) There exists a PTM S_2 which on input $i \in I$ and $v \in \{0,1\}$ finds $x \in D_i$ such that $B_i(x) = v$.

(3) For every PPT A there exists a negligible ν_A such that $\forall$ k large enough

$$\mathbf{P}\left[z = v \ : \ i \stackrel{R}{\leftarrow} I \cap \{0,1\}^k \ ; \ v \stackrel{R}{\leftarrow} \{0,1\} \ ; \ x \stackrel{R}{\leftarrow} D_i^v \ ; \ z \stackrel{R}{\leftarrow} A(i,x) \right] \leq \frac{1}{2} + \nu_A(k)$$

Definition 2.5.2 Let I be a set of indices and for $i \in I$ let D_i be finite. A collection of trapdoor predicates is a set $B = \{B_i : D_i \to \{0,1\}\}_{i \in I}$ satisfying the following conditions. Let $D_i^v = \{x \in D_i, B_i(x) = v$.

(1) There exists a polynomial p and a PTM S_1 which on input 1^k finds pairs (i, t_i) where $i \in I \cap \{0,1\}^k$ and $|t_i| < p(k)$ The information t_i is referred to as the trapdoor of i.

(2) There exists a PTM S_2 which on input $i \in I$ and $v \in \{0,1\}$ finds $x \in D_i$ such that $B_i(x) = v$.

(3) There exists a PTM A_1 such that for $i \in I$ and trapdoor t_i, $x \in D_i$ $A_1(i, t_i, x) = B_i(x)$.

(4) For every PPT A there exists a negligble ν_A such that $\forall$ k large enough

$$\mathbf{P}\left[z = v \ : \ i \stackrel{R}{\leftarrow} I \cap \{0,1\}^k \ ; \ v \stackrel{R}{\leftarrow} \{0,1\} \ ; \ x \stackrel{R}{\leftarrow} D_i^v \ ; \ z \leftarrow A(i,x)\right] \leq \frac{1}{2} + \nu_A(k)$$

Note that this definition implies that D_i^0 is roughly the same size as D_i^1.

2.5.1 Examples of Sets of Trapdoor Predicates

A Set of Trapdoor Predicates Based on the Quadratic Residue Assumption

Let Q_n denote the set of all *quadratic residues* (or *squares*) modulo n; that is, $x \in Q_n$ iff there exists a y such that $x \equiv y^2 \bmod n$.

Recall that the Jacobi symbol $(\mathbf{J}_n(x))$ is defined for any $x \in Z_n^*$ and has a value in $\{-1, 1\}$; this value is easily computed by using the law of quadratic reciprocity, even if the factorization of n is unknown. If n is prime then $x \in Q_n \Leftrightarrow (\mathbf{J}_n(x)) = 1$; and if n is composite, $x \in Q_n \Rightarrow (\mathbf{J}_n(x)) = 1$. We let J_n^{+1} denote the set $\{x \mid x \in Z_n^* \wedge (\mathbf{J}_n(x)) = 1\}$, and we let $\tilde{Q}_n$ denote the set of *pseudo-squares* modulo n: those elements of J_n^{+1} which do *not* belong to Q_n. If n is the product of two primes then $|Q_n| = |\tilde{Q}_n|$, and for any pseudo-square y the function $f_y(x) = y \cdot x$ maps Q_n one-to-one onto $\tilde{Q}_n$.

The *quadratic residuousity problem* is: given a composite n and $x \in J_n^{+1}$, to determine whether x is a square or a pseudo-square modulo n. This problem is believed to be computationally difficult, and is the basis for a number of cryptosystems.

The following theorem informally shows for every n, if the quadratic residusosity is hard to compute at all then it is hard to distinguish between squares and non-squares for almost everywhere.

Theorem 2.5.3 *[93, 94]: Let $S \subset \{ns.t.n = pq, p, q, \ primes\}$ If there exists a probabilistic polynomial-time algorithm O such that for $n \in S$,*

$$prob(O(n,x) \text{ decides correctly whether } x \in J_n^{+1}) > \frac{1}{2} + \epsilon \ , \tag{2.1}$$

where this probability is taken over the choice of $x \in J_n^{+1}$ and O's random choices, then there exists a probabilistic algorithm B with running time polynomial in $\epsilon^{-1}, \delta^{-1}$ and $|n|$ such that for all $n \in S$, for all $x \in J_n^{+1}$,

$$prob(B(x,n) \text{ decides correctly whether } x \in Q_n | x \in J_n^{+1}) > 1 - \delta \ , \tag{2.2}$$

where this probability is taken over the random coin tosses of B.

Namely, a probabilistic polynomial-time bounded adversary can not do better (except by a smaller than any polynomial advantage) than guess at random whether $x \in J_n$ is a square mod n, if quadratic residuosity problem is not in polynomial time.

This suggests immediately the following set of predicates: Let

$$QR_{n,z}(x) = \left\{ \begin{array}{l} 0 \text{ if } x \text{ is a square mod } n \\ 1 \text{ if } x \text{ is a non-square mod } n \end{array} \right\}$$

where $QR_{n,z} : J_n^{+1} \rightarrow \{0,1\}$ and $I_k = \{n\#z \mid n = pq, \ |p| = |q| = \frac{k}{2}, \ p \text{ and } q \text{ primes}, \ (\mathbf{J}_n(z)) = +1, \ z$ a non-square mod $n\}$. It is clear that $QR = \{QR_{n,z}\}$ is a set of trapdoor predicates where the trapdoor information associated with every index $< n, z >$ is the factorization $< p, q >$. Lets check this explicitly.

(1) To select pairs (i, t_i) at random, first pick two primes, p and q, of size $\left|\frac{k}{2}\right|$ at random, determining n. Next, search until we find a non-square z in Z_n^* with Jacobi symbol $+1$. The pair we have found is then $(< n, z >, < p, q >)$. We already know how to do all of these operations in expected polynomial time .

(2) Follows from the existence of the following algorithm to select elements out of $D_{n,z}^v$:

- To select $x \in D_{n,z}^0$, let $x = y^2 \bmod n$ where y is an element of Z_n^* chosen at random.
- To select $x \in D_{n,z}^1$, let $x = zy^2 \bmod n$ where y is an element of Z_n^* chosen at random.

(3) To compute $QR_{n,z}(x)$ given $< p, q >$, we compute $(\mathbf{J}_p(x))$ and $(\frac{x}{q})$. If both are $+1$ then $QR_{n,z}(x)$ is 0, otherwise it is 1.

(4) This follows from the Quadaratic Residuosity Assumption and the above theorem.

A Set of Trapdoor Predicates Based on the RSA Assumption

Define $B_{n,e}(x) =$ the least significant bit of $x^d \bmod n$ for $x \in Z_n^*$ where $ed = 1 \bmod \phi(n)$. Then, to select uniformly an $x \in Z_n^*$ such that $B_{n,e}(x) = v$ simply select a $y \in Z_n^*$ whose least significant bit is v and set $x = y^e \bmod n$. Given d it is easy to compute $B_{n,e}(x) =$ least significant bit of $x^d \bmod n$.

The security of this construction follows trivially from the definition of collection of hard core predicates for the RSA collection of functions.

C H A P T E R 3

Pseudo-random bit generators

In this chapter, we discuss the notion of pseudo-random generators. Intuitively, a $PSRG$ is a deterministic program used to generate *long* sequence of bits which *looks like random sequence*, given as input a *short random sequence* (the input seed).
The notion of $PSRG$ has applications in various fields:

Cryptography:
: In the case of private key encryption, Shannon showed (see lecture 1) that the length of the clear text should not exceed the length of the secret key, that is, the two parties have to agree on a very long string to be used as the secret key. Using a $PSRG$ G, they need to agree only on a short seed r, and exchange the message $G(r) \bigoplus m$.

Algorithms Design:
: An algorithm that uses a source of random bits, can manage with a shorter string, used as a seed to a $PSRG$.

Complexity Theory:
: Given a probabilistic algorithm, an important question is whether we can make it deterministic. Using the notion of a $PSRG$ we can prove, assuming the existence of one-way function that $BPP \subseteq \cap_\epsilon DTIME(2^{n^\epsilon})$

In this chapter we will define good pseudo random number generators and give constructions of them under the assumption that one way functions exist.

We first ask where do can we actually find **truly** random bit sequences. [1]

3.0.2 Generating Truly Random bit Sequences

Generating a one-time pad (or, for that matter, any cryptographic key) requires the use of a "natural" source of random bits, such as a coin, a radioactive source or a noise diode. Such sources are absolutely essential for providing the initial secret keys for cryptographic systems.

However, many natural sources of random bits may be defective in that they produce *biased* output bits (so that the probability of a one is different than the probability of a zero), or bits which are *correlated* with

[1] some of the preliminary discussion in the following three subsections is taken from Rivest's survey article on cryptography which appears in the handbook of computer science

each other. Fortunately, one can remedy these defects by suitably processing the output sequences produced by the natural sources.

To turn a source which supplies biased but uncorrelated bits into one which supplies unbiased uncorrelated bits, von Neumann proposed grouping the bits into pairs, and then turning 01 pairs into 0's, 10 pairs into 1's, and discarding pairs of the form 00 and 11 [193]. The result is an unbiased uncorrelated source, since the 01 and 10 pairs will have an identical probability of occurring. Elias [73] generalizes this idea to achieve an output rate near the source entropy.

Handling a correlated bit source is more difficult. Blum [37] shows how to produce unbiased uncorrelated bits from a biased correlated source which produces output bits according to a known finite Markov chain.

For a source whose correlation is more complicated, Santha and Vazirani [170] propose modeling it as a *slightly random source*, where each output bit is produced by a coin flip, but where an adversary is allowed to choose *which* coin will be flipped, from among all coins whose probability of yielding "Heads" is between δ and $1-\delta$. (Here δ is a small fixed positive quantity.) This is an extremely pessimistic view of the possible correlation; nonetheless U. Vazirani [190] shows that if one has *two, independent*, slightly-random sources X and Y then one can produce "almost independent" ϵ-biased bits by breaking the outputs of X and Y into blocks $\mathbf{x}, \mathbf{y}$ of length $k = \Omega(1/\delta^2 \log(1/\delta) \log(1/\epsilon))$ bits each, and for each pair of blocks $\mathbf{x}, \mathbf{y}$ producing as output the bit $\mathbf{x} \cdot \mathbf{y}$ (the inner product of $\mathbf{x}$ and $\mathbf{y}$ over $GF(2)$). This is a rather practical and elegant solution. Chor and Goldreich [53] generalize these results, showing how to produce independent ϵ-biased bits from even worse sources, where some output bits can even be completely determined.

These results provide effective means for generating truly random sequences of bits—an essential requirement for cryptography—from somewhat defective natural sources of random bits.

3.0.3 Generating Pseudo-Random Bit or Number Sequences

The one-time pad is generally impractical because of the large amount of key that must be stored. In practice, one prefers to store only a short random key, from which a long pad can be produced with a suitable cryptographic operator. Such an operator, which can take a short *random* sequence x and deterministically "expand" it into a *pseudo-random* sequence y, is called a *pseudo-random sequence generator*. Usually x is called the *seed* for the generator. The sequence y is called "pseudo-random" rather than random since not all sequences y are possible outputs; the number of possible y's is at most the number of possible seeds. Nonetheless, the intent is that for all practical purposes y should be indistinguishable from a truly random sequence of the same length.

It is important to note that the use of pseudo-random sequence generator reduces *but does not eliminate* the need for a natural source of random bits; the pseudo-random sequence generator is a "randomness expander", but it must be given a truly random seed to begin with.

To achieve a satisfactory level of cryptographic security when used in a one-time pad scheme, the output of the pseudo-random sequence generator must have the property that an adversary who has seen a portion of the generator's output y must remain unable to efficiently predict other unseen bits of y. For example, note that an adversary who knows the ciphertext C can guess a portion of y by correctly guessing the corresponding portion of the message M, such as a standardized closing "Sincerely yours,". We would not like him thereby to be able to efficiently read other portions of M, which he could do if he could efficiently predict other bits of y. Most importantly, the adversary should not be able to efficiently infer the seed x from the knowledge of some bits of y.

How can one construct secure pseudo-random sequence generators?

Classical Pseudo-random Generators are Unsuitable

Classical techniques for pseudo-random number generation [116, Chapter 3] which are quite useful and effective for Monte Carlo simulations are typically unsuitable for cryptographic applications. For example, *linear* feedback shift registers [100] are well-known to be cryptographically insecure; one can solve for the feedback pattern given a small number of output bits.

Linear congruential random number generators are also insecure. These generators use the recurrence

$$X_{i+1} = aX_i + b \pmod{m} \tag{3.1}$$

to generate an output sequence $\{X_0, X_1, \ldots\}$ from secret parameters a, b, and m, and starting point X_0. It is possible to infer the secret parameters given just a few of the X_i [150]. Even if only a fraction of the bits of each X_i are revealed, but a, b, and m are known, Frieze, Håstad, Kannan, Lagarias, and Shamir show how to determine the seed X_0 (and thus the entire sequence) using the marvelous *lattice basis reduction* (or "L^3") algorithm of Lenstra, Lenstra, and Lovász [81, 123].

As a final example of the cryptographic unsuitability of classical methods, Kannan, Lenstra, and Lovasz [114] use the L^3 algorithm to show that the binary expansion of any algebraic number y (such as $\sqrt{5} = 10.001111000110111\ldots$) is insecure, since an adversary can identify y exactly from a sufficient number of bits, and then extrapolate y's expansion.

3.0.4 Provably Secure Pseudo-Random Generators: Brief overview

This section provides a brief overview of the history of the modern history of pseudo random bit generators. Subsequent section define these concepts formally and give constructions.

The first pseudo-random sequence generator proposed for which one can prove that it is impossible to predict the next number in the sequence from the previous numbers assuming that it is infeasible to invert the RSA function is due to Shamir [177]. However, this scheme generates a sequence of *numbers* rather than a sequence of *bits*, and the security proof shows that an adversary is unable to predict the next *number*, given the previous numbers output. This is not strong enough to prove that, when used in a one-time pad scheme, each *bit* of the message will be well-protected.

Blum and Micali [39] introduced the first method for designing provably secure pseudo-random *bit* sequence generators, based on the use of one-way predicates. They call a pseudo-random bit generator secure if an adversary cannot guess the next bit in the sequence from the prefix of the sequence, better than guessing at random. Blum and Micali then proposed a particular generator based on the difficulty of computing discrete logarithms. Blum, Blum, and Shub [34] propose another generator, called the *squaring generator*, which is simpler to implement and is provably secure assuming that the quadratic residuosity problem is hard. Alexi, Chor, Goldreich, and Schnorr [6] show that the assumption that the quadratic residuosity problem is hard can be replaced by the weaker assumption that factoring is hard. A related generator is obtained by using the RSA function. Kaliski shows how to extend these methods so that the security of the generator depends on the difficulty of computing elliptic logarithms; his techniques also generalize to other groups [112, 113]. Yao [194] shows that the pseudo-random generators defined above are *perfect* in the sense that no probabilistic polynomial-time algorithm can guess with probability greater by a non-negligible amount than 1/2 whether an input string of length k was randomly selected from $\{0,1\}^k$ or whether it was produced by one of the above generators. One can rephrase this to say that a generator that passes the next-bit test is perfect in the sense that it will *pass all polynomial-time statistical tests*. The Blum-Micali and Blum-Blum-Shub generators, together with the proof of Yao, represent a major achievement in the development of provably secure cryptosystems. Impagliazzo, Luby, Levin and Håstad show that actually the existence of a one-way function is equivalent to the existence of a pseudo random bit generator which passes all polynomial time statistical tests.

3.1 Definitions

Definition 3.1.1 Let X_n, Y_n be probability distributions on $\{0,1\}^n$ (That is, by $t \in X_n$ we mean that $t \in \{0,1\}^n$ and it is selected according to the distribution X_n). We say that $\{X_n\}$ is **poly-time indistin-**

guishable from $\{Y_n\}$ if $\forall PTM\ A$, $\forall$ polynomial Q, $\exists n_0$, s.t $\forall n > n_0$,

$$|\Pr_{t \in X_n}(A(t) = 1) - \Pr_{t \in Y_n}(A(t) = 1)| < \frac{1}{Q(n)}$$

i.e., for sufficiently long strings, no PTM can tell whether the string was sampled according to X_n or according to Y_n.
Intuitively, Pseudo random distribution would be a indistinguishable from the uniform distribution. We denote the uniform probability distribution on $\{0,1\}^n$ by U_n. That is, for every $\alpha \in \{0,1\}^n$, $\Pr_{x \in U_n}[x = \alpha] = \frac{1}{2^n}$.

Definition 3.1.2 We say that $\{X_n\}$ is **pseudo random** if it is poly-time indistinguishable from $\{U_n\}$. That is, $\forall PTM\ A$, $\forall$ polynomial Q, $\exists n_0$, such that $\forall n > n_0$,

$$|\Pr_{t \in X_n}[A(t) = 1] - \Pr_{t \in U_n}[A(t) = 1]| < \frac{1}{Q(n)}$$

Comments:

The algorithm A used in the above definition is called a polynomial time *statistical test.* (Knuth, vol. 2, suggests various examples of statistical tests). It is important to note that such a definition cannot make sense for a single string, as it can be drawn from either distribution.

If $\exists A$ and Q such that the condition in definition 2 is violated, we say that X_n fails the test A.

Definition 3.1.3 A polynomial time deterministic program $G : \{0,1\}^k \to \{0,1\}^{\hat{k}}$ is a pseudo-random generator (PSRG) if the following conditions are satisfied.

1. $\hat{k} > k$
2. $\{G_{\hat{k}}\}_{\hat{k}}$ is pseudo-random where $G_{\hat{k}}$ is the distribution on $\{0,1\}^{\hat{k}}$ obtained as follows: to get $t \in G_{\hat{k}}$:

   pick $x \in U_k$

   set $t = G(x)$

That is, $\forall PTMA, \forall$ polynomial $Q, \forall$ sufficiently large k,

$$|\Pr_{t \in G_{\hat{k}}}(A(t) = 1) - \Pr_{t \in U_{\hat{k}}}(A(t) = 1)| < \frac{1}{Q(\hat{k})} \tag{3.2}$$

3.2 The Existence Of A Pseudo-Random Generator

Next we prove the existence of $PSRG$'s, if length-preserving one way permutations exist. It has been shown that if one-way functions exist (without requiring them to be length-preserving permutations) then one-way functions exist, but we will not show this here.

Theorem 3.2.1 *Let $f : \{0,1\}^* \to \{0,1\}^*$ be a length preserving one-way permutation. Then*

1. *$\exists\, PSRG\quad G : \{0,1\}^k \to \{0,1\}^{k+1}$ (such G is called an extender).*
2. *$\forall$ polynomial Q, $\exists\, PSRG\quad G^Q : \{0,1\}^k \to \{0,1\}^{Q(k)}$.*

Proof:
Proof of 1: Let f be as above. Let B be the *hard core bit* of f. (that is, B is a boolean predicate, $B : \{0,1\}^* \to \{0,1\}$, s.t it is efficient to compute $B(x)$ given x, but given only $f(x)$, it is hard to compute $B(x)$ with probability greater than $\frac{1}{2} + \epsilon$ for non-negligible ϵ.) Recall that we showed last class that every OWF $f(x)$ can be converted into $f_1(x, r)$ for which $B'(x, r) = \sum_1^{|x|} x_i r_i \bmod 2$ is a hard core bit. For notational ease, assume that B is already a hard-core bit for f.

Define

$$G^1(x) = f(x) \circ B(x)$$

($\circ$ denotes the string concatenation operation). We will prove that $G^1(x)$ has the required properties. Clearly, G^1 is computed in poly-time, and for $|x| = k$, $|G^1(x)| = k + 1$. It remains to show that the distribution $\{G^1_{k+1}\}$ is pseudo random.

Intuition : Indeed, knowing $f(x)$ should not help us to predict $B(x)$. As f is a permutation, $f(U_k)$ is uniform on $\{0,1\}^k$, and any separation between U_{k+1} and G_{k+1} must caused by the hard core bit. We would like to show that any such separation would enable us to predict $B(x)$ given only $f(x)$ and obtain a contradiction to B being a hard core bit for f.

We proceed by contradiction: Assume that (G is not good) $\exists$ statistical test A, polynomial Q s.t

$$\Pr_{t \in G_{k+1}}(A(t) = 1) - \Pr_{t \in U_{k+1}}(A(t) = 1) > \frac{1}{Q(k+1)}$$

(Note that we have dropped the absolute value from the inequality 3.2. This can be done wlog. We will later see what would change if the the other direction of the inequality were true).

Intuitively, we may thus interpret that if A answers 1 on a string t it is more likely that t is drawn from distribution G_{k+1}, and if A answers 0 on string t that it is more likely that t is drawn from distribution U_{k+1}.

We note that the probability that $A(f(x) \circ b)$ returns 1, is the sum of the weighted probability that A returns 1 conditioned on the case that $B(x) = b$ and conditioned on the case $B(x) = 1$. By, the assumed separation above, we get that it is more likely that $A(f(x) \circ b)$ will return 1 when $b = B(x)$. This easily translates to an algorithm for predicting the hard core bit of $f(x)$.

Formally, we have

$$\begin{aligned}
\Pr_{x \in U_k, b \in U_1}[A(f(x) \circ b) = 1] &= \Pr[A(f(x) \circ b) = 1 \mid b = B(x)] \cdot \Pr[b = B(x)] \\
&\qquad + \Pr[A(f(x) \circ b) = 1 \mid b = \overline{B(x)}] \cdot \Pr[b = \overline{B(x)}] \\
&= \tfrac{1}{2}(\alpha + \beta)
\end{aligned}$$

where $\alpha = \Pr[A(f(x) \circ b) = 1 \mid b = B(x)]$ and $\beta = \Pr[A(f(x) \circ b) = 1 \mid b = \overline{B(x)}]$.

From the assumption we therefore get

$$\begin{aligned}
\Pr_{x \in U_k}[A(f(x) \circ B(x)) = 1] - \Pr_{x \in U_k}[A(f(x) \circ b) = 1] &= \alpha - \tfrac{1}{2}(\alpha + \beta) \\
&= \tfrac{1}{2}(\alpha - \beta) \\
&> \tfrac{1}{Q(k)}.
\end{aligned}$$

We now exhibit a polynomial time algorithm A' that on input $f(x)$ computes $B(x)$ with success probability significantly better than $1/2$.
A' takes as input $f(x)$ and outputs either a 0 or 1.

1. choose $b \in \{0,1\}$
2. run $A(f(x) \circ b)$
3. If $A(f(x) \circ b) = 1$, output b, otherwise output $\overline{b}$.

Notice that, when dropping the absolute value from the inequality 3.2, if we take the second direction we just need to replace b by $\overline{b}$ in the definition of A'.

Claim 3.2.2 $\Pr[A'(f(x) = B(x))] > \frac{1}{2} + \frac{1}{Q(k)}$.

Proof:

$$\begin{aligned}
\Pr[A'(f(x) = b)] \; & \Pr[A(f(x) \circ b) = 1 \mid b = B(x)] \Pr[b = B(x)) \\
& \Pr[A(f(x) \circ b) = 0 | b = \overline{B(x)}) \Pr[b = \overline{B(x)}] \\
& \qquad +\tfrac{1}{2}\alpha + \tfrac{1}{2}(1-\beta) \\
& = \tfrac{1}{2} + \tfrac{1}{2}(\alpha - \beta) \\
& > \tfrac{1}{2} + \tfrac{1}{Q(k)}.
\end{aligned}$$

∎

This contradicts the hardness of computing $B(x)$. It follows that G^1 is indeed a $PSRG$.
Proof of 2*:* Given a $PSRG$ G that expands random strings of length k to pseudo-random strings of length $k+1$, we need to show that, $\forall$ polynomial Q, $\exists \, PSRG \, G^Q : \{0,1\} \to \{0,1\}^{Q(k)}$. We define G^Q by first using G $Q(k)$ times as follows:

$$\begin{array}{lcl}
x & \to G \to & f(x) \circ B(x) \\
f(x) \circ B(x) & \to G \to & f(f(x)) \circ B(f(x)) \\
f^2(x) \circ B(f(x)) & \to G \to & f^3(x) \circ B(f^2(x)) \\
 & \bullet & \\
 & \bullet & \\
 & \bullet & \\
f^{Q(k)-2}(x) \circ B(f^{Q(k)-1}(x)) & \to G \to & f^{Q(k)}(x) \circ B(f^{Q(k)-1}(x))
\end{array}$$

The output of $G^Q(x)$ is the concatenation of the last bit from each string *i.e.*,

$$G^Q(x) = B(x) \circ B(f(x)) \circ \cdots \circ B(f^{Q(k)-1}(x) =$$

$$b_1^G(x) \circ b_2^G(x) \circ \cdots \circ b_{Q(|x|)}^G(x)$$

Clearly, G^Q is poly-time and it satisfies the length requirements. We need to prove that the distribution generated by G^Q, $G^Q(U_k)$, is poly-time indistinguishable from $U_{Q(k)}$. We proceed by contradiction, (and show that it implies that G is not $PSRG$)
If G_{Qk} is not poly-time indistinguishable from $U_{Q(k)}$, $\exists$ statistical test A, and $\exists$ polynomial P, s.t.

$$\Pr_{t \in G_{Q(k)}}(A(t) = 1) - \Pr_{t \in U_{Q(k)}}(A(t) = 1) > \frac{1}{P(k)}$$

(As before we omit the absolute value). We now define a sequence $D_1, D_2, ..., D_{Q(k)}$ of distributions on $\{0,1\}^{Q(k)}$, s.t. D_1 is uniform (i.e. strings are random), $D_{Q(k)} = G_{Q(k)}$, and the intermediate D_i's are distributions composed of concatenation of random followed by pseudorandom distributions. Specifically,

$t \in D_1$ is obtained by letting	$t = s$	where $s \in U_{Q(k)}$
$t \in D_2$ is ontained by letting	$t = s \circ B(x)$	where $s \in U_{Q(k)-1}, x \in U_k$
$t \in D_3$ is ontained by letting	$t = s \circ B(x) \circ B(f(x))$	where $s \in U_{Q(k)-2}, x \in U_k$
	•	
	•	
	•	
$t \in D_{Q(k)}$ is ontained by letting	$t = B(x) ... \circ B(f^{Q(k)-1}(x))$	where $x \in U_k$

Since the sequence 'moves' from $D_1 = U_{Q(k)}$ to $D_{Q(k)} = G_{Q(k)}$, and we have an algorithms A that distinguishes between them, there must be two successive distributions between which A distinguishes.
i.e. $\exists\ i$, $1 \leq i \leq Q(k)$, s.t.

$$\Pr_{t \in D_i}(A(t) = 1) - \Pr_{t \in D_{i+1}}(A(t) = 1) > \frac{1}{P(k)Q(k)}$$

We now present a poly-time algorithm A' that distinguishes between U_{k+1} and G_{k+1}, with success probability significantly better than $\frac{1}{2}$, contradicting the fact that G is a $PSRG$.

A' works as follows on input $\alpha = \alpha_1 \alpha_2 ... \alpha_{k+1} = \alpha' \circ b$

1. Choose $1 \leq i \leq q(k)$ at random.
2. Let
$$t = \gamma_1 \circ ... \circ \gamma_{Q(k)-i-1} \circ b \circ b_1^G(\alpha') \circ b_2^G(\alpha') \circ ... \circ b_i^G(\alpha')$$
where the γ_j are chosen randomly.
(Note that $t \in D_i$ if $\alpha' \circ b \in U_{k+1}$, and that $t \in D_{i+1}$ if $\alpha' \circ b \in G_{k+1}$.)
3. We now run $A(t)$. If we get 1, A' returns 0 If we get 0, A' returns 1
(i.e if A returns 1, it is interpreted as a vote for D_i and therefore for $b \neq B(\alpha')$ and $\alpha \in U_{k+1}$. On the other hand, if A returns 0, it is interpreted as a vote for D_{i+1} and therefore for $b = B(\alpha')$ and $\alpha \in G_{k+1}$.)

It is immediate that:

$$\Pr_{\alpha \in U_{k+1}}(A'(\alpha) = 1) - \Pr_{\alpha \in G_{k+1}}(A'(\alpha) = 1) > \frac{1}{P(k)Q^2(k)}$$

The extra $\frac{1}{Q(k)}$ factor comes form the random choice of i. This violates the fact that G was a pseudo random generator as we proved in part 1. This is a contradiction

∎

3.3 Next Bit Tests

If a pseudo-random bit sequence generator has the property that it is difficult to predict the next bit from previous ones with accuracy greater than $\frac{1}{2}$ by a non-negligible amount in time polynomial in the size of the seed, then we say that the generator *passes the "next-bit" test.*

Definition 3.3.1 A *next bit test* is a special kind of statistical test which takes as input a prefix of a sequence and outputs a prediction of the next bit.

Definition 3.3.2 A (discrete) probability distribution on a set S is a function $D : S \to [0,1] \subset \mathbf{R}$ so that $\sum_{s \in S} D(s) = 1$. For brevity, probability distributions on $\{0,1\}^k$ will be subscripted with a k. The notation $x \in X_n$ means that x is chosen so that $\forall z \in \{0,1\}^n Pr[x = z] = X_n(z)$. In what follows, U_n is the uniform distribution.

Recall the definition of a pseudo-random number generator:

Definition 3.3.3 A *pseudo-random number generator (PSRG)* is a polynomial time deterministic algorithm so that:

1. if $|x| = k$ then $|G(x)| = \hat{k}$
2. $\hat{k} > k$,
3. $G_{\hat{k}}$ is pseudo-random[2], where $G_{\hat{k}}$ is the probability distribution induced by G on $\{0,1\}^{\hat{k}}$.

Definition 3.3.4 We say that a pseudo-random generator *passes the next bit test* A if for every polynomial Q there exists ,an integer k_0 such that for all $\hat{k} > k_0$ and $p < \hat{k}$

$$\Pr_{t \in G_{\hat{k}}} [A(t_1 t_2 \ldots t_p) = t_{p+1}] < \frac{1}{2} + \frac{1}{Q(k)}$$

Theorem 3.3.5 *G passes all next bit tests $\Leftrightarrow$ G passes all statistical tests.*

Proof:

($\Leftarrow$) Trivial.

($\Rightarrow$) Suppose, for contradiction, that G passes all next bit test but fails some statistical test A. We will use A to construct a next bit test A' which G fails. Define an operator $\odot$ on probability distributions so that $[X_n \odot Y_m](z) = X_n(z_n) \cdot Y_m(z_m)$ where $z = z_n \circ z_m, |z_n| = n, |z_m| = m$ ($\circ$ is concatenation). For $j \leq \hat{k}$ let $G_{j,\hat{k}}$ be the probability distribution induced by $G_{\hat{k}}$ on $\{0,1\}^j$ by taking prefixes. (That is $G_{j,\hat{k}}(x) = \sum_{z \in \{0,1\}^{\hat{k}}, z \text{ extends } x} G_{\hat{k}}(z)$.)

Define a sequence of distributions $H_i = G_{i,\hat{k}} \odot U_{\hat{k}-i}$ on $\{0,1\}^{\hat{k}}$ of "increasing pseudo-randomness." Then $H_0 = U_{\hat{k}}$ and $H_{\hat{k}} = G_{\hat{k}}$. Because G fails A, A can differentiate between $U_{\hat{k}} = H_0$ and $G_{\hat{k}} = H_{\hat{k}}$; that is, $\exists Q \in \mathcal{Q}[x]$ so that $|Pr_{t \in H_0}[A(t) = 1] - Pr_{t \in H_{\hat{k}}}[A(t) = 1]| > \frac{1}{Q(k)}$. We may assume without loss of generality that $A(t) = 1$ more often when t is chosen from $U_{\hat{k}}$ (otherwise we invert the output of A) so that we may drop the absolute value markers on the left hand side. Then $\exists i, 0 \leq i \leq \hat{k} - 1$ so that $Pr_{t \in H_i}[A(t) = 1] - Pr_{t \in H_{i+1}}[A(t) = 1] > \frac{1}{\hat{k}Q(k)}$.

The next bit test A' takes $t_1 t_2 \ldots t_i$ and outputs a guess for t_{i+1}. A' first constructs

$$s_0 = t_1 t_2 \ldots t_i 0 r_{i+2} r_{i+3} \ldots r_{\hat{k}}$$

$$s_1 = t_1 t_2 \ldots t_i 1 \hat{r}_{i+2} \hat{r}_{i+3} \ldots \hat{r}_{\hat{k}}$$

where r_j and $\hat{r}_j$ are random bits for $i + 2 \leq j \leq \hat{k}$. A' then computes $A(s_0)$ and $A(s_1)$.

[2] A pseudo-random distribution is one which is polynomial time indistinguishable from $U_{\hat{k}}$

If $A(s_0) = A(s_1)$, then A' outputs a random bit.

If $0 = A(s_0) = \overline{A(s_1)}$, then A' outputs 0.

If $1 = A(s_0) = \overline{A(s_1)}$, then A' outputs 1.

Claim 3.3.6 By analysis similar to that done in the previous lecture, $Pr[A'(t_1 t_2 \dots t_i) = t_{i+1}] > \frac{1}{2} + \frac{1}{kQ(k)}$.

Thus we reach a contradiction: A' is a next bit test that G fails, which contradicts our assumption that G passes all next bit tests.

∎

3.4 Examples of Pseudo-Random Generators

Each of the one way functions we have discussed induces a pseudo-random generator. Listed below are these generators (including the Blum/Blum/Shub generator which will be discussed afterwards) and their associated costs. See [39, 34, 164].

Name	One way function	Cost of computing one way function	Cost of computing j^{th}bit of generator
RSA	$x^e \bmod n, n = pq$	k^3	jk^3
Rabin	$x^2 \bmod n, n = pq$	k^2	jk^2
Blum/Micali	$\mathrm{EXP}(p, g, x)$	k^3	jk^3
Blum/Blum/Shub	(see below)	k^2	$\max(k^2 \log j, k^3)$

3.4.1 Blum/Blum/Shub Pseudo-Random Generator

The Blum/Blum/Shub pseudo-random generator uses the (proposed) one way function $g_n(x) = x^2 \bmod n$ where $n = pq$ for primes p and q so that $p \equiv q \equiv 3 \bmod 4$. In this case, the squaring endomorphism $x \mapsto x^2$ on $\mathbf{Z}_n^*$ restricts to an isomorphism on $(\mathbf{Z}_n^*)^2$, so g_n is a permutation on $(\mathbf{Z}_n^*)^2$. (Recall that every square has a unique square root which is itself a square.)

Claim 3.4.1 The least significant bit of x is a hard bit for the one way function g_n.

The j^{th} bit of the Blum/Blum/Shub generator may be computed in the following way:

$$B(x^{2^j} \bmod n) = B(x^\alpha \bmod m)$$

where $\alpha \equiv 2^j \bmod \phi(n)$. If the factors of n are known, then $\phi(n) = (p-1)(q-1)$ may be computed so that α may be computed prior to the exponentiation. $\alpha = 2^j \bmod \phi(n)$ may be computed in $O(k^2 \log j)$ time and x^α may be be computed in k^3 time so that the computation of $B(x^{2^j})$ takes $O(\max(k^3, k^2 \log j))$ time.

An interesting feature of the Blum/Blum/Shub generator is that if the factorization of n is known, the $2^{\sqrt{n}}th$ bit can be generated in time polynomial in $|n|$. The following question can be raised: let $G^{BBS}(x, p, q) = B(f^{2^{\sqrt{n}}}(x)) \circ \dots \circ B(f^{2^{\sqrt{n}}+2k}(x))$ for $n = pq$ and $|x| = k$. Let G_{2k}^{BBS} be the distribution induced by G^{BBS} on $\{0,1\}^{2k}$.

Open Problem 3.4.2 Is this distribution G_{2k}^{BBS} pseudo-random? Namely, can you prove that

$$\forall Q \in \mathbf{Q}[x], \forall PTM\ A, \exists k_0, \forall k > k_0 |Pr_{t \in G_{2k}^{BBS}}[A(t) = 1] - Pr_{t \in U_{2k}}[A(t) = 1]| < \frac{1}{Q(2k)}$$

The previous proof that G is pseudo-random doesn't work here because in this case the factorization of n is part of the seed so no contradiction will be reached concerning the difficulty of factoring.

More generally,

Open Problem 3.4.3 Pseudo-random generators, given seed x, implicitly define an infinite string $g_1^x g_2^x \ldots$. Find a pseudo-random generator so that the distribution created by restricting to any polynomially selected subset of bits of g^x is pseudo-random. By polynomially selected we mean examined by a polynomial time machine which can see g_i^x upon request for a polynomial number of i's (the machine must write down the i's, restricting $|i|$ to be polynomial in $|x|$).

3.5 Concrete security and practical constructions

As discussed above, a generator is a deterministic algorithm G that takes as input a seed s of length k, and outputs a string $G(s)$ of some length m where $m > k$. The motivation comes from the fact that good quality randomness is hard or expensive to get. Thus a useful paradigm is to get enough randomness to specify a seed s, and then run a deterministic algorithm G, namely the generator, to expand this seed into a longer sequence $G(s)$ which has good "pseudorandomness" properties. These properties must suffice for various applications. These applications might include simulations or other kinds of probabilistic algorithms. Or they might be cryptographic; for example, replace a one-time pad with a scheme in which the shared key is a seed and we use the output of the generator on the seed in place of the pad.

We would like to have a notion of security for pseudorandom bit generators that will guarantee the security of applications like the above. We would also like constructions of generators that meet such a notion. Roughly, we want the output of the generator to "look random" in some way. Of course, it cannot be fully random, since it has only s bits of entropy in it. So we seek some appropriate notion of pseudorandomness. There are roughly two kinds of properties we want. The first is that the output sequences of the generator have good statistical characteristics. This means, for example, that about half the bits are typically one, and the rest 0; that the sequence 11 shows up about $m/4$ times, and so on. The second is unpredictability. Given a prefix of $G(s)$, it should not be easy to predict, or compute, the rest of $G(s)$.

Yet, listing some desirable properties in this rough way does not suffice to give a clear picture of the goal. As we have seen before, we are getting some list of properties that should be necessary for pseudorandomness, but there is no reason to think they are sufficient. We want instead a definition in the style of our definitions for PRFs or PRPs. Above we discussed this in the complexity-theoretic, or asymptotic setting. Here we provide the concrete version. The ideas are the same; it is just a question of how we measure complexity.

Then we discuss how PSRGs might be built in practice from block ciphers (discussed in Chapter 4), the latter being modeled as PRFs (discussed in Chapter 5).

3.5.1 The notion of security for PRGs

We want that when s is chosen at random, the string $G(s)$ is approximately random, and are trying to figure out an appropriate way to capture what this "approximately" means. As a start, consider some application A that needs a lot of randomness. This might be, say, a simulation program. View A as taking a long random string y as input, and then executing to produce some result. For simplicity, say A outputs just a bit, 0 or 1. Now, consider running A on an input y that is chosen differently: $y = G(s)$ where s was chosen at random. A is not "told" in any way that its input is no longer truly random; it is just fed this different y. If G is a "good" pseudorandom bit generator, what we would like is that A's operation is no different under $y = G(s)$ than under a random y. In other words, $G(s)$ was "random enough" that the simulation done by A goes through just as well, and A returns a 0 or 1 with just about the same probability it would if y were random.

Thus, G is a good pseudorandom bit generator if an application using as source of randomness $y = G(s)$ where s was random, performs equally well as if the same application used instead a truly random sequence of bits y. In other words, in practice, we can substitute any use of a long random sequence by a use of $G(s)$ where s was random.

Take this one step further, and view the application as a test, or adversary. It actively *tries* to behave differently in the following two cases: its input is a truly random string y, and its input is $y = G(s)$ where s was random. That is, it tries to distinguish these two settings. If it does not succeed, G is a good pseudorandom bit generator. Certainly, if even an adversary cannot tell $G(s)$ for random s apart from a truly random string, neither can some disinterested application.

As usual, in this first high level view, we are pushing under the rug the computational and quantitative aspects of the notion. As the adversary A invests more time into its distinguishing task, it will do better. For example, if it has 2^k time, it can try all possible seeds, and quite easily tell whether its given input y is an output of G on some seed or not. But that's OK. All we need is that adversaries that don't use too much time don't do too well: the outputs of the generator look random as long as you don't have too many computing cycles available with which to analyze them.

Let us now provide the formal definition of security. Let $G\colon \{0,1\}^k \to \{0,1\}^l$ be a function, where $m > k$. We want to associate to G an insecurity function $\mathbf{InSec}^{\mathrm{prg}}_G(\cdot)$ whose value at t is the maximum possible probability of being able to "break" G as a pseudorandom bit generator, when the attacker is limited to time t. That is, consider the best attacking strategy that can be implemented by an algorithm running for t steps; the probability it "breaks" G is $\mathbf{InSec}^{\mathrm{prg}}_G(t)$. What we now need to do now is pin down exactly what we mean by "break".

The attacker A (often called a statistical test) gets input a string y of length m, and outputs a bit. It knows that y is either a random string, or $y = G(s)$ where s was random, but is not told which. It outputs a bit, and we view an output of 1 as an indication that A is saying "I think y is pseudorandom."

Definition 3.5.1 Let $G\colon \{0,1\}^k \to \{0,1\}^m$ be a function, where $m > k$, and let A be an attacking algorithm. Let

$$\begin{aligned} \mathbf{Succ}^{\mathrm{prg}}_G(A) &= \mathbf{P}\left[\, A(y) = 1 \;:\; s \stackrel{R}{\leftarrow} \{0,1\}^k \;;\; y \leftarrow G(s) \,\right] \\ \mathbf{Succ}^{\mathrm{prg}}_{\mathrm{Rand}}(A) &= \mathbf{P}\left[\, A(y) = 1 \;:\; y \stackrel{R}{\leftarrow} \{0,1\}^m \,\right] . \end{aligned}$$

We call the difference between them the "advantage" of A:

$$\mathbf{Adv}^{\mathrm{prg}}_G(A) \;=\; \mathbf{Succ}^{\mathrm{prg}}_G(A) - \mathbf{Succ}^{\mathrm{prg}}_{\mathrm{Rand}}(A) .$$

Finally for any t we let

$$\mathbf{InSec}^{\mathrm{prg}}_G(t) \;=\; \max_A \{\, \mathbf{Adv}^{\mathrm{prg}}_G(A) \,\}$$

where the maximum is over all A running in time t. ▮

Informally, we call G a pseudorandom bit generator if $\mathbf{InSec}^{\mathrm{prg}}_G(t)$ is small for reasonably high values of q.

3.5.2 Possible constructions of a PRG from a PRF: Which ones work?

In practice, pseudorandom generators are built in various ways. For example they might use Linear Feedback Shift Registers (LFSRs). These kinds of constructions typically have good statistical properties, but are predictable. Attacks are known on many such constructions. They don't meet the kind of notion of security we defined above.

Better generators can be built based on cryptographic primitives. Specifically we can use block ciphers, thinking of them, in the analysis, as PRFs or PRPs. Letting $F\colon \{0,1\}^k \times \{0,1\}^l \to \{0,1\}^l$ be a block cipher, here are some candidate constructions.

Below, we identify any string with an integer in the natural way. Thus, we will sometimes perform arithmetic operations on strings, such as writing $s+1$ where s is a string; it is understood that you look at s as a number, add 1 modulo $2^{|s|}$, and re-convert to a string. Similarly we may provide an integer i as an argument in place of a string, and it naturally means that the integer is first converted to a string. The length of a string corresponding to a given integer is context dependent, for example sometimes k and sometimes l, and we won't bother to distinguish these situations notationally. We let $n < 2^l$ be some parameter which influences the number of output bits of the generators; specifically, $m = nl$ are output.

Construction 1: G_1 takes a seed s of length l and returns

$$G_1(s) \;=\; F_1(s).F_2(s).\cdots.F_n(s)\;.$$

That is, the block cipher is invoked on known, constant key values, with the seed as input. This block cipher has key length l and output length $m = nl$.

Construction 2: G_2 takes a seed s of length k and returns

$$G_2(s) \;=\; F_s(1).F_s(2).\cdots.F_s(n)\;.$$

That is, the block cipher is invoked always on the same key, this key being the seed of the generator. The input points are known, fixed sequence of increasing values. This block cipher has key length k and output length $m = nl$.

Construction 3: G_3 takes a seed s of length k and returns

$$G_3(s) \;=\; F_s(s).F_s(s+1).\cdots.F_s(s+n-1)\;.$$

That is, the block cipher is invoked always on the same key, this key being the seed of the generator. The input points are however also functions of the key s, being a sequence determined by s. This block cipher has key length k and output length $m = nl$.

Construction 4: G_4 takes a seed s of length k and returns

$$G_4(s) \;=\; F_s(1).F_{s+1}(2).\cdots.F_{s+n-1}(n)\;.$$

That is, the block cipher is invoked on different keys related to the seed s, and on known inputs. This block cipher has key length k and output length $m = nl$.

What can we say about the security of these constructions, with regard to their meeting the kind of notion of security we defined above? That is, how close to random does $G(s)$ look when s is a random, hidden seed?

Construction 1 might have good statistical properties if F is a block cipher like DES. But it is insecure because it is predictable. Given the first block y_1 of $G(s) = y_1 \ldots y_n$, an adversary can compute y_2. That's because the keys under which the block cipher is being evaluated are known, and the block cipher is invertible. The adversary can compute $F_1^{-1}(y_1)$ and get back s. Then it can compute $y_2 = F_2(s)$.

Let us look at this attack more formally in the setting of our definition. To prove the claim that G_1 is insecure, we define the following attacker A_1, that gets as input an $m = nl$ bit string y. It wants to know whether y is random or $y = G(s)$ for a random s. How is the above weakness useful? There is no prediction to do here, since all of y is given to A_1. The idea is that A_1 predicts what y_2 should be according to the above, and bets that y is pseudorandom if the prediction is correct. Here is the full description of A_1.

Attacker $A_1(y)$
  Parse y as $y = y_1 \ldots y_n$ where $|y_i| = l$ for all $i = 1, \ldots, n$
  Let $s' = F_1^{-1}(y_1)$ and let $y_2' = F_2(s')$
  If $y_2' = y_2$ then return 1, else return 0

"Parse" here simply means that A_1 breaks y up into blocks of size l bits each, and names the i-th block y_i. To see how well A_1 does, we compute the quantities from Definition 3.5.1. Namely we claim that:

$$\begin{aligned} \mathbf{Succ}^{\mathrm{prg}}_{G_1}(A_1) &= 1 \\ \mathbf{Succ}^{\mathrm{prg}}_{\mathrm{Rand}}(A_1) &\leq 2^{-l}\;. \end{aligned}$$

If this is true, then clearly

$$\mathbf{Adv}_G^{\mathrm{prg}}(A_1) \;=\; \mathbf{Succ}_G^{\mathrm{prg}}(A_1) - \mathbf{Succ}_{\mathrm{Rand}}^{\mathrm{prg}}(A_1) \;\geq\; 1 - 2^{-l} \; .$$

Namely, the advantage of adversary A_1 is almost one. It does alarmingly well at distinguishing the output of G_1 from random strings. So G_1 is not a secure pseudorandom bit generator.

Now, how did we get the two success probabilities above? Look at the definition of the first quantity as given in Definition 3.5.1. It says that $y = G(s)$ where s is the seed in the experiment $s \stackrel{R}{\leftarrow} \{0,1\}^k \;;\; y \leftarrow G(s)$. Then the invertibility of F means that s', in the code of A_1, must equal s. So y_2' must equal y_2. So A_1 always outputs one.

In the second case, y is random. Adversary A_1 recovers some value s'; we don't really know much about what it might be. However, no matter what it is, the probability (over the choice of y) that $y_2' = y_2$ is 2^{-l}, since y_2 is random.

There is an important aspect of this attack that should be emphasized. Namely, the attack works *even if the block cipher is very secure.* The attack on the generator property was possible because the generator was poorly designed, not because the tool in use, namely the block cipher, was weak. Even for a strong block cipher F, construction G_1 is a very poor pseudorandom bit generator. These are the kinds of attacks we worry about.

Now, what about the other constructions? It seems hard to find an attack on G_2, and similarly for G_3 and G_4. Does that mean they are secure? Maybe not; maybe we are just not seeing the attack.

This is where the provable security approach takes a different turn from the classical approach. Whereas the latter would view a construction as good if some effort failed to break it, we seek instead a proof of security.

What does that mean? Not an absolute proof, since we don't in fact know whether or not the block cipher is secure. But recall what we said above about the fact that the attack on G_1 was due to poor design of G_1, not due to any weakness in the block cipher. What we want to prove is that such attacks don't exist. The form of the proof is a reduction; we want to show that the security of the block cipher (as a PRF) implies the security of the generator construction. So if F is secure, so is the generator based on it. That is a very useful thing to know, since it means we can stop looking for the kind of attacks we found on G_1.

In the next section, we will provide such a proof of security for the second construction above, namely G_2. But not for G_3 or G_4. That leads us to the question of what is their status: we know no attacks, yet we have no proofs of security.

The suggestion of the provable security approach is that when no proof is known, assume the worst. Treat the constructions as insecure. This is certainly a conservative approach, but as long as constructions that are provably secure and as efficient as doubtful ones are known, we may as well stick to them.

This may seem odd if we look more closely at G_3 and G_4. Shouldn't these be "better" than G_2? After all, G_2 evaluates the block cipher at fixed and known points, whereas for G_3, an adversary does not even know the point on which the block cipher is evaluated, let alone the key under which it is done. Similarly, in G_4, the key varies; shouldn't that help?

This kind of reasoning is spurious. Don't think that complicating a design necessarily helps. It could hurt. In general, evaluating a cipher at a point related to its key, as done by G_3, is not advocated for uses of ciphers in private key cryptography.

Although we don't know of attacks on G_3, G_4 when F is some specific block cipher like DES, it is possible to prove that they are weak constructions in a different sense. We can show that the assumption that F is a PRF is not enough to guarantee the security of these constructions. Namely, we can show that there are secure PRFs under which G_3, G_4 are insecure generators. (A later version of these notes might prove these results.) This means they are pretty dubious constructions, because we don't really understand what properties of F are necessary to make them secure.

3.5.3 Proof of security of G_2

Let us now see why the security of G_2 is guaranteed under the assumption that F is secure. The theorem below considers the construction in a slightly more general form than above.

Theorem 3.5.2 *Let $F\colon \{0,1\}^k \times \{0,1\}^l \to \{0,1\}^L$ be a PRF, and $n \leq 2^l$ an integer. Let $G\colon \{0,1\}^k \to \{0,1\}^{nL}$ be defined by*

$$G(s) \;=\; F_s(1).F_s(2).\cdots.F_s(n)\,.$$

Then

$$\mathbf{InSec}_G^{\mathrm{prg}}(t) \;\leq\; \mathbf{InSec}_F^{\mathrm{prf}}(t',n)$$

where $t' = t + O(n(l+L))$. ▌

The theorem upper bounds the insecurity of G in terms of that of F. The way to read the theorem is to think that by assumption $\mathbf{InSec}_F^{\mathrm{prf}}(t',n)$ is small. Meaning, our assumption is that F is secure. The theorem then tells us that $\mathbf{InSec}_G^{\mathrm{prg}}(t)$ is also small, meaning that G is also secure.

Before going on, it may be worth explaining a small point: why the restriction $n \leq 2^l$ in the above? If we set $n = 2^l + 1$, the last block of output of G is $F_s(2^l+1)$, which makes no sense, since the domain of $\{0,1\}^l$ of F has only 2^l different points. We might decide to do the addition modulo 2^l, but then the construction is insecure, because the first block and the (2^l+1)-th output blocks are the same. So the upper bound on n is necessary.

The main step in proving the theorem is the following lemma which provides a different kind of insight into what is going on.

Lemma 3.5.3 *Let F, n and G be as in the theorem above. Given any adversary A attacking G, we can construct a distinguisher D_A for F such that*

$$\mathbf{Adv}_G^{\mathrm{prg}}(A) \;\leq\; \mathbf{Adv}_F^{\mathrm{prf}}(D_A)\,,$$

and, furthermore, the running time of D_A is that of A plus $O(n(l+L))$, and D_A makes n oracle queries. ▌

Let us first see why the lemma implies the theorem. We will then discuss the meaning and proof of the lemma. It is simply a question of taking the maximum on both sides, which preserves the inequality, but here it is in more detail.

Proof of Theorem 3.5.2: We have

$$\mathbf{InSec}_G^{\mathrm{prg}}(t) \;=\; \max_A \{\, \mathbf{Adv}_G^{\mathrm{prg}}(A) \,\} \;=\; \mathbf{Adv}_G^{\mathrm{prg}}(A_*)$$

where A_* is a "best" adversary, meaning one achieving the maximum possible advantage within the class of all adversaries running for time at most t. Now Lemma 3.5.3 says that there is a distinguisher D_* such that $\mathbf{Adv}_G^{\mathrm{prg}}(A_*) \leq \mathbf{Adv}_F^{\mathrm{prf}}(D_*)$, and moreover D_* has running time that of A_* plus $O(n(l+L))$. Since A_* has running time at most t, distinguisher D_* has running time at most $t + O(n(l+L))$. Then certainly

$$\mathbf{Adv}_F^{\mathrm{prf}}(D_*) \;\leq\; \max_D \{\, \mathbf{Adv}_F^{\mathrm{prf}}(D) \,\} \;=\; \mathbf{InSec}_F^{\mathrm{prf}}(t',n)\,.$$

Here the max is over all distinguishers D that run in time at most t' and make at most n oracle queries. ▌

Before proving Lemma 3.5.3 let us try to interpret what it says. The best way to view it is as a challenge-response game. Suppose some hacker claims to have a very good algorithm A for breaking G. It runs in time t, where t is quite small, and has high advantage. According to the lemma, I can use A to construct a distinguisher D that breaks F with probability at least as high as the probability that A broke G. Furthermore, D is not much less efficient than A. But this is hard to believe, because F was supposed to be secure.

The best cryptanalysts in the scientific community do not know how to break F, yet we just found a way to do it, using A. Either the world has gone topsy-turvy, or A cannot exist. We prefer to believe the latter.

This is a reductionist argument, modeled after those in complexity theory. Analogously, if I prove that a certain problem is NP-complete, I am saying that a solution to this problem would provide a solution to the satisfiability problem. Since the best minds have not been able to solve the satisfiability problem, a solution to the new problem then becomes unexpected.

Proof of Lemma 3.5.3: Given A attacking G, our job is to define D_A attacking F, and achieving an advantage at least as great as that of G. Remember that distinguisher D gets an oracle for a function $g\colon \{0,1\}^l \rightarrow \{0,1\}^L$, and must output a bit. It will use A as a subroutine to help it output this bit. Remember that A takes input an nL bit string y, and returns 0 or 1. We will first define D_A and then explain why it works.

Distinguisher D_A^g
  For $i = 1, \ldots, n$ do $y_i \leftarrow g(i)$
  Let $y = y_1 \ldots y_n$
  Let $b = A(y)$
  Return b

Distinguisher D_A queries its oracle at the n points $1, \ldots, n$, and forms an nL-bit string y out of the responses. It feeds this to A, who returns some bit b as answer. D_A then returns b as its own answer.

We now claim that

$$\begin{aligned} \mathbf{Succ}_F^{\mathrm{prf}}(D_A) &= \mathbf{Succ}_G^{\mathrm{prg}}(A) \\ \mathbf{Succ}_{R^{l,L}}^{\mathrm{prf}}(D_A) &= \mathbf{Succ}_{\mathrm{Rand}}^{\mathrm{prg}}(A) . \end{aligned}$$

We will justify this a little later. For now, let us use it to conclude the proof of the lemma. That is easily done. Subtracting, we get

$$\mathbf{Succ}_F^{\mathrm{prf}}(D_A) - \mathbf{Succ}_{R^{l,L}}^{\mathrm{prf}}(D_A) = \mathbf{Succ}_G^{\mathrm{prg}}(A) - \mathbf{Succ}_{\mathrm{Rand}}^{\mathrm{prg}}(A) ,$$

and then we just note that the quantity on the left is $\mathbf{Adv}_F^{\mathrm{prf}}(D_A)$ while that on the right is $\mathbf{Adv}_G^{\mathrm{prg}}(A)$. Finally, look at the code of D_A and note that it makes n oracle queries. Its running time? It calls A, so that's t steps. The other operations are $O(n(l+L))$. This proves the lemma. It remains to justify the crucial pair of equations above.

Take first the claim that $\mathbf{Succ}_F^{\mathrm{prf}}(D_A) = \mathbf{Succ}_G^{\mathrm{prg}}(A)$. To justify this, we must look at how the these quantities are defined. In the first case, we are trying to estimate the probability that D_A^g returns 1 when its oracle g is chosen via $s \leftarrow \{0,1\}^k$; $g \leftarrow F_s$. The crucial observation is that if g is chosen like this, then the sequence y that D_A constructs is exactly $G(s)$. Now look at the definition of $\mathbf{Succ}_G^{\mathrm{prg}}(A)$. This is the probability that $A(G(s)) = 1$ when s is chosen at random. But since D is returning $A(y)$ with $y = G(s)$, its probability of returning 1 is exactly that of A in the same setting.

Next consider the claim that $\mathbf{Succ}_{R^{l,L}}^{\mathrm{prf}}(D_A) = \mathbf{Succ}_{\mathrm{Rand}}^{\mathrm{prg}}(A)$. To justify it we must again first return to the definitions. Look back at the definition of $\mathbf{Succ}_{R^{l,L}}^{\mathrm{prf}}(D_A)$. It tells us that we must evaluate the probability that D_A returns 1 when g is a random function. But if g is random, the sequence $g(1) \ldots g(n)$ is just a random string of nL bits. So D_A ends up feeding A a random string y. But that's exactly what happens in the experiment underlying $\mathbf{Succ}_{\mathrm{Rand}}^{\mathrm{prg}}(A)$. Since D_A simply returns $A(y)$, we get the claimed equality. ∎

This proof is a model for many others that will follow, and it is worth spending the time to understand it. Understanding it means making sure you know exactly why each claim or step in it is true. Where and how is it using the definitions? Where does the construction enter, and how is the definition of D_A used? Make sure you know it all.

A test of understanding is to try to do the same proof with the third construction, $G_3(s) = F_s(s).\cdots.F_s(s+n-1)$. The proof does not go through. Can you see where it fails, and why?

CHAPTER 4

Block ciphers and modes of operation

Block ciphers are the central tool in the design of practical protocols for private key cryptography. They are the main available "technology" we have at our disposal. This chapter will take a brief look at these ciphers and the state of their art.

It is important to stress that block ciphers are tools, not end-protocols. As with any tools, one has to learn to use them well. Even a good block cipher will not give you security if you use it poorly; but well-used, they are powerful tools indeed. Our emphasis in this course is on how to use block ciphers well, not on how to design or analyze block ciphers themselves. (There are many other good sources for information about the latter issues.) Accordingly, the purpose of this chapter is to get the reader acquainted with what these objects are like, and what kinds of properties they have or appear to have, in terms of efficiency and security. Most importantly, we want to get a sense of how we can later *model* block ciphers in some formal way so that schemes using them can be meaningfully analyzed; this is the subject of Chapter 5 on pseudorandom functions.

Having looked at some block ciphers, we will look at the "modes of operation." These are block cipher based protocols for encryption. They will give us a sense of what kind of analysis tasks we face.

4.1 What is a block cipher?

A block cipher is a function $F\colon \{0,1\}^k \times \{0,1\}^l \to \{0,1\}^l$ that takes two inputs, a k-bit key K and an l-bit "plaintext" x, to return an l-bit "ciphertext" $y = F(K, x)$. The *key-length* k and the *block-length* l are parameters associated to the block-cipher and vary from cipher to cipher, as of course does the design of the algorithm itself. For each key $K \in \{0,1\}^k$ we let $F_K\colon \{0,1\}^l \to \{0,1\}^l$ be the function defined by $F_K(x) = F(K, x)$. For any block cipher, and any key K, the function F_K is a *permutation* on $\{0,1\}^l$. This means that it is a bijection, ie. a one-to-one and onto function of $\{0,1\}^l$ to $\{0,1\}^l$. Accordingly it has an inverse, and we can denote it F_K^{-1}. This function also maps $\{0,1\}^l$ to $\{0,1\}^l$, and of course we have $F_K^{-1}(F_K(x)) = x$ and $F_K(F_K^{-1}(y)) = y$ for all $x, y \in \{0,1\}^l$. We let $F^{-1}\colon \{0,1\}^k \times \{0,1\}^l \to \{0,1\}^l$ be defined by $F^{-1}(K, y) = F_K^{-1}(y)$; this is the inverse cipher to F.

The block cipher is a public and fully specified algorithm. Both the cipher F and its inverse F^{-1} are easily computable. Meaning given K, x we can compute $F(K, x)$, and given K, y we can compute $F^{-1}(K, y)$.

In typical usage, a random key K is chosen and kept secret between a pair of users. The function F_K is then used by the two parties to process data in some way before they send it to each other. The adversary may see input-output examples of F_K, meaning pairs of the form (x, y) where $y = F_K(x)$, but is not directly shown the key K. Security relies on the secrecy of the key. (This is necessary, but not always sufficient, for security!) So at a first cut at least, you might view the adversary's goal as recovering the key K given some input-output examples of F_K. The block cipher should be designed to make this task computationally difficult. Later we will refine this view, but it is the classical one, so let's go with it for now.

How do block ciphers work? Lets take a look at some of them to get a sense of this.

4.2 Two block ciphers

From the host of block-ciphers around we pick two. The first and oldest, namely DES, and, at the other end, one of the most promising new-comers, RC6.

4.2.1 Data Encryption Standard

The Data Encryption Standard (DES) is the quintessential block cipher, and even if it is now, from some points of view, feeling its age and in need of replacement as a standard, no discussion of block ciphers should omit it. It is a remarkably well-engineered algorithm which has had a powerful influence on cryptography and is still in widespread use.

A brief history

In 1972 the NBS (National Bureau of Standards, now NIST, the National Institute of Science and Technology) initiated a program for data protection and wanted as part of it an encryption algorithm that could be standardized. They put out a request for such an algorithm. In 1974, IBM responded with a design based on their "Lucifer" algorithm. This design would eventually evolve into the DES. The cipher had a key-length of 56, and a block-length of 64.

Many bodies adopted DES as a standard: ANSI (American National Standards Institute), American Bankers Association, ANSI's Financial Institution Wholesale Security Group, etc, etc.

The standard was supposed to be reviewed every five years to see whether it should be re-adopted. Although there was some expectation that it would quickly need revision, it was in fact re-adopted again and again. Only very recently is the replacement of DES being seriously considered, in the AES (Advanced Encryption Standard) effort.

Indeed, DES proved remarkably secure. There was always, of course, the possibility of an exhaustive key-search attack, but for a fair length of time, the key size of 56 was deemed prohibitive enough. Attacks significantly different from key search emerged only in the nineties, and even then don't break DES in a sense significant in practice. But with today's technology, 56 bits is too small a key size for many security applications.

Construction

The construction is described in FIPS 46 [140], the original standards document. We refer the reader there. The following discussion is a quick guide that you can follow if you have the FIPS document at your side. (A full version of these notes ought of course to have its own pictures and descriptions so that it is self-contained ... maybe some day ...).

Begin at page 87 where you see a big picture. The input is 64 bits and in addition there is a 56 bit key K. (They say 64, but actually every eighth bit is ignored. It can be set to the parity of the previous seven.) Notice the algorithm is public. You operate with a hidden key, but nothing about the algorithm is hidden.

The first thing the input is hit with is something called the initial permutation, or IP. This just shuffles bit positions. That is, each bit is moved to some other position. How? In a fixed and specified way: see page 88. Similarly, right at the end, notice they apply the inverse of the same permutation. From now on, ignore these. They do not affect security. (As far as anyone can tell, they are there to make loading the chips easier.)

The essence of DES is in the round structure. There are 16 rounds. Each round i has an associated subkey K_i which is 48 bits long. The subkeys $K_1, \ldots, K_{16}$ are derived from the main key K, in a manner explained on page 95 of the FIPS document.

In each round, the input is viewed as a pair (L_i, R_i) of 32 bit blocks, and these are transformed into the new pair (L_{i+1}, R_{i+1}), via a certain function f that depends on a subkey K_i pertaining to round i. The structure of this transformation is important: it is called the Feistel transformation.

The Feistel transformation, in general, is like this. For some function g known to the party computing the transformation, it takes input (L, R) and returns (L', R') where $L' = R$ and $R' = g(R) \oplus L$. A central property of this transformation is that it is a permutation, and moreover if you can compute g then you can also invert the transform. Indeed, given (L', R') we can recover (L, R) via $R = L'$ and $L = g(R) \oplus R'$. For DES, the role of g in round i is played by $f(K_i, \cdot)$, the round function specified by the subkey K_i. Since $\mathrm{DES}_K(\cdot)$ is a sequence of Feistel transforms, each of which is a permutation, the whole algorithm is a permutation, and knowledge of the key K permits computation of $\mathrm{DES}_K^{-1}(\cdot)$.

Up to now the structure has been quite generic, and indeed many block-ciphers use this high level design: a sequence of Feistel rounds. For a closer look we need to see how the function $f(K_i, \cdot)$ works. See the picture on page 90 of the FIPS document. Here K_i is a 48 bit subkey, derived from the 56 bit key in a way depending on the round number. The 32 bit R_i is first expanded into 48 bits. How? In a precise, fixed way, indicated by the table on the same page, saying E-bit selection table. It has 48 entries. Read it as which inputs bits are output. Namely, output bits 32, 1, 2, 3, 4, 5, then 4, 5 again, and so on. It is NOT random looking! In fact barring that 1 and 32 have been swapped (see top left and bottom right) it looks almost sequential. Why did they do this? Who knows. That's the answer to most things about DES.

Now K_i is XORed with the output of the E-box and this 48 bit input enters the famous S-boxes. There are eight S-boxes. Each takes 8 bits to 6 bits. Thus we get out 32 bits. Finally, there is a P-box, a permutation applied to these 32 bits to get another 32 bits. You can see it on page 91.

What are the S-boxes? Each is a fixed, tabulated function, which the algorithm stores as tables in the code or hardware. You can see them on page 93. How to read them? Take the 6 bit input $b_1, b_2, b_3, b_4, b_5, b_6$. Interpret the first two bits as a row number, between 1 and 4. Interpret the rest as a column number, 1 through 16. Now index into the table.

Well, without going into details, the main objective of the above was to give you some idea of the kind of structure DES has. Of course, the main questions about the design are: why, why and why? What motivated these design choices? We don't know too much about this, although we can guess a little.

Speed

How fast can you compute DES? Let's begin by looking at hardware implementations since DES was in fact designed to be fast in hardware. We will be pretty rough; you can find precise estimates, for different architectures, in various places.

You can get over a Gbit/sec throughput using VLSI. Specifically at least 1.6 Gbits/sec, maybe more. That's pretty fast.

Some software figures I found quoted are: 12 Mbits/sec on a HP 9000/887; 9.8 Mbits/sec on a DEC Alpha 4000/610. The performance of DES in software is not as good as one would like, and is one reason people seek alternatives.

Security

Practical cryptanalysis of DES focuses on key-recovery. They formulate the problem facing the cryptanalyst like this. A 56-bit key K is chosen at random. Let $F_K = \mathrm{DES}_K$. Let q be some parameter.

Given: The adversary has a sequence of q input-output examples of F_K, say $(x_1, y_1), \ldots, (x_q, y_q)$, where $y_i = F_K(x_i)$ for $i = 1, \ldots, q$ and $x_1, \ldots, x_q$ are all distinct.

Find: The adversary must find the key K.

Two kinds of "attack" models are considered within this framework:

Known-message attack: $x_1, \ldots, x_q$ are any distinct points; the adversary has no control over them, and must work with whatever it gets.

Chosen-message attack: $x_1, \ldots, x_q$ are chosen by the adversary, perhaps even adaptively. That is, imagine it has access to an "oracle" for the function F_K. It can feed the oracle x_1 and get back $y_1 = F_K(x_1)$. It can then decide on a value x_2, feed the oracle this, and get back y_2, and so on.

Clearly a chosen-message attack gives the adversary much more power, and is also less realistic in practice.

Now, what about the security of DES? The most obvious attack is exhaustive key search.

Exhaustive key search: Go through all possible keys $K' \in \{0,1\}^k$ until you find the right one, namely K. How do you know when you hit K? If $\text{DES}_{K'}(x_1) = y_1$, you bet that $K' = K$. (Remember the assumption is that the adversary has input-output examples $(x_1, y_1), \ldots, (x_q, y_q)$ of the real function DES_K). Of course, you could be wrong. But the "chance" of being wrong is small, and gets much smaller if you do more such tests. It turns out that two tests is quite enough. That is, this attack only needs $q = 2$, a very small number of input-output examples.

Let us now describe the attack in more detail. For $i = 1, \ldots, 2^{56}$ let K_i denote the i-th 56-bit string (in lexicographic order). The following algorithm implements the attack, returning in practice the key K with "high probability".

For $i = 1, \ldots, 2^{56}$ do
  If $\text{DES}_{K_i}(x_1) = y_1$
    then if $\text{DES}_{K_i}(x_2) = y_2$ then return K_i

How long does this take? In the worst case, 2^{56} DES computations. (It turns out you can use a property of DES to reduce this to 2^{55}, but we won't bother with these details.) Even if you use the above mentioned 1.6 Gbits/sec chip to do these computation, the search takes about 6,000 years. So key search appears to be infeasible.

Yet, this conclusion is actually too hasty. We will return to key search and see why later.

Differential and linear cryptanalysis: The discovery of a less trivial attack waited until 1990. Differential cryptanalysis is capable of finding the key using 2^{47} input-output examples (that is, it requires $q = 2^{47}$). However, differential cryptanalysis required a chosen-message attack.

Linear cryptanalysis improved differential in two ways. The number of input-output examples required is reduced to 2^{43}, but also only a known-message attack is required.

These were major breakthroughs in cryptanalysis. Yet, their practical impact is small. Why? It difficult to obtain 2^{43} input-output examples. Furthermore, simply storing all these examples requires about 140 terabytes of data.

So what's the best possible attack? The answer is exhaustive key search. What we ignored above is *parallelism.*

Key search machines: A few years back it was argued that one can design a $1 million machine that does the exhaustive key search in about 3.5 hours. More recently, a DES key search machine was actually built, at a cost of $250,000. It finds the key in 56 hours, or about 2.5 days. The builders say it will be cheaper to build more machines now that this one is built.

Thus DES is feeling its age. Yet, it would be a mistake to take away from this discussion the impression that DES is weak. Rather, what the above says is that it is an impressively strong algorithm. After all these years, the best practical attack known is still exhaustive key search. That says a lot for its design and its designers.

Later we will see that that we would like security properties from a block cipher that go beyond resistance to key-recovery attacks. It turns out that from that point of view, a limitation of DES is its block size. Birthday attacks "break" DES with about $q = 2^{32}$ input output examples. (The meaning of "break" here is very different from above.) Here 2^{32} is the square root of 2^{64}, meaning to resist these attacks we must have bigger block size. The next generation of ciphers has taken this into account.

4.2.2 RC6

There is an on-going effort to standardize a new algorithm to replace DES. It will be called the AES (Advanced Encryption Standard). The requirements are that it have a block size of 128, and a key size of at least 80. In response to the call, many proposals have been made. RC6 is one of the front-runners. Refer to [163] for its description and properties.

RC6 illustrates how cipher design has evolved: towards simplicity. It is a succinct and elegant algorithm which can be completely described in very little space. There are no large tables.

It has variable key and block size. The "standard" version sets both the key and block size to 128 bits.

The algorithm is relatively new, and not much cryptanalysis has yet been performed. It is fast in software.

Pretty much everything you would want to know is in [163] so we won't say more here.

4.3 Some Modes of operation

Let a block cipher F be fixed, and assume two parties share a key K for this block cipher. This gives them the ability to compute the functions $f(\cdot) = F_K(\cdot)$ and $f^{-1}(\cdot) = F_K^{-1}(\cdot)$. This function can be applied to an input of l-bits. Typically the block size l is 64 or 128. Yet in practice we want to process much larger inputs, say text files to encrypt. To do this one uses a block cipher in some mode of operation. There are several of these. We will illustrate by describing three of them that exhibit different kinds of features.

We will divide the presentation of the modes of operation into two parts. We first introduce some basic operations, then instantiate them in various ways to get the actual modes.

4.3.1 Three basic operations

We look at three modes: ECB (Electronic Code-Book), CBC (Cipher Block Chaining) and CTR (Counter). In each case there is an encryption process which takes an nL-bit string x and returns a string y. An associated decryption process recovers x from y. The encryption operation is performed using an oracle for a function $f\colon \{0,1\}^l \to \{0,1\}^L$. In typical usage, f will be $f = F_K$, the block cipher induced permutation that the two parties sharing K can compute, in which case $l = L$. But it is conceptually useful to separate the usage of the key from an application of the function $f = F_K$ that the key enables one to compute, because later we will want to plug in other functions in the role of f. So for now just imagine that you have available a subroutine (that is, an oracle) for computing a function $f\colon \{0,1\}^l \to \{0,1\}^L$. For ECB and CBC, the decryption requires the ability to compute f^{-1}, and thus in particular requires that f be a permutation, which is true when $f = F_K$ for a block cipher F. On the other hand CTR mode only needs access to f even for the decryption, and thus does not even require f to be a permutation.

Algorithm $\mathsf{E\text{-}ECB}^f(x_1 \ldots x_n)$	Algorithm $\mathsf{D\text{-}ECB}^{f^{-1}}(y_1 \ldots y_n)$
For $i = 1, \ldots, n$ do $y_i \leftarrow f(x_i)$	For $i = 1, \ldots, n$ do $x_i \leftarrow f^{-1}(y_i)$
Return $y_1 \ldots y_n$	Return $x_1 \ldots x_n$

CBC mode processes the data based on some *initial vector* IV which is an l-bit string.

Algorithm $\mathsf{E\text{-}CBC}^f_{\mathrm{IV}}(x_1 \ldots x_n)$
  $y_0 \leftarrow \mathrm{IV}$
  For $i = 1, \ldots, n$ do $y_i \leftarrow f(y_{i-1} \oplus x_i)$
  Return $y_0 y_1 \ldots y_n$

Algorithm $\mathsf{D\text{-}CBC}^{f^{-1}}(y_0 y_1 \ldots y_n)$
  For $i = 1, \ldots, n$ do $x_i \leftarrow f^{-1}(y_i) \oplus y_{i-1}$
  Return $x_1 \ldots x_n$

Unlike $\mathsf{E\text{-}ECB}^f(x)$ this operation is not length preserving: the output is l-bits longer than the input.

CTR mode also uses an auxiliary value, an "initial value" IV which is an integer in the range $0, 1, \ldots, 2^l - 1$.

Algorithm $\mathsf{E\text{-}CTR}^f_{\mathrm{IV}}(x_1 \ldots x_n)$
  For $i = 1, \ldots, n$ do $y_i \leftarrow f(\langle \mathrm{IV} + i \rangle) \oplus x_i$
  Return $\langle \mathrm{IV} \rangle y_1 \ldots y_n$

Algorithm $\mathsf{D\text{-}CTR}^f(\langle \mathrm{IV} \rangle y_1 \ldots y_n)$
  For $i = 1, \ldots, n$ do $x_i \leftarrow f(\langle \mathrm{IV} + i \rangle) \oplus y_i$
  Return $x_1 \ldots x_n$

Here addition is done modulo 2^l, and $\langle j \rangle$ denotes the binary representation of integer j as an l-bit string. Notice that in this case, recovery did not require access to f^{-1}, and in fact did not even require that f be a permutation. Also notice the efficiency advantage over CBC: the encryption is parallelizable.

4.3.2 Some modes of operation

The setting for using the modes of operation is that a block cipher F is agreed upon, and two communicating parties share a key K for this cipher that was chosen at random from $\{0,1\}^k$ and is initially unknown to the adversary. They then set $f = F_K$ in the above, and also suitably choose any other parameters. Here in more detail are several possible ways to compute a ciphertext y corresponding to the plaintext $x = x_1 \ldots x_n$. All of these are symmetric encryption schemes, and their security in this light will be discussed in the chapter on symmetric encryption.

Electronic codebook mode. The ciphertext is computed as $y = \mathsf{E\text{-}ECB}^{F_K}(x_1 \ldots x_n)$. If F is a block cipher then knowledge of K enables computation of F_K^{-1}, and thus the ciphertext y can be decrypted to recover x, by a party knowing K, by following the reversal process $\mathsf{D\text{-}ECB}^{F_K^{-1}}$ outlined above.

Cipher-block chaining mode. To encipher x, pick an l-bit IV at random and set $y = \mathsf{E\text{-}CBC}^{F_K}_{\mathrm{IV}}(x_1 \ldots x_n)$. If F is a block cipher then knowledge of K enables computation of F_K^{-1}, and thus the ciphertext y can be decrypted to recover x, by a party knowing K, by following the reversal process $\mathsf{D\text{-}CBC}^{F_K^{-1}}$ outlined above.

Note that this encryption procedure is probabilistic: each time a plaintext x is to be encrypted, a new random IV is chosen by the encryptor. The choices of IV are made independently each time.

Counter based cipher-block chaining mode. In this mode of operation, the encryptor maintains an l-bit counter c, which is an integer in the range $0, 1, \ldots, 2^l - 1$, initially 0. To encrypt plaintext x the encryptor sets $y = \mathsf{E\text{-}CBC}^{F_K}_{\langle c \rangle}(x_1 \ldots x_n)$. The encryptor then increments c via $c \leftarrow c + n$. Note that this encryption procedure is stateful. In Homework Problem 2.1 you are asked to show that this version of CBC encryption is insecure.

Counter based CTR mode. In this mode of operation, the encryptor maintains an l-bit counter c, which is an integer in the range $0, 1, \ldots, 2^l - 1$, initially 0. To encrypt plaintext x the encryptor sets $y = \mathsf{E\text{-}CTR}^{F_K}_c(x_1 \ldots x_n)$. The encryptor then increments c via $c \leftarrow c + n$. A party knowing K can recover x from y via the reversal process $\mathsf{D\text{-}CTR}^{F_K}$ outlined above.

Note that this encryption procedure is stateful: the sender must maintain a counter and be sure to increment it appropriately after each encryption. We stress that only the sender maintains a counter. The receiver does not, and thus there is no synchronization problem.

Randomized CTR mode. To encrypt x, pick an l-bit integer IV at random in the range $0, 1, \ldots, 2^l - 1$, and set $y = \mathsf{E\text{-}CTR}^{F_K}_{\mathrm{IV}}(x_1 \ldots x_n)$. A party knowing K can recover x from y via the reversal process $\mathsf{D\text{-}CTR}^{F_K}$ outlined above. Note that this encryption procedure is randomized.

4.4 Exercises

Exercise 1: Show how to use the complementation property of DES to speed up exhaustive key search by a factor of two.

CHAPTER 5

Pseudo-random functions

Pseudorandom function families are one of the fundamental primitives of cryptographic protocol design. Most importantly, they provide a basis for private key cryptography.

As compared to one-way functions, pseudorandom function families are a more "high level" primitive. They have very strong properties, and it is this that makes them useful.

Below we will discuss both what are pseudorandom function families and various candidate examples for them.

5.1 Function families

A *function family* is a map $F\colon \mathit{Keys}(F) \times \mathit{Dom}(F) \to \mathit{Range}(F)$. Here $\mathit{Keys}(F)$ is the set of keys of F; $\mathit{Dom}(F)$ is the domain of F; and $\mathit{Range}(F)$ is the range of F. The two-input function F takes a key K and input x to return a point y we denote by $F(K,x)$.

For any key $K \in \mathit{Keys}(F)$ we define the map $F_K\colon \mathit{Dom}(F) \to \mathit{Range}(F)$ by $F_K(x) = F(K,x)$. Thus, F specifies a collection of maps, one for each key. That's why we call F a family of functions.

Most often $\mathit{Keys}(F) = \{0,1\}^k$ and $\mathit{Dom}(F) = \{0,1\}^l$ and $\mathit{Range}(F) = \{0,1\}^L$ for some values of k, l, L. But there are some times where the domain or range could be sets of different kinds, containing strings of varying lengths.

There is some probability distribution on the set of keys $\mathit{Keys}(F)$. Most often, just the uniform distribution. So with $\mathit{Keys}(F) = \{0,1\}^k$, we are just drawing a random k-bit string as a key.

We denote by $K \stackrel{R}{\leftarrow} \{0,1\}^k$ the operation of selecting a random k-bit string from $\{0,1\}^k$ and naming it K. Then $f \stackrel{R}{\leftarrow} F$ denotes the operation: $K \stackrel{R}{\leftarrow} \{0,1\}^k$; $f \leftarrow F_K$. In other words, let f be the function F_K where K is a randomly chosen key. We are interested in the input-output behavior of this function.

A *permutation* is a map whose domain and range are the same set, and the map is a bijection, ie. one-to-one and onto. That is, $f\colon D \to R$ is a permutation if $D = R$ and $x \neq y$ implies $f(x) \neq f(y)$ for all $x, y \in D$.

We say that F is a family of permutations if $\mathit{Dom}(F) = \mathit{Range}(F)$ and each F_K is a permutation on this common set.

Example 5.1.1 A block cipher is a family of permutations. For example, DES is a family of permutations with $\mathit{Keys}(\mathrm{DES}) = \{0,1\}^{56}$ and $\mathit{Dom}(\mathrm{DES}) = \{0,1\}^{64}$ and $\mathit{Range}(\mathrm{DES}) = \{0,1\}^{64}$. Here $k = 56$ and $l = L = 64$. Similarly for RC6.

There are two function families that we fix. One is $R^{l,L}$, the family of all functions of $\{0,1\}^l$ to $\{0,1\}^L$, and the other is P^l, the family of all permutations on $\{0,1\}^l$. A random member of the first is simply a random function; a random member of the second is a random permutation.

What are these families, more precisely?

The key of a function in $R^{l,L}$ is simply the full description of the function. That is,

$$Keys(R^{l,L}) \quad = \quad \{ \, (y_1, \ldots, y_{2^l}) \; : \; y_1, \ldots, y_{2^l} \in \{0,1\}^L \, \}$$

is the set of all sequences of length 2^l in which each entry of a sequence is an L-bit string. For any $x \in \{0,1\}^l$ we interpret x as an integer in the range $\{1, \ldots, 2^l\}$ and set

$$R^{l,L}((y_1, \ldots, y_{2^l}), x) = y_x \; .$$

Notice that the key space is very large; it has size 2^{L2^l}. Naturally, since there is a key for every function of l-bits to L-bits, and this is the number of such functions. The key space is equipped with the uniform distribution, so that $f \stackrel{R}{\leftarrow} R^{l,L}$ is the operation of picking a random function of l-bits to L-bits.

On the other hand, for P^l, the key space is

$$Keys(P^l) \quad = \quad \{ \, (y_1, \ldots, y_{2^l}) \; : \; y_1, \ldots, y_{2^l} \in \{0,1\}^l \text{ and } y_1, \ldots, y_{2^l} \text{ are all distinct } \} \; .$$

For any $x \in \{0,1\}^l$ we interpret x as an integer in the range $\{1, \ldots, 2^l\}$ and set

$$P^l((y_1, \ldots, y_{2^l}), x) = y_x \; .$$

The key space is again equipped with the uniform distribution, so that $f \stackrel{R}{\leftarrow} P^l$ is the operation of picking a random permutation on $\{0,1\}^l$. In other words, all the possible permutations on $\{0,1\}^l$ are equally likely.

We will hardly ever actually think about these two families in terms of this formalism. Indeed, it is worth pausing here to see how to think about them more intuitively, because they are important objects.

Above, F is a finite object, consisting of functions of l to L. For example, $l = L = 64$. There is no function applying, say, to 128 bit inputs. Sometimes we want to consider families of finite families. An infinite family of functions is a collection $F^1, F^2, F^3, \ldots$, written $\{F^n\}_{n \geq 1}$. Each F^n is a (finite) family of functions. Here n is a security parameter. The numbers l, L, k now turn into functions of n, and for each n, the finite family F^n has input length $l(n)$ and output length $L(n)$ and key length $k(n)$.

We will sometimes discuss things in terms of finite families, sometimes infinite ones. The difference is largely a question of what we construct.

5.2 Random functions and permutations

A random function of l-bits to L-bits is a function chosen randomly from $R^{l,L}$. That is, all functions are equally likely to be chosen. The function is identified with its table, which gives the output value for any associated input value. This table is exactly what we called the key above.

We will consider settings in which you have black-box access to a function g. This means that there is a box to which you can give any value x of your choice (provided x is in the domain of g), and box gives you back $g(x)$. But you can't "look inside" the box; your only interface to it is the one we have specified.

Different things may be placed in the box, but one such thing is a random function. This means that someone picked g at random from $R^{l,L}$, and put g in the box. Now, you query the box, with various values of inputs, all from $\{0,1\}^l$. What do you see?

Each time you give the box an input, you get back a random L-bit string, with the sole constraint that if you twice give the box the same input x, it will be consistent, returning both times the same output $g(x)$. That is exactly what is a random function.

In other words, a random function of l-bits to L-bits can be thought of as a box which given any input $x \in \{0,1\}^l$ returns a random number, except that if you give it an input you already gave it before, it returns the same thing as last time. It is this "dynamic" view that we suggest the reader have in mind in thinking about random functions.

The dynamic view can be thought of as following program. The program maintains a table of entries of the form (x, y).

> **Input:** $x \in \{0,1\}^l$
> If an entry (x, y) exists in table return y and exit
> Else flip coins to determine a string $y \in \{0,1\}^L$ and return it
> Also put (x, y) into the table.

The answer on any point is random and independent of the answers on other points.

Another way to think about a random function is as a large, pre-determined random table. The entries are of the form (x, y). For each x someone has flipped coins to determine y and put it into the table.

We are more used to the idea of picking points at random. Here we are picking a *function* at random.

Let's do some simple probabilistic computations to understand random functions. Fix $x \in \{0,1\}^l$ and $y \in \{0,1\}^L$. Then:

$$\mathbf{P}\left[\, f(x) = y \;:\; f \stackrel{R}{\leftarrow} R^{l,L} \,\right] = 2^{-L} \,.$$

Notice it doesn't depend on l. Also it doesn't depend on the values of x, y.

Now fix $x_1, x_2 \in \{0,1\}^l$ and $y \in \{0,1\}^L$. Then:

$$\mathbf{P}\left[\, f(x_1) = f(x_2) = y \;:\; f \stackrel{R}{\leftarrow} R^{l,L} \,\right] = \begin{cases} 2^{-2L} & \text{if } x_1 \neq x_2 \\ 2^{-L} & \text{if } x_1 = x_2 \end{cases}$$

This illustrates *independence*. Finally fix $x_1, x_2 \in \{0,1\}^l$ and $y \in \{0,1\}^L$. Then:

$$\mathbf{P}\left[\, f(x_1) \oplus f(x_2) = y \;:\; f \stackrel{R}{\leftarrow} R^{l,L} \,\right] = \begin{cases} 2^{-L} & \text{if } x_1 \neq x_2 \\ 0 & \text{if } x_1 = x_2 \text{ and } y \neq 0^L \\ 1 & \text{if } x_1 = x_2 \text{ and } y = 0^L \end{cases}$$

Similar things hold for the sum of more than two things.

5.3 Motivation for PRFs

Before providing the definition of pseudorandom function families let us talk about the motivation and applications.

5.3.1 The shared random function model

In private key cryptography, the parties A and B share a key K which the adversary doesn't know. They want to use this key to achieve various things. In particular, to encrypt and authenticate the data they send to each other.

A key is (or ought to be) a short string. Suppose however that we allow the parties a very long shared string. The form it takes is a random function f of l bits to L bits, for some pre-specified l, L. This is called the shared random function model.

The shared random function model cannot really be realized because, as we saw, random functions are just too big to even store. It is a conceptual model.

To work in this model, we give the parties oracle access to f. They may write down $x \in \{0,1\}^l$ and in one step be returned $f(x)$.

Now it turns out that the shared random function model is a very convenient model in which to think about cryptography, formulate schemes, and analyze them. In particular, we will see many examples where we design schemes in the shared random function model and prove them secure. This is true for a variety of problems, but most importantly for encryption and message authentication. The proof of security here is absolute: we do not make any restrictions on the computational power of the adversary, but are able to simply provide an upper bound on the success probability of the adversary.

But now what? We have schemes which are secure but a priori can't be efficiently realized, since they rely on random functions. That's where pseudorandom function families come in. A PRF family is a family F of functions indexed by small keys (eg. 56 or 128 bits). However, it has the property that if K is shared between A and B, and we use F_K in place of a random function f in some scheme designed in the shared random function model, the resulting scheme is still secure as long as the adversary is computationally bounded.

In other words, what we want is that PRFs can be used in place of random functions in shared key schemes. Pseudorandom functions are designed to make this happen for as wide a range of applications as possible. A pseudorandom function is specified by a short key K, and the parties need only store this key. Then, they use this function in place of the random function in the scheme. And things should work out, in the sense that if the scheme was secure when a random function was used, it should still be secure.

This is a very rough idea. Technically, it is not always true: this is the intuition. Pseudorandom functions don't always work. That is, you can't substitute them for random functions in any usage of the latter and expect things to work out. But if used right, it works out in a large number of cases. How do we identify these cases? We have to provide a formal definition of a pseudorandom function family and see how to test whether, in a particular usage of random functions, it is OK to substitute pseudorandom ones. In this context we stress one important point straightaway. Pseudorandom functions rely on the key K being *secret*. The adversary is not given K and cannot directly compute the function. (Of course it might gain some information about values of F_K on various points via the usage of F_K by the legitimate parties, but that will be OK.) In other words, you can substitute *shared, secret* random functions by PRFs, but not public ones.

Pseudorandom functions are an intriguing notion and a powerful tool. Try to appreciate the strength and utility of the claim. Whether you want to design a scheme for encryption, authentication, or some other purpose, design it in the shared random function model. Then simply substitute the random function with a pseudorandom one, and things should still be secure!

5.3.2 Modeling block ciphers

One of the primary motivations for the notions of pseudorandom functions (PRFs) and pseudorandom permutations (PRPs) is to model block ciphers and thereby enable the security analysis of protocols that use block ciphers.

Classically the security of DES or other block ciphers has been looked at only with regard to key recovery. That is, analysis of a block cipher F has focused on the following question: Given some number of input-output examples $(x_1, F_K(x_1)), \ldots, (x_q, F_K(x_q))$ where K is a random, unknown key, how hard is it to find K?

Of course, security requires that it be hard to find K unless q is quite large. Yet, even a cursory glance at common block cipher usages shows that hardness of key recovery is not *sufficient* for security.

Take for example the CTR mode of operation discussed in Chapter 2. Suppose that the block cipher had the following weakness: Given $C, F_K(C+1), F_K(C+2)$, it is possible to compute $F_K(C+3)$. Then clearly the encryption scheme is not secure, because if an adversary happens to know the first two message blocks, it can figure out the third message block from the ciphertext. (It is perfectly reasonable to assume the adversary already knows the first two message blocks. These might, for example, be public header information, or the name of some known recipient.) This means that if CTR mode encryption is to be secure, the block cipher must have the property that given $C, F_K(C+1), F_K(C+2)$, it is computationally infeasible to compute $F_K(C+3)$. Let us call this property SP1, for "security property one".

Of course, anyone who knows the key K can easily compute $F_K(C+3)$ given $C, F_K(C+1), F_K(C+2)$. And it is hard to think how one can do it without knowing the key. But there is no guarantee that someone *cannot* do this without knowing the key. That is, confidence in the security of F against key recovery does *not* imply that SP1 is true.

This phenomenon continues. As we see more usages of ciphers, we build up a longer and longer list of security properties SP1, SP2, SP3, ... that are necessary for the security of some block cipher based application.

Furthermore, even if SP1 is true, CTR mode encryption may still be weak. SP1 is not *sufficient* to guarantee the security of CTR mode encryption. Similarly with other security properties that one might naively come up with.

This long list of necessary but not sufficient properties is no way to treat security. What we need is *one single* "MASTER" property of a block cipher which, if met, *guarantees* security of *lots of* natural usages of the cipher. This property is that the block cipher is a pseudorandom permutation (or function).

5.4 Pseudorandom functions and Permutations

As we discussed above, what we want is functions which can be used in place of random functions in cryptographic schemes like MACs. We will have a keyed family F of functions. The parties will share a key K specifying a function $f = F_K$, and then use this function in place of the random function in the scheme. What we want is that this "works out." Namely, the scheme should still be secure.

The notion of pseudorandom function families was designed to make this happen. It considers usages of a random function in which the description of the function is hidden and the function is only accessed in a black box way. Roughly, a pseudo-random function has the property that if you see input-output examples of the function, they look like input-output examples of a random function.

The central notion is that of a distinguisher.

5.4.1 Pseudorandom Functions

Let $F\colon \{0,1\}^k \times \{0,1\}^l \to \{0,1\}^L$ be a family of functions. Now, consider two worlds, in each of which you are given a box containing a function g, as follows:

World 0: You are given a box for a function g drawn at random from $R^{l,L}$, namely $g \stackrel{R}{\leftarrow} R^{l,L}$. (So g is just a random function of l bits to L bits.)

World 1: You are given a box for a function g drawn at random from F, namely $g \stackrel{R}{\leftarrow} F$. (If F is keyed via keys of length K, this means someone chose a key K at random, let $g = F_K$, and put g in the box. But they did NOT give you K!)

Now someone puts you in one of these worlds and asks you which one you are in. To find out, you are allowed to query the function g on points of your choice. Based on its input-output behavior, you must eventually make some decision as to which world you are in. Note that the only way you are allowed to access the box is by giving it an input x and obtaining output $g(x)$; you can't otherwise "look inside" the box.

The quality of pseudo-random family F can be thought of as measured by the difficulty of telling, in the above game, whether a function belongs in World 0 or World 1.

Intuitively, the game just models some way of "using" the function g in an application like an encryption scheme. If it is not possible to distinguish the input-output behavior of a random member of F from a truly random function, the application should behave in roughly the same way whether it uses a function from F or a random function. Later we will see exactly how this works out; for now let us continue to develop the notion. But right away one warning should be issued. Pseudorandom functions can't be substituted for random functions in *all* usages of random functions. To make sure it is OK in a particular application, you have to make sure that it falls within the realm of applications for which the formal definition below can be applied.

Example 5.4.1 Suppose F is a family of functions which happens to have the property that $F_K(0^l) = 0^L$ for all keys $K \in \{0,1\}^k$. Then F is *not* a pseudorandom function family. This is because we could easily tell whether we were in world 0 or world 1. Query the given function g at 0^l. If we get back 0^L we predict we are in world 1, else in world 0. The probability of being wrong is the probability that $g(0^l) = 0^L$ if g is random, and this is at most 2^{-L}.

The act of trying to tell which world you are in is formalized via the notion of a *distinguisher*. This is an algorithm which is provided oracle access to a function g and tries to decide if g is random or pseudorandom. (Ie. whether it is in world 0 or world 1.) A distinguisher can only interact with the function by giving it inputs and examining the outputs for those inputs; it cannot examine the function directly in any way. Intuitively, a family is pseudorandom if the probability that the distinguisher says 0 is roughly the same regardless of which world it is in. We capture this mathematically by via the two "success" probabilities defined below.

Definition 5.4.2 Let $F: \{0,1\}^k \times \{0,1\}^l \to \{0,1\}^L$ be a family of functions, and let D be an algorithm that takes an oracle for a function $g: \{0,1\}^l \to \{0,1\}^L$, and outputs a bit. Let

$$\begin{aligned} \mathbf{Succ}^{\mathrm{prf}}_F(D) &= \mathbf{P}\left[D^g = 1 \,:\, g \stackrel{R}{\leftarrow} F \right] \\ \mathbf{Succ}^{\mathrm{prf}}_{R^{l,L}}(D) &= \mathbf{P}\left[D^g = 1 \,:\, g \stackrel{R}{\leftarrow} R^{l,L} \right] . \end{aligned}$$

We call the difference between them the "advantage" of D:

$$\mathbf{Adv}^{\mathrm{prf}}_F(D) = \mathbf{Succ}^{\mathrm{prf}}_F(D) - \mathbf{Succ}^{\mathrm{prf}}_{R^{l,L}}(D) .$$

Finally for any t, q we let

$$\mathbf{InSec}^{\mathrm{prf}}_F(t,q) = \max_D \{ \mathbf{Adv}^{\mathrm{prf}}_F(D) \}$$

where the maximum is over all D running in time t and making at most q oracle queries. ∎

Here t is on some fixed RAM model and includes the size of the code.

If $\mathbf{Adv}^{\mathrm{prf}}_F(D)$ is small, it means that D is outputting 1 just about as often in world 0 as in world 1, meaning it is not doing a good job of telling which world it is in. If this quantity is large (meaning close to one) then the distinguisher is doing well, meaning our family F is not pseudorandom.

For the family to be pseudorandom, the advantage of a distinguisher must be small, no matter what strategy the distinguisher tries. However, we expect that the advantage grows as the distinguisher invests more effort (more time, more queries) in the process. We can't expect it to be impossible for *any* distinguisher to tell the two worlds apart, but then neither is this what we need. Conforming to a general theme in cryptographic definitions, we only care what happens when the adversary (here the distinguisher) is limited in the amount of resources it can bring to the problem. This is because in reality, adversaries are limited; for example, they can't do 2^{80} steps of computation. To capture this we define the insecurity function $\mathbf{InSec}^{\mathrm{prf}}_F(\cdot,\cdot)$ as above. This is a function associated to any function family F. Once we fix F, this function becomes fixed. We may not always know its value at all points, but the value exists.

The strength of this definition lies in the fact that it does not specify anything about the kinds of strategies that can be used by a distinguisher; it only limits its time and number of queries. A distinguisher can use whatever means desired to distinguish the function as long as it stays within the specified resource bounds. Nonetheless, it should fail, except possibly with probability $\mathbf{InSec}^{\mathrm{prf}}_F(t,q)$ where t is its running time and q the number of its queries.

There is one feature of the above parameterization about which everyone asks. Recall that k is the key-length for F. Obviously, the key length is a fundamental determinant of security: larger key length will typically mean more security. Yet, the key length k does not appear explicitly in the insecurity function $\mathbf{InSec}_F^{\mathrm{prf}}(t,q)$. Why is this? $\mathbf{InSec}_F^{\mathrm{prf}}(t,q)$ is in fact a function of k, but without knowing more about F it is difficult to know what kind of function. The truth is that the key length itself does not matter: what matters is just the advantage a distinguisher can obtain. In a well-designed block cipher, $\mathbf{InSec}_F^{\mathrm{prf}}(t,q)$ should be about $t/2^k$. But that is really an ideal; in practice we should not assume ciphers are this good.

Notice that the definition of a PRF F does not make any explicit requirement on how fast it is to compute $F_K(x)$ given K and x; it only talks about security. Of course, in practice it is important that the computation of the function is fast and that the key K is short.

5.4.2 Pseudorandom Permutations

Recall that a block cipher F is a family of *permutations*; each individual function F_K in the family is a permutation. So it is somewhat unfair to ask a block-cipher to behave like a family of random functions; it is fairer to ask that it behave like a family of random permutations. This leads to the notion of a pseudorandom permutation, in which we proceed exactly as above, but replace $R^{l,L}$ with P^l.

Let $F\colon \{0,1\}^k \times \{0,1\}^l \to \{0,1\}^l$ be a family of functions. (Not necessarily permutations!) Now, consider two worlds, in each of which you are given a box containing a function g, as follows:

World 0: You are given a box for a function g drawn at random from P^l, namely $g \stackrel{R}{\leftarrow} P^l$. (So g is just a random permutation of l bits to l bits.)

World 1: You are given a box for a function g drawn at random from F, namely $g \stackrel{R}{\leftarrow} F$.

Definition 5.4.3 Let $F\colon \{0,1\}^k \times \{0,1\}^l \to \{0,1\}^l$ be a family of functions, and let D be an algorithm that takes an oracle for a function $g\colon \{0,1\}^l \to \{0,1\}^l$, and outputs a bit. Let

$$\begin{aligned} \mathbf{Succ}_F^{\mathrm{prp}}(D) &= \mathbf{P}\left[\, D^g = 1 \;:\; g \stackrel{R}{\leftarrow} F \,\right] \\ \mathbf{Succ}_{P^l}^{\mathrm{prp}}(D) &= \mathbf{P}\left[\, D^g = 1 \;:\; g \stackrel{R}{\leftarrow} P^l \,\right] . \end{aligned}$$

We call the difference between them the "advantage" of D:

$$\mathbf{Adv}_F^{\mathrm{prp}}(D) = \mathbf{Succ}_F^{\mathrm{prp}}(D) - \mathbf{Succ}_{P^l}^{\mathrm{prp}}(D) .$$

Finally for any t,q we let

$$\mathbf{InSec}_F^{\mathrm{prp}}(t,q) = \max_D \{ \mathbf{Adv}_F^{\mathrm{prp}}(D) \}$$

where the maximum is over all D running in time t and making at most q oracle queries. ▮

This $\mathbf{InSec}_F^{\mathrm{prp}}(\cdot,\cdot)$ is an insecurity function associated to any function family F. Once we fix F, this function becomes fixed.

5.4.3 An example

Intuitively, a if F is a PRF then, when K is random but unknown, the input-output behavior of F_K must resemble that of a random function. It may be worth seeing this illustrated in a very simply instance. Take a very simple property of a random function, that we discussed above in Section 5.2. Namely fix $x \in \{0,1\}^l$ and $y \in \{0,1\}^L$. Then

$$\mathbf{P}\left[\, f(x) = y \;:\; f \stackrel{R}{\leftarrow} R^{l,L} \,\right] = \frac{1}{2^L} .$$

Now, consider the quantity

$$\mathbf{P}\left[\, f(x) = y \;:\; f \stackrel{R}{\leftarrow} F \,\right] .$$

If F is a good PRF, this value should be close to the value in the random case, namely 2^{-L}. How close? This is determined by the insecurity function of F, once we have decided how much time, and how many queries, this property corresponds to. The latter needs some explaining.

Let us first give an answer and then justify it. Let $\epsilon = \mathbf{InSec}_F^{\mathrm{prf}}(t, 1)$ where t is to be specified later. We claim that

$$\mathbf{P}\left[\, f(x) = y \;:\; f \stackrel{R}{\leftarrow} F \,\right] \;\leq\; \frac{1}{2^L} + \epsilon \,. \tag{5.1}$$

In fact it is also true that $\mathbf{P}\left[\, f(x) = y \;:\; f \stackrel{R}{\leftarrow} F \,\right] \geq 2^{-L} - \epsilon$, but since this example is for illustrative purposes only, we illustrate only the upper bound.

Let us now proceed to formally prove that Equation (5.1) is correct. How do we set up the proof? There are not too many choices; like all these proofs in cryptography it is a reductionist argument.

Imagine the claim were false, so that $\mathbf{P}\left[\, f(x) = y \;:\; f \stackrel{R}{\leftarrow} F \,\right]$ is much larger than $\frac{1}{2^L}$. But we know, from the above that if f were random, the probability would be 2^{-L}. This means we have an event whose probability is different depending on which world we are in. But that means we have a means of telling which world we are in: just see whether the event happens, and, if so, bet on world 1 (ie. f is pseudorandom) because the event is more likely here; if not, bet on world 0 because the event is less likely here. Formally, define a distinguisher D as follows:

Distinguisher D^f
  Let $w \leftarrow f(x)$
  If $w = y$ then return 1 else return 0

Now we see that

$$\begin{aligned}
\mathbf{Succ}_F^{\mathrm{prf}}(D) &= \mathbf{P}\left[\, f(x) = y \;:\; f \stackrel{R}{\leftarrow} F \,\right] \\
\mathbf{Succ}_R^{\mathrm{prf}}(D) &= \mathbf{P}\left[\, f(x) = y \;:\; f \stackrel{R}{\leftarrow} R^{l,L} \,\right] .
\end{aligned}$$

We know the second quantity is 2^{-L}, so

$$\begin{aligned}
\mathbf{Adv}_F^{\mathrm{prf}}(D) &= \mathbf{P}\left[\, f(x) = y \;:\; f \stackrel{R}{\leftarrow} F \,\right] - \mathbf{P}\left[\, f(x) = y \;:\; f \stackrel{R}{\leftarrow} R^{l,L} \,\right] \\
&= \mathbf{P}\left[\, f(x) = y \;:\; f \stackrel{R}{\leftarrow} F \,\right] - \frac{1}{2^L} .
\end{aligned}$$

Or, rewriting,

$$\mathbf{P}\left[\, f(x) = y \;:\; f \stackrel{R}{\leftarrow} F \,\right] \;\leq\; \mathbf{Adv}_F^{\mathrm{prf}}(D) + \frac{1}{2^L} .$$

However, D makes only one oracle query. So we also know that

$$\mathbf{Adv}_F^{\mathrm{prf}}(D) \;\leq\; \mathbf{InSec}_F^{\mathrm{prf}}(t, 1)$$

as long as t is larger than the running time of D. The running time of D is very small. It suffices to set $t = O(l + L)$. Now combining the last two inequalities above we get Equation (5.1).

It is worth adding a word of caution. What is happening here is not going to *always* be true. Namely, not every event true for a random function happens with probability only slightly different for a pseudo-random function. Only certain kinds of events. Which kinds? The ones, intuitively, which can be "measured" by a distinguisher. In other words, ones for which a proof like the above works.

5.4.4 Substituting random functions by PRFs

The above examples, although simple, indicate roughly what we expect will happen when PRFs are substituted for random functions. The probability of an event changes by an amount proportional to the quality (security) of the PRF family. What we will see is that when we have some cryptographic scheme using a (shared, secret) random function, we will be able to view the success of the adversary as just such an event. When the PRF replaces the random function, the probability of this event does not grow too much, meaning if the adversary had small probability of success in the random case, this probability is still not too much. Put another way, if the probability of success of the adversary changes by too much, we would have a distinguisher.

Of course we must be careful to not blindly assume this is true no matter what the scheme. We must each time carefully go through a constructive argument (proof) like in the above examples. After a few of these however, you will get the feel and be able to tell, by looking at a scheme, whether or not the argument is likely to go through.

Namely, not every event true for a random function happens with probability only slightly different for a pseudo-random function. Only certain kinds of events. Which kinds? The ones, intuitively, which can be "measured" by a distinguisher. In other words, ones for which a proof like the above works.

5.4.5 The birthday attack

Suppose you are given an oracle for a function g, where g is either drawn randomly from a family of permutations, or g is a random function. How can you tell which? Query the oracle at successive points $x_1, x_2, \ldots$, and get back values $y_1, y_2, \ldots$. You know that if g were a permutation, the values $y_1, y_2, \ldots$ must be distinct. If g was a random function, they may or may not be distinct. So, if they are distinct, bet on a permutation.

Surprisingly, this is pretty good distinguisher, as we will argue below. The reason is the birthday paradox. If you are not familiar with this, you may want to look at the Appendix A.1 of these notes for a refresher, and then come back to the following, which invokes results from Appendix A.1.

Proposition 5.4.4 *[[12]] Let $P\colon \{0,1\}^k \times \{0,1\}^l \to \{0,1\}^l$ be any family of permutations. Then there is a distinguisher D, making q queries ($2 \le q \le 2^{(l+1)/2}$) and using $t = O(ql)$ time, such that*

$$\mathbf{Adv}_P^{\mathrm{prf}}(D) \quad \ge \quad 0.3 \cdot \frac{q(q-1)}{2^l} \, .$$

As a consequence

$$\mathbf{InSec}_P^{\mathrm{prf}}(t,q) \quad \ge \quad 0.3 \cdot \frac{q(q-1)}{2^l} \, .$$

Proof: The birthday attack works like this:

Distinguisher D^g
For $i = 1, \ldots, q$ do
  Let x_i be the i-th l-bit string in lexicographic order
  Let $y_i = g(x_i)$
End For
If $y_1, \ldots, y_q$ are all distinct output 1, else output 0

Clearly $\mathbf{Succ}_P^{\mathrm{prp}}(D) = 1$, because if g is a permutation then $y_1, \ldots, y_q$ are all distinct.

What if g is a random function? What is the chance that $y_1, \ldots, y_q$ are all distinct? This is the birthday problem. We are throwing q balls into $N = 2^l$ buckets and asking what is the chance of at least one collision. So we can use the Fact in Appendix A.1 to say that

$$\begin{aligned}\mathbf{Succ}^{\mathrm{prf}}_{R^{l,L}}(D) &= 1 - C(N,q) \\ &\leq 1 - 0.3 \cdot \frac{q(q-1)}{2^l} \,.\end{aligned}$$

Hence

$$\begin{aligned}\mathbf{Adv}^{\mathrm{prf}}_{P}(D) &= \mathbf{Succ}^{\mathrm{prf}}_{F}(D) - \mathbf{Succ}^{\mathrm{prf}}_{R^{l,L}}(D) \\ &\geq 1 - \left(1 - 0.3 \cdot \frac{q(q-1)}{2^l}\right) \\ &= 0.3 \cdot \frac{q(q-1)}{2^l} \,.\end{aligned}$$

Which was the claim. To obtain the second part, just take the maximum. ▮

5.4.6 PRFs versus PRPs

When we come to analyses of block cipher based constructions, we will find a curious dichotomy. Analyses are considerably simpler and more natural assuming the block cipher is a PRF. Yet, PRPs are what block ciphers more naturally approximate. To bridge the gap, we relate the two insecurity functions. The following says, roughly, that the birthday attack is the best possible one. A particular family of permutations P may have insecurity in the PRF sense that is greater than that in the PRP sense, but only by an amount of $q(q-1)/2^{l+1}$, the collision probability term in the birthday attack.

Proposition 5.4.5 *[[12]] Suppose $P\colon \{0,1\}^k \times \{0,1\}^l \to \{0,1\}^l$ is a family of permutations. Then*

$$\mathbf{InSec}^{\mathrm{prf}}_{P}(t,q) \;\leq\; \frac{q(q-1)}{2^{l+1}} + \mathbf{InSec}^{\mathrm{prp}}_{P}(t,q) \,.$$

Proof: Let A by any distinguisher for P versus $R^{l,l}$ that makes q oracle queries and runs for time at most t. We show that

$$\mathbf{Adv}^{\mathrm{prf}}_{P}(A) \;\leq\; \mathbf{Adv}^{\mathrm{prp}}_{P}(A) + \frac{q(q-1)}{2^{l+1}} \,. \tag{5.2}$$

The Proposition follows from the definitions of the insecurity functions.

Let $\overline{A}$ denote the distinguisher that first runs A to obtain an output bit b and then returns $\bar{b}$, the complement of b. Let $\mathbf{P}_1[\cdot]$ denote the probability under the experiment $f \stackrel{R}{\leftarrow} R^{l,l}$ and let $\mathbf{P}_2[\cdot]$ denote the probability under the experiment $f \stackrel{R}{\leftarrow} P^l$. Then

$$\begin{aligned}\mathbf{Adv}^{\mathrm{prf}}_{P}(A) &= \mathbf{P}\left[A^f = 1 \,:\, f \stackrel{R}{\leftarrow} F\right] - \mathbf{P}_1\left[A^f = 1\right] \\ &= \mathbf{P}_1\left[\overline{A}^f = 1\right] - \mathbf{P}\left[\overline{A}^f = 1 \,:\, f \stackrel{R}{\leftarrow} F\right] \\ &= \mathbf{P}_1\left[\overline{A}^f = 1\right] - \mathbf{P}_2\left[\overline{A}^f = 1\right] + \mathbf{P}_2\left[\overline{A}^f = 1\right] - \mathbf{P}\left[\overline{A}^f = 1 \,:\, f \stackrel{R}{\leftarrow} F\right] \\ &= \mathbf{P}_1\left[\overline{A}^f = 1\right] - \mathbf{P}_2\left[\overline{A}^f = 1\right] + \mathbf{P}\left[A^f = 1 \,:\, f \stackrel{R}{\leftarrow} F\right] - \mathbf{P}_2\left[A^f = 1\right] \\ &= \mathbf{P}_1\left[\overline{A}^f = 1\right] - \mathbf{P}_2\left[\overline{A}^f = 1\right] + \mathbf{Adv}^{\mathrm{prp}}_{P}(A) \,.\end{aligned}$$