7-21-2002

The Future of Cryptography Under Quantum Computers

Marco A. Barreno
Dartmouth College

Dartmouth College Computer Science Technical Report TR2002 - 425

The Future of Cryptography Under Quantum Computers

Marco A. Barreno
marco.barreno@alum.dartmouth.org

July 21, 2002

Senior Honors Thesis
Advisor Sean W. Smith

Contents

Abstract

Cryptography is an ancient art that has passed through many paradigms, from simple letter substitutions to polyalphabetic substitutions to rotor machines to digital encryption to public-key cryptosystems. With the possible advent of quantum computers and the strange behaviors they exhibit, a new paradigm shift in cryptography may be on the horizon. Quantum computers could hold the potential to render most modern encryption useless against a quantum-enabled adversary. The aim of this thesis is to characterize this convergence of cryptography and quantum computation.

We provide definitions for cryptographic primitives that frame them in general terms with respect to complexity. We explore the various possible relationships between **BQP**, the primary quantum complexity class, and more familiar classes, and we analyze the possible implications for cryptography.

Chapter 1

Preliminaries

1.1 Motivation

Cryptography is an ancient art that has passed through many paradigms, from simple letter substitutions to polyalphabetic substitutions to rotor machines to digital encryption to public-key cryptosystems. With the possible advent of quantum computers and the strange behaviors they exhibit, a new paradigm shift in cryptography may be on the horizon. Quantum computers may hold the potential to render most modern encryption useless against a quantum-enabled adversary. The aim of this thesis is to characterize this convergence of cryptography and quantum computation. We are not concerned so much with particular algorithms as with cryptography in general. To this end, we will examine primitives that constitute the core of modern cryptography and analyze the complexity-theoretical implications for them of quantum computation.

1.2 Overview

This chapter will be devoted to introducing the subjects to be discussed in this thesis. In Chapter 2 we define and analyze the basic cryptographic primitives, and in Chapter 3 we give an introduction to quantum computation and discuss some results that could have implications for cryptography. Chapter 4 is devoted to bringing together the cryptographic and quantum pieces and characterizing their intersection. Finally, in Chapter 5 we summarize our conclusions and suggest possible avenues for future work.

1.3 Introduction to cryptographic primitives

1.3.1 Basics and terminology

The term *cryptography* refers to the art or science of designing *cryptosystems* (to be defined shortly), while *cryptanalysis* refers to the science or art of breaking them. Although *cryptology* is the name given to the field that includes both of these, we will generally follow the common practice (even among many professionals and researchers in the field) of using the term "cryptography" interchangeably with "cryptology" to refer to the making and breaking of cryptosystems.

The main purpose of cryptography is to protect the interests of parties communicating in the presence of adversaries. A cryptosystem is a mechanism or scheme employed for the purpose of providing such protection. We examine several cryptosystems in this paper, spanning a wide range of cryptographic uses. We shall now take a moment to introduce the cryptographic primitives to be discussed. They will be formally defined and analyzed in Chapter 2. For a more comprehensive review of cryptographic concepts the reader is directed to Rivest's chapter in the *Handbook of Theoretical Computer Science* [20], and for a wide-ranging treatment of the application of those concepts the reader is referred to Schneier's book *Applied Cryptography* [22]. The definitions presented in Chapter 2, however, are meant to construct a more general complexity-theoretical framework for discussing the primitives than can be found currently in the literature.

1.3.2 Symmetric-key cryptography

Symmetric-key, or *secret-key*, cryptography is characterized by the use of one key, kept secret, that both parties in communication use to encrypt and decrypt messages. Modern symmetric-key cryptosystems come in two main flavors: *block ciphers* and *stream ciphers*. A block cipher operates on larger blocks of text (often 64-bit blocks), performing a particular scrambling function on the block. A simple block cipher will always encrypt the same plaintext block to the same ciphertext block, though more advanced techniques such as *block chaining* can negate this effect. A stream cipher, on the other hand, operates on smaller units—often just one byte or one bit at a time—and produces an output stream in which the encryption of each unit of ciphertext depends on the sequence of units for some length before it.

The same piece of plaintext will generally encrypt to a different ciphertext at different times. A stream cipher is conceptually very similar to a pseudo-random number generator (see below) and, in fact, is often implemented in the same way.

The main purpose of such a cryptosystem is, of course, to thwart an adversary in his or her attempt to intercept or disrupt communications. The adversary may have various types of information available with which to attack the cryptosystem. In a *ciphertext-only* attack, the adversary knows nothing but a number of ciphertexts polynomial in the input size (the input size is the sum of the sizes of the key and message). Note that in this paper we always assume that the adversary has full knowledge of the algorithm used. In a *known-plaintext* attack, the adversary has access to a polynomial number of plaintext-ciphertext pairs. In a *chosen-ciphertext* attack, the adversary may select a polynomial number of ciphertexts for which to see the plaintext. One might also encounter *adaptive chosen-plaintext* or *adaptive chosen-ciphertext* attacks, in which the adversary need not choose all the plaintexts or ciphertexts at once but may see some results before making further selections. For simplicity's sake, we do not address the additional complexity of these last three attacks, and we concern ourselves here with ciphertext-only and known-plaintext attacks.

1.3.3 One-way hash functions

Informally, a *one-way function* is a function that is easy to compute but difficult to invert. We are concerned primarily with cryptographically relevant one-way functions, and these tend to fall into two major categories: *one-way hash functions* and *trapdoor functions*. We will discuss trapdoor functions in the context of public-key cryptography, but here we introduce one-way hash functions.

Various other properties are sometimes associated with the concept of a one-way function, such as that it must be one-to-one [11, 18] or that it must be *honest*, meaning that for any x in the domain, $f(x)$ may be no more than polynomially smaller than x [11, 15]. A one-way hash function used in cryptographic applications, such as MD5 or SHA, generally has neither of these properties. Its purpose is to create a smaller, usually fixed-size value such that it is difficult to find a message that hashes to any particular value, or even any two messages that hash to the same value. Because the message space tends to be much bigger than the space of hash values, the hash function is not one-to-one, and it clearly cannot be honest with a fixed-size output. For a detailed look at cryptographic one-way hash functions, including myriad real-world examples, see Schneier's book [22, Chapter 18]. Although there are differing opinions on just what should constitute a one-way function, we will attempt to make some generalizations and draw conclusions relevant to cryptography.

1.3.4 Trapdoor functions and public-key cryptography

A trapdoor function is a one-way function with a corresponding piece of information (the trapdoor) that helps one easily to compute the inverse of the function. Trapdoor functions are crucial to public-key cryptography.

Public-key cryptography was conceived by Diffie and Hellman in 1976 [7], though Merkle had previously developed some of the key concepts [16]. It is characterized by different encryption and decryption keys; each user makes his or her encryption key publicly available but keeps the decryption key secret. Anyone can encrypt messages using any public key, but the ciphertexts can be decrypted only by the user possessing the decryption key corresponding to the key used for encryption. This is the trapdoor function: encryption is the one-way operation, and the private key is the trapdoor information allowing the user to invert the function and decrypt messages. The best-known example is the RSA cryptosystem, so named for the initials of its inventors Rivest, Shamir, and Adleman [21]. RSA uses modular exponentiation as the trapdoor function, and its difficulty is based on the difficulty of factoring large numbers.

1.3.5 Digital signing

The objective of *digital signing* is to provide a means by which it can be proved that a person has seen and acknowledged a particular document. Each signature must be associated, with high probability, with one particular person and one particular document. A digital signature can also act as proof of identity because only the person possessing the correct private key can generate signatures verified by the corresponding public key.

Digital signing is closely related to public-key cryptography. Many public-key cryptosystems can also be used as digital signature system by simply reversing the order of operations: "encrypt" using the private key to generate the signature, and verify it by "decrypting" with the public key; this works because the operations are inverses of each other. Only the possessor of the particular private key can generate a signature that is correctly verified by the corresponding public key, and the signature for each document is different (again, with high probability).

1.3.6 Pseudorandom number generation

Of crucial importance to many cryptographic applications is a source of randomness. Because natural randomness is somewhat difficult to come by in large amounts, it is important to design *pseudorandom number generators* to supply numbers that appear to be random. The appearance of randomness is usually defined by difficulty of predicting the next number (or bit), given the ones produced so far.

1.4 Complexity theory

The analysis of computational resources required to solve problems is the realm of *complexity theory*, pioneered in 1965 by Hartmanis and Stearns [13]. Complexity theory is concerned with comparing the inherent difficulty of computational problems. The salient measure is the asymptotic time or space required of an algorithm in terms of some size parameter n of the input. An algorithm runs in, say, $O(n^2)$ time (pronounced "big-oh of n squared") if its running time can be bounded asymptotically by some constant multiple of n^2. In general, the set of functions $f(n)$ obeying a particular asymptotic bound $g(n)$ can be denoted as follows:

$$O(g(n)) = \{f(n) : \begin{array}{l} \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\} \end{array}$$

This definition comes directly from *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein [5], in which the reader will also find further discussion of asymptotic notation and the growth of functions.

1.4.1 Overview of complexity classes

When discussing time complexity of algorithms or problems (or space complexity, but in this paper we are concerned primarily with time complexity) it is useful to group them into *complexity classes*, or classes of problems that share the same asymptotic upper limit on running time for a particular model of computation. Examples of such limits include:

- constant time: $O(1)$
- linear time: $O(n)$
- polynomial time: $O(n^k)$ for some constant $k > 0$
- exponential time: $O(2^{n^k})$ for some constant $k > 0$

Examples of models of computation, to be discussed below, include deterministic Turing machines, probabilistic Turing machines, nondeterministic Turing machines, oracle Turing machines, and quantum computers. We present here a brief overview of some relevant complexity classes; for a thorough treatment of non-quantum complexity classes, please see Johnson's chapter of the *Handbook of Theoretical Computer Science* [15].

The basic division between tractable and intractable problems is quite universally held to be the line between polynomial and exponential time. Exactly which problems are solvable in polynomial time and which in exponential depends somewhat on the model of computation at one's disposal when solving the problem.

Deterministic Turing machines

We assume the reader has at least a basic familiarity with Turing machines; see, for example, Sipser's book [25] or Hopcroft, Motwani, and Ullman's book [14] for a comprehensive introduction. The complexity class **P** is the class of problems solvable on a deterministic Turing machine in polynomial time. Because deterministic Turing machines are essentially equivalent to digital computers, provided the computer has enough memory to be treated as infinite for the given problem, **P** is sometimes taken to be the class of tractable problems (especially when contrasted with the class **NP**).

Nondeterministic Turing machines

A nondeterministic Turing machine is a Turing machine that can make nondeterministic guesses during computation. The effect is that when an NTM (we will often abbreviate various Turing machines by their initials like this) makes such a guess, it essentially follows both (or all if more than two) execution paths and accepts the input if any execution path enters an accepting configuration.

Bounded probabilistic Turing machines

A probabilistic Turing machine is a Turing machine that is deterministic except that it can employ a source of randomness, such as flipping a coin, in making decisions. The complexity class **BPP** is the class of problems solvable in polynomial time by probabilistic Turing machines with bounded error probability. A probabilistic Turing machine has bounded error probability if the probability that it yields an incorrect answer is uniformly bounded below $\frac{1}{2}$ for all inputs; in other words, if the input is in the language then the BPTM accepts with probability strictly greater than $\frac{1}{2}$ and if the input is not in the language then it rejects with probability strictly greater than $\frac{1}{2}$. In either case, the lengths of all computations are on the same order. **BPP** is introduced and analyzed in Gill's 1977 paper [10]. In reality, **BPP** is usually held to be a better description of tractable problems than **P** because a source of true or good-enough approximate randomness is obtained easily enough.

Unambiguous Turing machines

An unambiguous Turing machine is simply an NTM with at most one accepting configuration for each possible input string. The class of problems solvable in polynomial time by a UTM is **UP**. This class is important because it is closely tied to the existence of one-way functions, as we shall see later on in Section 4.1.1.

Quantum Turing machines

A quantum Turing machine is a Turing machine that can use quantum mechanical operations in performing calculations. **BQP** is the class of problems solvable in bounded-error polynomial time by QTMs, analogous to **BPP** for non-quantum machines. We will be exploring QTMs in more depth in Chapter 4; for background and definitions, see papers by Deutsch [6] and Bernstein and Vazirani [2].

Oracle Turing machines

An oracle Turing machine is a Turing machine with a special oracle tape. The Turing machine can ask a question of the oracle by writing to the oracle tape and then entering a special oracle state; after a single time step, the answer will replace the question on the oracle tape. The oracle can be thought of as a problem the Turing machine gets to solve for free.

When a proof is given that involves an OTM, it is an example of *relativized complexity*, or complexity analysis relative to an oracle. There are at least two reasons why such proofs can be interesting. First, the relativized proof becomes a non-relativized proof if ever a tractable algorithm is devised to perform the same function as the oracle. And second, since many proof techniques relativize, that is, remain valid when applied in a relativized setting, it can be useful to demonstrate different oracles relative to which open questions are answered in different ways. If this can be done it means that proving the question one way or another will be difficult and require unusual technique

If O is an oracle, we denote by M^O an oracle Turing machine that can query O in its computations.

There is one more class that we will mention: **PSPACE** is the class of problems that take polynomial *space* to solve (and unspecified time). It is well known that $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$ and $\mathbf{BPP} \subseteq \mathbf{PSPACE}$, but it is not known whether any of those inclusions are proper.

Note that there is a distinction to be drawn among what we have been referring to generally as "problems." Turing machines recognize languages, or, equivalently, solve decision problems. Given a string, a Turing machine will return *accept* or *reject*. Most of the problems we are concerned with, however, are not decision problems but functions that produce a value other than merely accept or reject. These functions are described by classes **FP** analogous to **P**, **FNP** analogous to **NP**, and so on (Grollmann and Selman [11] refer to **FP** as **PSV**, **FNP** as **NPSV**, etc.). See Papadimitriou's book [18] for further explanation and analysis of this distinction. In this paper, we will not distinguish between classes of languages and classes of functions until it becomes crucial to draw the distinction clearly for Theorem 2.

1.4.2 Hard problems vs. easy problems

We have some division between "easy" and "hard" complexity classes. So far this has been the division between polynomial and exponential classes, and

BPP has replaced **P** as the main polynomial class. Now we want to make this bound movable. The hard of tomorrow may be more restricting than the hard of today, but we want these definitions to withstand the shift.

When talking about cryptography and complexity-theoretic security, we need to define what is considered to be "hard" and what is "easy." Generally the line between tractable and intractable has been taken to be the polynomial/exponential line. That is, a problem is considered tractable if the running time of an algorithm to solve it is $O(n^k)$ for some constant k, and a problem is considered intractable if it cannot be bound by such a limit. Though there are classes between polynomial and exponential, by far the most commonly discussed superpolynomial bound is exponential time.

Classifying decision problems as easy and hard by these standards depends on the model of computation used, of course. Exactly which problems are solvable in polynomial time depends on whether you can make coin flips, choose nondeterministically, etc.

In this paper we shall use the notation $\textsc{Easy}(n)$ to refer to classes that are feasible as n grows large and $\textsc{Hard}(n)$ to refer to classes that are infeasible as n grows large, judging by the most powerful model(s) of computation available. In Chapter 4 we will examine which problems will be in $\textsc{Easy}(n)$ and which will be in $\textsc{Hard}(n)$ if a quantum computer is built and quantum computing becomes available as a model of computation.

1.5 Quantum computers

In a nutshell, a *quantum computer* is a computer built to make use of quantum mechanical effects in its computations. No one has yet succeeded in building a quantum computer of significant size, and indeed there are fundamental difficulties that may prevent a large-scale quantum computer from ever being built. If a quantum computer were built, however, it would have powers exceeding the known powers of a classical (i.e. non-quantum) computer, due to the quantum mechanical effects. In particular, it has been shown that some problems critical to cryptography (to be discussed in Chapter 3) can be solved on a quantum computer in much less time than the best known time for a classical algorithm, suggesting that the advent of large-scale quantum computers may have very significant implications for the field of cryptography. Exploring the extent and nature of these implications is the main purpose of this thesis.

Chapter 2

Complexity-Generalized Cryptography

Before we discuss in detail the effects quantum computers will have on cryptography, it is necessary to define and review some important cryptographic concepts. In this section we will present some fundamental elements of cryptography as they are relevant to the subject. We assume the reader has a basic familiarity with cryptography, but we will review the key details.

It is important to keep in mind that the security of the systems we are concerned with is measured in the sense of computational complexity, not the information-theoretic sense. A cryptosystem is *information-theoretically secure* if the ciphertext (along with knowledge of the algorithm) does not give the adversary enough information to find the plaintext. The standard example of such a system is the one-time pad, under which each message is xor'd with a different random key of the same length as the message. Since any plaintext of that length could encrypt to the same ciphertext, given the appropriate key, the adversary cannot determine any information about the message (other than perhaps the length). A cryptosystem is still *computationally secure*, on the other hand, even if an adversary has enough information to recover the message in theory but the computation requires too much time to be feasible.

Primitive	Definition	Security & feasibility
Symmetric cryptography	Section 2.2.1	Section 2.3.1
One-way hash functions	Section 2.2.2	Section 2.3.2
Public-key cryptography	Section 2.2.3	Section 2.3.3
Digital signing	Section 2.2.4	Section 2.3.4
Pseudorandom number generation	Section 2.2.5	Section 2.3.5

Table 2.1: Cryptographic primitives and sections in which discussed and analyzed

2.1 Complexity and cryptography

Central to measuring the success of a cryptosystem is assessing the ease of using it and difficulty of breaking it. Complexity theory is the language we use to do this, but we take what we believe to be a novel approach in our treatment of the subject. This chapter presents a formulation of some important cryptographic primitives that frees them from being tied to any particular model of computing or any specific notion of easy and hard. We discuss using and breaking these primitives independently of a particular easy-hard boundary so that the discussion will be germane to any model of computation.

First we lay out the definitions of the primitives. After we have established our definitions, we will present a brief analysis of the feasibility of using the primitives and difficulty of breaking them in terms of our complexity-generalized definitions.

2.2 Definitions of cryptographic primitives

Wherever $\mathcal{M}$ and $\mathcal{C}$ appear in definitions, they should be taken to be the *message space* and *ciphertext space*, respectively, each a set of strings over some alphabet (not necessarily the same).

The cryptographic primitives we will discuss are listed in Table 2.1, which summarizes where the definitions and complexity analysis can be found for each.

2.2.1 Definition of symmetric-key cryptography

We begin with our complexity-generalized definition of a symmetric cryptosystem.

Primitive Definition 1 *Symmetric* $= \{f, g, \mathcal{M}, \mathcal{K}, \mathcal{C}\}$ *such that:*

- $f : \mathcal{M} \times \mathcal{K} \longrightarrow \mathcal{C}$ *and* $g : \mathcal{C} \times \mathcal{K} \longrightarrow \mathcal{M}$ *are the encryption and decryption functions, respectively*
- $\mathcal{K}$ *is the* key space*, a set of strings over some alphabet*
- $g(f(m,k),k) = m$ *for all* $k \in \mathcal{K}, m \in \mathcal{M}$

An instance of a symmetric cryptosystem consists of encrypting and decrypting functions (which may be the same function), a message space, a key space, and a ciphertext space. In order to discuss the cracking problem, we must first introduce some additional concepts.

Definition 1 *For any message space* $\mathcal{M}$*, let* $\mathcal{M}_{\emptyset} = \mathcal{M} \cup \{\emptyset\}$*, where* $\emptyset$ *means no message at all.*

We define $\mathcal{M}_{\emptyset}$ to be the space of possible messages for a given message space augmented with $\emptyset$, or "no message." This should be taken to mean that an element $m \in \mathcal{M}_{\emptyset}$ could be any message in $\mathcal{M}$ or simply no message at all.

When we say that an adversary has cracked a cryptosystem, we mean that the adversary can decrypt messages encrypted under the cryptosystem without prior knowledge of the key. This requires being able to identify the particular key used to encrypt a given message. It is not cracking a cryptosystem for an adversary to "decrypt" a message with some key that was not used to encrypt it and get a "plaintext" that is not the one encrypted and gives no information about the real plaintext. However it is done, then, the correct key must be identified. In order to capture this requirement without being concerned with the details, we define an oracle.

Definition 2 (Identification Oracle) *Given an instance of a primitive, such as Symmetric, the identification oracle* $\mathcal{I}$ *identifies a particular key, or message in the case of OneWayHash. A Turing machine* $M^{\mathcal{I}}$ *may query the oracle with a key, or a message, and the oracle will answer* ***True*** *if the given key or message is "the one we are after."*

This definition is left intentionally general because the particulars do not concern us at this time. Whether this can easily be implemented as a real test rather than an oracle query depends on the type of attack. For a ciphertext-only attack in which the attacker knows something about the structure of the message and has enough ciphertext, this test returns **True** if the decryption for the key in question is intelligible (i.e. fits the known structure). An obvious case is when the message is known to be, say, English text in ASCII; $\mathcal{I}$ would return **True** if decryption with the key in question produced a message recognizable as English. In a known-plaintext attack, the test returns **True** if it decrypts the given ciphertext into the corresponding plaintext.

Whether the attacker has "enough ciphertext" in a ciphertext-only attack is measured by *unicity distance*, introduced by Shannon in his 1949 paper [23]. The unicity distance for a message with a certain structure is the message length needed to guarantee with high probability that there is only one plaintext that could produce the given ciphertext with any key. The unicity distance for ASCII English text encrypted with various algorithms ranges from about 8.2 to 37.6 characters for keys of length 56 to 256 bits [22, p.236], so for this case it may be reasonable to assume that most messages under consideration will be longer than the unicity distance and the test can be performed without difficulty.

Note that the unicity distance test is meaningless for a one-time pad for the same reasons that any ciphertext-only attack against a one-time pad is theoretically impossible; namely, a particular ciphertext could be produced by *any* message of the correct length given the right key, so there is no way to distinguish one key from another if each decrypts the ciphertext to a message that fits the structure.

We now turn to the cracking problem for a symmetric cryptosystem.

Definition 3 *The cracking problem*

$$\mathit{crack[Symmetric]}^{\mathcal{I}} : \{\mathcal{C} \times \mathcal{M}_{\emptyset}\}^{+} \longrightarrow \mathcal{G},$$

given an instance of Symmetric and relative to an oracle $\mathcal{I}$, takes a polynomial number p of ciphertext/plaintext pairs and produces a function $g' \in \mathcal{G} : \mathcal{C} \longrightarrow \mathcal{M}$ that can then be used to decipher enciphered messages. The cracking problem is to compute $\mathit{crack[Symmetric]}^{\mathcal{I}}(c_1, m_1, c_2, m_2, \ldots, c_p, m_p) = g'$

such that:

$$\begin{aligned}&\exists(k \in \mathcal{K})\forall(c \in \mathcal{C})\exists(m \in \mathcal{M}) :\\ &\qquad (\mathcal{I}(k) = \textbf{\textit{True}}) \wedge (m = g(c,k)) \wedge (m = g'(c))\end{aligned}$$

There is an acknowledged drawback to this definition of the cracking problem: it presents an all-or-nothing approach. A real cryptosystem would be considered compromised if the adversary could reliably decrypt half of the messages but not all of them, though such a situation does not count as cracking under our definition. This is one area in which further work could extend the results presented here.

2.2.2 Definition of one-way hash functions

A *one-way hash function* $f(x)$ computes a hash value h (often but not necessarily of fixed length) for any x in the domain. A value x is said to be the *pre-image* of h if $f(x) = h$. The purpose of a one-way hash function is to produce a value h that can be treated as uniquely associated with pre-image x for practical uses. A one-way hash function is called *collision-free* if it is hard to find two pre-images w and y such that $f(w) = f(y)$. Remember that most cryptographically significant real-world hash functions are neither one-to-one nor honest (see Section 1.3.3).

Here $\mathcal{Z}^+$ is the set of positive integers.

Primitive Definition 2 *OneWayHash* $= \{f, \mathcal{D}, \mathcal{R}, l\}$ *such that:*

- $f : \mathcal{D} \times \mathcal{Z}^+ \longrightarrow \mathcal{R}$ *is the hash function*
- $\mathcal{D}$ *is the domain of* f
- $\mathcal{R}$ *is the range of* f
- f *is not necessarily one-to-one but* f *is onto*
- $l \in \mathcal{Z}^+$ *is the length of the hash value:* $|f(x,l)| = l$

Cracking a one-way hash function means finding any pre-image that hashes to a particular hash value. Note that this does not mean that a hash function must be completely collision-free to be uncrackable by our definition, but only that a collision for a particular hash value cannot easily be found. The cracking function takes as input the hash value to be cracked as well as a polynomial number of generated message/hash value pairs.

Definition 4 *The cracking problem*

$$crack[OneWayHash]^{\mathcal{I}} : \mathcal{Z}^{+} \times (\mathcal{D} \times \mathcal{R})^{*} \longrightarrow \mathcal{G},$$

given an instance of OneWayHash and relative to an oracle $\mathcal{I}$, takes a length and a polynomial number p of message/hash value pairs, where all hash values have the given length. It produces a function $g' \in \mathcal{G} : \mathcal{R} \longrightarrow \mathcal{D}$ that returns a pre-image for the given hash value of the correct length (and is undefined for r of any other length). The cracking problem is to compute $crack[OneWayHash]^{\mathcal{I}}(l, d_1, r_1, d_2, r_2, \ldots, d_p, r_p) = g'$ such that:

$$\forall (r \in \mathcal{R} \textit{ where } |l| = r) \exists (d, d' \in \mathcal{D}) : \\ (\mathcal{I}(d) = \textbf{\textit{True}}) \wedge (r = f(d, l)) \wedge (r = f(d', l)) \wedge (d' = g'(r))$$

Sometimes the notion of cracking a hash function is broadened to include findinga *claw*, or any two messages that produce the same hash value. A hash function is *claw-free* if for that hash function it is infeasible to find any two pre-images that hash to the same hash value. We do not require the function to be claw-free because it would further complicate matters; this may be another avenue for future work.

2.2.3 Definition of public-key cryptography

A *trapdoor function* is a one-way function with a special property: there exists some information that allows anyone who knows the information to invert the function easily, while that inversion is hard without knowledge of this secret trapdoor information. Public-key cryptosystems are essentially trapdoor functions: anybody can encrypt a message that only the intended recipient can read because that recipient has the trapdoor information necessary to invert the encryption. It is important to note here that trapdoor functions have not been proven to exist; rather, like most complexity-theoretic issues in cryptography, given the current evidence it is likely that they exist and therefore that public-key cryptography is possible (for more on the existence of one-way functions, see Section 4.1.1).

Before defining public-key cryptosystems, we first discuss key pairs. The key pair defines the trapdoor function for a cryptosystem: the public key parameterizes a one-way function for encryption, while the private key constitutes the trapdoor information that allows the recipient to invert the function.

Note that this definition refers to f and g, which are, respectively, the encryption and decryption functions parameterized by the keys.

Definition 5 *A key pair space $\mathcal{K}_p[f, g] = \{k_f \in \mathcal{K}_f, k_g \in \mathcal{K}_g\}$ is the set of all* valid *key pairs for functions $f : \mathcal{D}_f \times \mathcal{K}_f \longrightarrow \mathcal{R}_f$ and $g : \mathcal{D}_g \times \mathcal{K}_g \longrightarrow \mathcal{R}_g$, where $\mathcal{D}_f$ and $\mathcal{D}_g$ are the domains (excluding the key parameter) and $\mathcal{R}_f$ and $\mathcal{R}_g$ are the ranges of f and g, respectively. The exact definition of validity depends on the application, but generally each key pair must exhibit behavior with f and g both* correct *for the application and* unique *to that key pair. k_f and k_g must be similar in size so that $|k_f| = O(|k_g|)$.*

The keys k_f and k_g parameterize the encrypting and decrypting functions, respectively.

We are now ready to define public-key cryptosystems.

Primitive Definition 3 *PublicKey* $= \{f, g, \mathcal{M}, \mathcal{C}, \mathcal{K}_p[f, g]\}$ *such that:*

- *$f : \mathcal{M} \times \mathcal{K}_f \longrightarrow \mathcal{C}$ and $g : \mathcal{C} \times \mathcal{K}_g \longrightarrow \mathcal{M}$ are the public encryption and decryption functions, respectively ($\mathcal{M}$ here corresponds to $\mathcal{D}_f$ in the key pair definition, and $\mathcal{C}$ to $\mathcal{D}_g$)*
- *$\mathcal{K}_p[f, g]$ is the space of valid encryption/decryption key pairs ($k_e \in \mathcal{K}_f, k_d \in \mathcal{K}_g$) where k_e is the public encryption key and k_d is the private decryption key*
- *A key pair (k_e, k_d) is valid for functions f and g if both correctness and uniqueness hold.*
  - *(Correctness)*

$$\forall (m \in \mathcal{M}) : (g(f(m, k_e), k_d) = m)$$

$$\forall (c \in \mathcal{C}) : (f(g(c, k_d), k_e) = c)$$

  - *(Uniqueness)*

$$\forall (m \in \mathcal{M}, k_d' \neq k_d \in \mathcal{K}_g) : (g(f(m, k_e), k_d') \neq m)$$

$$\forall (c \in \mathcal{C}, k_e' \neq k_e \in \mathcal{K}_f) : (f(g(c, k_d), k_e') \neq c)$$

At the heart of every public-key cryptosystem is a trapdoor function (or function pair: f and g may or may not be the same function) parameterized by a key pair in some form. Note that this definition is general enough to consider function pairs as key pairs: k_f and k_g can be functions and f and g can merely apply the given function to the given message or ciphertext.

The correctness criterion for valid key pairs stipulates that encrypting a message with f, k_e and then decrypting it with g, k_d will reproduce the original message, and vice-versa. The uniqueness criterion requires that no two keys encrypt any one message to the same ciphertext (or decrypt any one ciphertext to the same message).

Definition 6 *The cracking problem*

$$\textit{crack[PublicKey]}^{\mathcal{I}} : \mathcal{K}_f \longrightarrow \mathcal{G},$$

given an instance of PublicKey, takes a public key and produces a function $g' \in \mathcal{G} : \mathcal{C} \longrightarrow \mathcal{M}$ that can then be used to decipher enciphered messages. The cracking problem is to compute $\textit{crack[PublicKey]}^{\mathcal{I}}(k_f) = g'$ such that:

$$\begin{aligned}&\exists(k_g \in \mathcal{K}_g)\forall(c \in \mathcal{C})\exists(m \in \mathcal{M}) : \\ &\qquad (\mathcal{I}(k_g) = \textbf{\textit{True}}) \wedge (m = g(c, k_g)) \wedge (m = g'(c))\end{aligned}$$

This cracking problem is very similar to crack[Symmetric]$^{\mathcal{I}}$. One notable difference is in $\mathcal{I}$. Here, identifying the correct decryption key is trivial: since the encryption key is public, one merely encrypts any message and tests whether the supposed decryption key correctly deciphers it. The identification oracle in this case, then, takes a public key as input.

2.2.4 Definition of digital signing

The concept of a digital signature is closely tied to public-key cryptosystems. Here again, each user has a public key and a private key, and the intent is that anyone can create a digital signature uniquely identifying himself or herself, which can be verified by anybody with the public key and forged by nobody without the private key.

Primitive Definition 4 *DigitalSign* $= \{f, g, \mathcal{M}, \mathcal{S}, \mathcal{K}_p[f, g]\}$ *such that:*

- $f : \mathcal{M} \times \mathcal{S} \times \mathcal{K}_g \longrightarrow \{\textbf{\textit{True}}, \textbf{\textit{False}}\}$ *and* $g : \mathcal{M} \times \mathcal{K}_f \longrightarrow \mathcal{S}$ *are the public verifying and signing functions, respectively*

- $\mathcal{M}$ *is the message space*
- $\mathcal{S}$ *is the signature space*
- $\mathcal{K}_p[f,g]$ *is the space of valid verifying/signing key pairs* $(k_v \in \mathcal{K}_f, k_s \in \mathcal{K}_g)$ *where* k_v *is the public verifying key and* k_s *is the private signing key*
- *A key pair* (k_v, k_s) *is valid for functions* f *and* g *if for all* $m, m_1 \in \mathcal{M}, s \in \mathcal{S}, k'_s \in \mathcal{K}_g$*:*

  (Correctness)

  - $(f(m, g(m, k_s), k_v) = \textbf{\textit{True}})$
- *It is also desirable for the key pair to have the uniqueness property:*

  (Uniqueness)

  - $(m_1 \neq m) \vee (k'_s \neq k_s) \Rightarrow (g(m_1, k'_s) \neq g(m, k_s))$
  - $(s \neq g(m, k_s)) \Rightarrow (f(m, s, k_v) = \textbf{\textit{False}})$

The correctness criterion requires all verifying keys to correctly verify signatures created with their corresponding signing keys. The uniqueness criterion specifies that no verifying key can produce a false positive, or **True** result for a signature generated with either another message or another signing key. The uniqueness criterion is usually slightly relaxed in practice, though with the intent that it be infeasible for an adversary to take advantage of this relaxation.

Many public-key cryptosystems can also function as digital signature schemes.[1] To sign a message, a user "decrypts" it with his or her private key. Any other user can then verify the signature by "encrypting" it and comparing the result to the original message. This scheme has some problems, however, including a signature that is as long as the original message. A slight modification makes this practice much more useful, though it sacrifices perfect uniqueness. By first using a one-way hash function to obtain a hash value h for the message and then signing h, a user can produce a useful signature much shorter (in most cases) than the message, while preserving

[1]Note: The Diffie-Hellman key exchange protocol[7] cannot be used as a signature scheme, but it also does not fit under our definition of PublicKey.

the useful properties of the signature. Uniqueness will be compromised to the extent that collisions of the hash function can be found.

Definition 7 *The cracking problem*

$$crack[DigitalSign]^{\mathcal{I}} : \mathcal{K}_f \longrightarrow \mathcal{G},$$

given an instance of DigitalSign, takes a public key and produces a function $g' \in \mathcal{G} : \mathcal{M} \longrightarrow \mathcal{S}$ *that can then be used to forge signatures. The cracking problem is to compute* $crack[DigitalSign]^{\mathcal{I}}(k_v) = g'$ *such that:*

$$\begin{aligned}&\exists(k_g \in \mathcal{K}_g)\forall(m \in \mathcal{M}) :\\&\quad(\mathcal{I}(k_g) = \textbf{\textit{True}}) \wedge (f(m, g(m, k_g), k_v) = \textbf{\textit{True}})\\&\qquad\qquad \wedge(f(m, g'(m), k_v) = \textbf{\textit{True}}))\end{aligned}$$

DigitalSign is cracked if the adversary can forge signatures that appear to be genuine. (Note that this definition does not explicitly consider finding a second message that has the same signature as a given message; while such a finding could be a useful attack, it is out of the scope of this analysis.) If the uniqueness criterion of signatures holds, then this can only be accomplished by duplicating the signature for each message exactly. If the uniqueness criterion does not hold, however, then it may be possible to find alternate signatures that seem to be genuine. For example, if DigitalSign were composed of a public-key cryptosystem and a one-way hash function, as described above, collisions in the hash function might lead to different messages producing the same signature when signed with the same key.

2.2.5 Definition of pseudorandom number generation

In defining pseudorandom number generators, we actually define pseudorandom binary bit generators. A sequence of bits is more useful cryptographically because it can be directly employed in the creation of a digital one-time pad, and it can easily be converted into a number sequence by grouping bits into binary numbers of the desired length.

Here $\mathcal{N}$ is the set of natural numbers.

Primitive Definition 5 *PseudoRandom* $= \{f, \mathcal{K}\}$ *such that:*

- $f : \mathcal{K} \times \mathcal{N} \longrightarrow \{0, 1\}^*$ *is the pseudorandom function*

- *$\mathcal{K}$ is the key space, a set of strings over some alphabet*
- *$f(k,p) = x_1x_2x_3 \ldots x_p$ for $k \in \mathcal{K}$, $p \in \mathcal{N}$, and $x_i \in \{0,1\}$. Let $f(k,p)_i$ denote x_i.*

The key for a pseudorandom number generator is the *seed* for the generation process. The output of f is a string of bits. The salient feature of the string, of course, is that it is hard to predict bit x_i given bits $x_1 \ldots x_{i-1}$ without knowledge of the seed. In general, the seed should ideally be a truly random string of bits; the pseudorandom number generator functions as a randomness expander and increases the length of the sequence without significantly increasing the feasibility of predicting the next bit.

Cracking a pseudorandom number generator, of course, involves being able to predict the next bit.

Definition 8 *The cracking problem*

$$\textit{crack[PseudoRandom]}^{\mathcal{I}} : \{0,1\}^* \times \mathcal{N} \longrightarrow \mathcal{G},$$

given an instance of PseudoRandom, takes a sequence of bits and the number of bits p in the sequence and produces a function $g' \in \mathcal{G} : \mathcal{N} \longrightarrow \{0,1\}$ that can then be used to predict any bit of a pseudorandom sequence up to the $(p+1)$th bit. The cracking problem is to compute

$$\textit{crack[PseudoRandom]}^{\mathcal{I}}(\{0,1\}^p, p) = g'$$

such that:

$$\exists(k \in \mathcal{K})\forall(p, q \leq p \in \mathcal{N}) : (\mathcal{I}(k) = \textbf{\textit{True}}) \wedge ((q \leq p+1) \Rightarrow (f(k,p)_q = g'(q)))$$

with probability greater than $\frac{1}{2} + \epsilon$ for some small ϵ.

The identification oracle here identifies the seed used to generate the pseudorandom bit sequence.

2.3 Complexity-generalized requirements for security and feasibility

In this section we discuss the complexity of cryptographic operations in terms of the size of the input, represented by n. In general, the size parameter n is length of the key. We will specify n for each primitive.

The complexity of each primitive has two parts: feasibility and security. The security requirement for each primitive is that its corresponding cracking program be in $\textsc{Hard}(n)$ for its size parameter n; this is what is necessary for the user's goals to be protected from an adversary's intervention. The feasibility requirements, if met, ensure that the primitive is usable; that is, the functions that make up the primitive must be in $\textsc{Easy}(n)$. In practice, it is relatively quite easy to create systems that meet the feasibility requirements, though ensuring that systems meet the security requirements can be tricky to impossible.

For each primitive we describe the size parameter and the particular feasibility requirements, as well as restate the security requirement.

2.3.1 Complexity requiremens of Symmetric

Size parameter

The size parameter for Symmetric is $n = |k|$.

Feasibility

Symmetric is feasible if both f and g are in $\textsc{Easy}(n)$.

Security

Symmetric is secure if crack[Symmetric]$^{\mathcal{I}}$ is in $\textsc{Hard}(n)$.

It is important to keep in mind that the actual computation times may depend on more than just this size parameter—for example, computation time for using Symmetric depends on the length of the message in some way, though the size parameter is only the length of the key. But the size parameters we focus on here are the significant ones for security: if using Symmetric is in $\textsc{Easy}(|k|)$ and breaking Symmetric is in $\textsc{Hard}(|k|)$ then the user can significantly increase security without significantly impacting time of use by increasing $|k|$.

2.3.2 Complexity requirements of OneWayHash

Size parameter

The size parameter for OneWayHash is $n = l$.

Feasibility

OneWayHash is feasible if $f(m)$ is in $\textsc{Easy}(n)$

Security

OneWayHash is secure if crack[OneWayHash]$^{\mathcal{I}}$ is in $\textsc{Hard}(n)$.

2.3.3 Complexity requirements of PublicKey

Size parameter

The size parameter for PublicKey is $n = |k_e|$. Note that $|k_e| = O(|k_d|)$.

Feasibility

PublicKey is feasible if f, g, and generating a valid key pair (k_f, k_g) are all in $\textsc{Easy}(n)$.

Security

PublicKey is secure if crack[PublicKey]$^{\mathcal{I}}$ is in $\textsc{Hard}(n)$.

2.3.4 Complexity requirements of DigitalSign

Size parameter

The size parameter for DigitalSign is $n = |k_v|$. Note that $|k_v| = O(|k_s|)$.

Feasibility

DigitalSign is feasible if f and g are in $\textsc{Easy}(n)$.

Security

DigitalSign is secure if crack[DigitalSign]$^{\mathcal{I}}$ is in $\textsc{Hard}(n)$.

2.3.5 Complexity requirements of PseudoRandom

Size parameter

The size parameter for PseudoRandom is $n = |k|$.

Feasibility

PseudoRandom is feasible if f is in $\textsc{Easy}(n)$.

Security

PseudoRandom is secure if crack[PseudoRandom]$^{\mathcal{I}}$ is in $\textsc{Hard}(n)$.

It is less than perfectly intuitive that the size parameter for PseudoRandom be the length of the seeding key, but the key is the source for the randomness that PseudoRandom expands into the pseudorandom sequence it produces as output. The connection should be clearer upon consideration of the fact that a brute-force search through the keyspace would find all pseudorandom sequences and thus crack PseudoRandom.

Chapter 3

Quantum Computers

3.1 Introduction to quantum computation

In this section we will present a very basic overview of the concepts behind quantum computing. Anyone wishing for a broader and quite readable overview should seek out Reiffel and Polak's introduction to the topic [19], and for a thorough treatment the reader is directed to Nielsen and Chuang's text [17]. Additionally, Brassard gives a quick summary of the state of quantum attacks on cryptography [3], and Fortnow takes a look at quantum computation from the point of view of a complexity theorist [8].

3.1.1 Qubits and quantum properties

A quantum computer operates on *qubits*, or quantum bits. Conceptually, a qubit is simply the quantum analog of a classical bit. The strange rules of quantum mechanics, however, endow qubits with some interesting properties that have no counterparts in the classical world. Two properties in particular interest us. The first is that a qubit can exist not just in one state or another, but in a *superposition* of different states. When we measure a qubit that is in a superposition of states, we force the *collapse of the wave function*, and from that point onward the qubit will be in only one of the states, which we will see as the result of our measurement. Just which state the superposition collapses into depends on the amplitudes of the various states in the superposition. In order to convey this more clearly, we introduce a standard notation used to represent these concepts.

The state of a single qubit alone can be thought of as a unit vector in a two-dimensional vector space with basis $\{|0\rangle, |1\rangle\}$. Here $|0\rangle$ and $|1\rangle$ are orthogonal vectors representing quantum states such as spin up and spin down or vertical and horizontal polarization. A qubit can be in state $|0\rangle$ or in state $|1\rangle$, but it can also be in a superposition $x|0\rangle + y|1\rangle$ of the two states. The complex amplitudes x and y determine which state we will see if we make a measurement. When an observer measures a qubit in this superposition, the probability that the observer will see state $|0\rangle$ is $|x|^2$ and the probability of seeing $|1\rangle$ is $|y|^2$. Note that because $x|0\rangle + y|1\rangle$ is a unit vector, the sum $|x|^2 + |y|^2$ must be equal to 1.

The second quantum-mechanical property that interests us is *quantum entanglement*, which ties qubits inextricably to each other over the course of operations. Because qubits can be entangled and interfere with each other, the state of a multiple-qubit system cannot be represented generally as a linear combination of the state vectors of each qubit; the interactions between each pair of qubits is as relevant as the state of each qubit itself. The state of the system, then, cannot be described in terms of a simple Cartesian product of the individual spaces, but rather a tensor product. We will not go into the mathematics behind tensor products here, but one significant consequence of this fact is that the number of dimensions of the combined space is the product rather than the sum of the numbers of dimensions in each of the component spaces. For example, the Cartesian product of spaces with bases $\{x, y, z\}$ and $\{u, v\}$, respectively, has basis $\{u, v, x, y, z\}$ with $2 + 3 = 5$ elements. The tensor product of the spaces, however, has basis $\{u \otimes x, u \otimes y, u \otimes z, v \otimes x, v \otimes y, v \otimes z\}$ with $2 \cdot 3 = 6$ elements, where $u \otimes x$ denotes the tensor product of vectors u and x. We write the tensor product $|0\rangle \otimes |0\rangle$ as $|00\rangle$, so the vector space for a two-qubit system has basis $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ and the vector space for a three-qubit system has basis $\{|000\rangle, |001\rangle, \ldots, |111\rangle\}$, and so on.

One other property of quantum computers that is notable for its difference from classical computation is that all operations are reversible. On one level, this is due to the fact that classical computations dissipate heat, and with it information, whereas quantum operations dissipate no heat and therefore retain all information across each calculation. Since reversible quantum gates exist that permit the full complement of familiar logical operations, however, this point need not concern us.

3.1.2 The parallel potential of quantum computers

It is through entanglement and superposition that quantum computers offer a potentially exponential speedup over classical computers. The fact that entanglement implies a tensor product rather than Cartesian product means that a system of multiple qubits has a state space that grows exponentially in the number of qubits. Furthermore, because a qubit or system of qubits can be in a superposition of states, one operator applied to such a system can operate on all the states simultaneously. This gives quantum computers enormous computational power: an operator can be applied to a superposition of all possible inputs, performing an exponential number of operations simultaneously! The implications will be profound if a working quantum computer can indeed be built.

There is a catch, however: since the result will be a superposition of the possible outputs, a measurement of the result will not necessarily reveal the desired answer. In fact, a simple measurement will find any one of the possible outputs, taken randomly from the probability distribution of the wave amplitudes: in the naïve case, we are no better off than with a classical computer, since we can measure only one randomly chosen result. The key to designing quantum algorithms, then, is finding clever methods for manipulating probability amplitudes so that the desired answer has a high probability of being measured at the end of the computation. This is far from easy in general, though some clever techniques have been explored, such as using a quantum Fourier transform to amplify answers that are multiples of the period of a function (this is the technique Shor used in his factoring algorithm, discussed in Section 3.2).

3.1.3 Decoherence

The main problem thus far prohibiting actual realization of a quantum computer (unless, of course, the NSA or a similar organization has quietly build one without public knowledge!) is *decoherence*, or the interaction of the quantum system with the environment, disturbing the quantum state and leading to errors in the computation. We will not discuss this problem further in this paper, except to mention that techniques of quantum error correction have been used successfully to combat some effects of decoherence, but there is still a long way to go before building a large-scale quantum computer will be possible. For a detailed look at quantum error correction and other issues in quantum information, see part III of Nielsen and Chuang's text [17].

3.2 Shor's factoring algorithm

In 1994, Peter Shor discovered an algorithm to factor numbers in bounded-probability polynomial time on a quantum computer, along with another to compute discreet logarithms. The factoring algorithm uses a reduction of the factoring problem to the problem of finding the period of a function, and it uses the quantum Fourier transform in finding the period. Quantum parallelism makes it possible to work with superpositions of all possible inputs, which is the key to the increased power of this algorithm when compared to classical algorithms. We do not present the algorithm here, as excellent sources describing the algorithm already exist, and the curious reader is directed to one of those sources. Shor detailed this algorithm, along with one to find discrete logarithms, in his 1994 paper and its later, more complete version [24]. For a clear, less technical explanation of the algorithm, see Rieffel and Polak's introduction [19].

3.3 Consequences

Shor's algorithms have obvious and potentially catastrophic implications for the field of cryptography. Many cryptosystems, including the popular RSA cryptosystem, depend for their security on the assumption that factoring large numbers is difficult; others depend on the difficulty of computing discrete logarithms. The discovery of this polynomial-time quantum factoring algorithm means that anyone with a quantum computer could easily crack RSA and many other cryptosystems, and possibly much more. The full potential of quantum computers is unknown. Though we will not address them here, a few other quantum algorithms have been discovered, such as Grover's search algorithm [12], and there has been some work on quantum attacks on cryptographic systems, such as the 1998 paper by Brassard, Høyer, and Tapp on quantum cryptanalysis of hash functions [4].

In the next chapter we will look at the possible strengths of quantum computers and assess their implications for the cryptographic primitives we defined in Chapter 2.

Chapter 4

Cryptographic Implications of Quantum Computers

Quantum computers may have much more in store for cryptography than merely the demise of RSA; on the other hand, it may turn out that they have no more power than classical computers after all and it is just coincidence that the quantum polynomial-time factoring algorithm was discovered before the classical one. Here we introduce the most relevant complexity class for quantum computers and investigate how it might fit into classical hierarchies of complexity classes. The implications for cryptography are then explored.

4.1 Complexity of quantum computation

In his seminal 1985 paper [6], Deutsch proposed a model for a universal quantum computer with properties beyond those possessed by a classical Turing machine. Bernstein and Vazirani formalized the definition of an efficient quantum Turing machine, or QTM, in their 1997 paper [2], and went on to discuss the computational power of a QTM. They introduced the complexity class **BQP**, an analog to **BPP** on classical computers, to represent the class of problems efficiently solvable on a QTM: **BQP** is the set of languages accepted with probability at least $\frac{2}{3}$ by a polynomial-time QTM.

It is common in complexity theory for the exact relationships between complexity classes to be unknown. Because quantum computing is such a young study and quantum effects introduce so much strangeness, even less is known about **BQP** and its relationships to other complexity classes than is common. Here we examine some possibilities and their consequences.

4.1.1 Known relationships

Definition 9 *We use the notation $\subset$ to indicate proper containment, while $\subseteq$ means either containment or equality, as usual.*

In 1977, Gill proposed the class **BPP** (defined in Section 1.4.1) and showed that

$$\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{PSPACE},$$

and while $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$, it is not known whether either $\mathbf{BPP} \subseteq \mathbf{NP}$ or its converse is true [10]. Bernstein and Vazirani demonstrated that

$$\mathbf{BPP} \subseteq \mathbf{BQP} \subseteq \mathbf{PSPACE} \text{ [2]}.$$

According to Johnson's chapter in the *Handbook of Theoretical Computer Science* [15],

$$\mathbf{P} \subseteq \mathbf{UP} \subseteq \mathbf{NP}.$$

In their 1988 paper on public-key cryptosystems, Grollmann and Selman proved that one-way functions exist if and only if $\mathbf{P} \neq \mathbf{UP}$ [11] (see Section 4.2.3). Their definition of a one-way function does not match our definition of a one-way hash function exactly, yet the result is significant. The primary difference is that they require a one-way function to be one-to-one, though we do not include that requirement; it does not make sense for hash functions.

There have also been significant relativized results proved, most notably that there exists an oracle relative to which $\mathbf{P} = \mathbf{BPP} = \mathbf{BQP} \neq (\mathbf{UP} \cup \mathbf{coUP})$ [9]. Also, there is some evidence that **BQP** is not as large as **NP**, including that "relative to an oracle chosen uniformly at random with probability 1 the class **NP** cannot be solved on a quantum Turing machine (QTM) in time $o(2^{n/2})$" [1].

4.1.2 Possibilities

The reader should keep in mind that none of the inclusions just discussed are known to be proper. It is theoretically still possible that $\mathbf{P} = \mathbf{NP}$, or even $\mathbf{P} = \mathbf{PSPACE}$, though these equalities are widely believed to be false.

Where is **BQP**?	*Symmetric*	*OneWayHash*	*PublicKey*	*DigitalSign*	*PseudoRandom*	Section
$\mathbf{BPP} = \mathbf{BQP} \subset \mathbf{NP}$	$\surd$	$\surd$	$\approx$	$\approx$	$\surd$	4.2.1
$\mathbf{BPP} \subset \mathbf{BQP}$	$\surd$	$\surd$	$\approx$	$\approx$	$\surd$	4.2.2
$\mathbf{UP} \subseteq \mathbf{BQP}$	$\surd$	$\times$	$\times$	$\times$	$\surd$	4.2.3
$\mathbf{NP} \subseteq \mathbf{BQP}$	$\times$	$\times$	$\times$	$\times$	$\times$	4.2.4

Table 4.1: Summary of estimated implications. $\surd$ denotes survival of the primitive under that possibility (meaning that feasible, secure instances of the primitive may still exist), $\times$ means no survival, and $\approx$ indicates limited survival

One extreme possibility for placement of **BQP** and impact on cryptography is

$$\mathbf{BPP} = \mathbf{BQP} \subset \mathbf{NP} \text{ and } \mathbf{UP} \not\subseteq \mathbf{BQP}.$$

The other extreme end is

$$\mathbf{NP} \subset \mathbf{BQP}$$

We ignore classes above **NP**, in particular **PSPACE**, as being irrelevant to the present discussion. It should be obvious to the reader that the former possibility characterizes the possibility with the least impact on cryptography, while the latter promises the most impact. In the following section, we explore the various possibilities for placement of **BQP** in the complexity hierarchies and the implications for cryptography. A summary of the possibilities considered and the section in which each is discussed are compiled in Table 4.1.

4.2 Implications

4.2.1 $\mathbf{BPP} = \mathbf{BQP} \subset \mathbf{NP}$

The case where **BQP** = **BPP** and both are properly included in **NP** is simple: **BQP** introduces no new consequences for cryptography not present in **BPP**. Note that for this to be true, however, classical equivalents for Shor's algorithms would have to exist. This implies the consequences of the next section, though coming from **BPP** rather than **BQP**.

4.2.2 BPP $\subset$ BQP

In this scenario **BQP** properly contains **BPP**, so some problems are in **BQP** but not in **BPP**. The most likely candidates for problems in **BQP**$-$**BPP** are, of course, factoring and discrete logarithms. These problems form the basis for the security of many public-key and digital signature cryptosystems in use today. We have no reason to believe that no public-key or digital signature schemes are possible at all, but construction of a quantum computer would in any case herald the demise of at least the cryptosystems based on the presumed difficulty of factoring and finding discrete logarithms.

4.2.3 UP $\subseteq$ BQP

The possibility **UP** $\subseteq$ **BQP** may be very closely related to **UP** $\subseteq$ **P**. If **UP** $\subseteq$ **P**, one-way functions as defined by Grollmann and Selman [11] cannot exist.

Theorem 1 (adapted from Grollmann and Selman [11]) *The following are equivalent:*

1. ***P*** $\neq$ ***UP***
2. *There exists a one-way function.*

This theorem is based on Grollmann and Selman's definition of one-way functions, which includes that they are one-to-one, but the result suggests that this possibility would affect at least some functions that fall under our definitions. OneWayHash, PublicKey, and DigitalSign all depend on the existence of some sort of one-way functions, so they could all potentially be affected. It may be that **UP** $\subseteq$ **BQP** implies that no one-way functions can exist using quantum computation, which seems to indicate that OneWayHash, PublicKey, and DigitalSign would be compromised. Further work could prove or disprove this result.

4.2.4 NP $\subseteq$ BQP

This result is highly unlikely, but it would have profound implications. As long as we have an oracle to correctly identify a key, any of these primitives can be cracked with nondeterministic guessing (in the case of OneWayHash, we would guess and check messages rather than keys).

Theorem 2 *If* $\textbf{\textit{NP}} \subseteq \textbf{\textit{BQP}}$ *then crack[Symmetric]*$^{\mathcal{I}}$ *is in* $\textsc{Easy}(n)$*.*

Proof. For this proof, we do differentiate between **NP** and **FNP** and between **BQP** and **FBQP** (which is the obvious analogue to **FNP**). $\textsc{Easy}(n)$ is a class of functions, not decision problems, so we must show essentially that **NP** $\subseteq$ **BQP** implies that crack[Symmetric]$^{\mathcal{I}}$ $\in$ **FBQP**.

Define an arbitrary ordering over the keys $k_1, k_2, \ldots, k_{2^n} \in \mathcal{K}$, where n is the length of a key, which is also the size parameter.

Let M be an NTM that takes two integers m and n, such that $m < n$, as input and runs the following algorithm:

- guess a key k_j
- if $j < m$ or $n < j$
- then *reject*
- else if $\mathcal{I}(k_j) =$ **True**
- then *accept*
- else *reject*

Since M is an NTM, it will accept whenever there is a key k_j in the range $k_m, k_{m+1}, \ldots, k_n$ such that $\mathcal{I}(k_j) =$ **True**. Because $\mathcal{I}$ is an oracle, it takes constant time. To make this a real system (i.e. non-relativized), $\mathcal{I}$ could be replaced by any equivalent test in $\textsc{Easy}(n)$, as discussed in Section 2.2.1.

Now consider the following algorithm C:

Perform a binary search over the keyspace to find the correct key k to crack Symmetric. To do this, first call M with the arguments $m = 1$, $n = 2^{n-1}$. If M accepts, k is in the lower half so the search iterates with $m = 1$, $n = 2^{n-2}$. If M rejects, k is in the upper half so the search iterates with $m = 2^{n-1} + 1$, $n = 2^{n-1} + 2^{n-2}$. The binary search proceeds normally and will find k in $O(\log 2^n) = O(n)$ iterations. Return k.

C implements crack[Symmetric]$^{\mathcal{I}}$ since it returns the key k such that $\mathcal{I}(k) = \mathbf{True}$. The quantum computer will have made $O(n)$ calls to M, which is an NTM and so recognizes a language in **NP**, which is in **BQP** by assumption. C makes $O(n)$ calls to polynomial-time M, so it is obviously in $\textsc{Easy}(n)$. Therefore $\mathbf{NP} \subseteq \mathbf{BQP}$ implies that crack[Symmetric]$^{\mathcal{I}}$ is in $\textsc{Easy}(n)$. $\square$

Similar proofs can be constructed for the other primitives using very similar algorithms, since each primitive has one type of key or another or a message that can be nondeterministically guessed.

Chapter 5

Conclusions

5.1 Complexity-generalized cryptography

We have attempted to define five cryptographic primitives in such a way that we can discuss their complexity without mentioning specific complexity classes or cryptographic algorithms. This gives us the framework in which to explore the possible positions of **BQP** within complexity class hierarchies and apply the exploration directly to the cryptographic primitives.

5.2 Philosophical and practical implications

Should quantum computers ever become a reality, there is the potential for a large paradigm shift to take place in the field of cryptography. It is unknown how **BQP** is related to other classes such as **BPP**, **UP**, and **NP**. The results for cryptography depend on the various possibilities for relationships between classes here. At the very least, RSA and other popular cryptosystems will be compromised against any adversary with access to a quantum computer, though some cryptosystems would not be affected and perhaps suitable replacements could be found for the compromised schemes. At worst (or best, if one is the adversary!), all five primitives discussed would be affected and current algorithms implementing them compromised.

5.3 Open questions and future work

The topics explored and results obtained in this thesis could potentially be the starting points for research in a number of different directions.

- This thesis presents a summary of possibilities for the future in Chapter 4 but does not rigorously prove many bounds or results. This leaves an obvious gap to be filled in by future research: formally prove the remaining results from Table 4.1.

- Here we discuss cracking in absolute terms, but in reality it may be possible for an adversary to recover, say, half of all messages encrypted with a particular key for a particular instance of Symmetric. This would not fall under the definition of crack[Symmetric]$^{\mathcal{I}}$ that we present here, but it certainly would be a problem for the users of that cryptosystem! It remains to be seen how this framework of complexity-generalized cryptography can be applied to such incomplete crackings.

- We assume when discussing quantum computers, as has every other researcher we have encountered, that "quantum computers" are full-fledged full-sized computers capable of factoring very large numbers. But are there perhaps interesting quantum effects we can make use of to solve significant problems with, say, a 20-qubit quantum computer?

Acknowledgments

I would like to extend a hearty thanks to Tom O'Connell, who gave me very constructive comments especially on the theorems of Chapter 4. And I am greatly indebted to my advisor Sean W. Smith, who guided me through the whole process with encouragement and helpful criticism alike. He was much more than reasonably patient as I pushed back draft deadlines again and again, and he helped me to find a vision for this thesis and make it a reality.

Bibliography

[1] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. Strengths and weaknesses of quantum computing. *SIAM J. Comput.*, 26(5):1510–1523, October 1997. arXiv:quant-ph/9701001.

[2] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM J. Comput.*, 26(5):1411–1473, October 1997.

[3] Gilles Brassard. Quantum information processing: The good, the bad and the ugly. In Burton S. Kaliski, Jr., editor, *CRYPTO '97*, volume 1294, pages 337–341. Springer, 1997.

[4] Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum cryptanalysis of hash and claw-free functions. In Claudio L. Lucchesi and Arnaldo V. Moura, editors, *LATIN '98*, volume 1380, pages 163–169. Springer, 1998.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press and the McGraw-Hill Book Company, second edition, 2001.

[6] David Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. In *Proceedings of the Royal Society of London Ser. A*, volume A400, pages 97–117, 1985.

[7] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[8] Lance Fortnow. One complexity theorist's view of quantum computing. In David Wolfram, editor, *Electronic Notes in Theoretical Computer Science*, volume 31. Elsevier Science Publishers, 2000.

[9] Lance Fortnow and John Rogers. Complexity limitations on quantum computation. In *13th Annual IEEE Conference on Computational Complexity*, pages 202–209. IEEE Computer Society, 1998.

[10] John Gill. Computational complexity of probabilistic Turing machines. *SIAM J. Comput.*, 6(4):675–695, December 1977.

[11] Joachim Grollmann and Alan L. Selman. Complexity measures for public-key cryptosystems. *SIAM J. Comput.*, 17(2):309–335, April 1998.

[12] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *ACM Symposium on Theory of Computing*, pages 212–219, 1996.

[13] Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, May 1965.

[14] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, second edition, 2001.

[15] David S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 2, pages 68–161. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.

[16] Ralph C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, 1978.

[17] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 2000.

[18] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[19] Eleanor Rieffel and Wolfgang Polak. An introduction to quantum computing for non-physicists. arXiv:quant-ph/9809016, 1998.

[20] Ronald L. Rivest. Cryptography. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 13, pages 718–755. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.

[21] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[22] Bruce Schneier. *Applied Cryptography.* John Wiley & Sons, Inc., second edition, 1996.

[23] Claude E. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28:656–715, 1949. Originally in confidential report "A Mathematical Theory of Cryptography" dated Sept. 1 1945, which has since been declassified.

[24] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 124–134. IEEE Computer Society Press, 1994. Updated 1996 version at arXiv:quant-ph/9508027 with title 'Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer'.

[25] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.