Directed Study Artifact: Cryptography

Sophie Vulpe and Samuel Qin

November 2021

1 Introduction

This directed study ran in the 2021-2022 Fall Term with students Sophie Vulpe and Samuel Qin, and instructor Dan Proulx. We met every Monday and Wednesday C block.

Throughout the study, we used two sources as our primary texts. The first was *Lecture Notes on Cryptography* by Shafi Goldwasser and Mihir Bellare, lecture papers written for a course of the same name at MIT. Additionally, we referenced *An Introduction to Mathematical Cryptography* by Jeffery Hoffstein, Jill Pipher, and Joseph H. Silverman, a textbook provided by Mr. Proulx, to supplement our knowledge and dive more into the theoretical aspects of cryptography.

1.1 Struggles and Successes

Time was a concern for this directed study. While the original syllabus had three meetings planned each week, we were only able to meet twice a week due to a scheduling conflict. Due to special schedules, some weeks only held one meeting. Beyond scheduling issues, we found the original syllabus to be unusable, as it skipped critical background knowledge and did not run chronologically, so we reformatted it as we went. We present the topics we covered in roughly chronological order in this summary.
As a result of poor syllabus design, the group decided to follow the *Lecture Notes on Cryptography* and work through successive chapters. This strategy worked quite well, and allowed us to go on tangents (e.g) studying additional public key cryptography through *An Introduction to Mathematical Cryptography*.

However, there was an unexpected struggle in working through the *Lecture Notes on Cryptography*: poor notation. The entire group found some notation in the *Lecture Notes on Cryptography* impossible to comprehend. We also found their logic to be extremely confusing and convoluted.

Despite these setbacks, we were still able to cover the majority of the syllabus,

covering chapters 1-10 in *Lecture Notes on Cryptography* as well as chapter 3 in *An Introduction to Mathematical Cryptography.*

1.2 Goals of this Paper

In this paper, we summarize the knowledge we have learned, rewrite and reformat the information in *Lecture Notes on Cryptography*, detail solutions to problems, and share implementations (code) of cryptographic methods. We also hope that this paper may serve as an introduction for future students interested in studying cryptography.

2 Prerequisite Knowledge

This section will aim to summarize and list some of the basic knowledge required before studying cryptography.

2.1 Combinatorics/Probability

Combinatorics and Probability form the backbone of security and feasibility. This section will provide a brief overview of this background knowledge, in order to contextualize later results.

2.1.1 Combinatorics

Combinatorics is an area of math that deals with counting the number of possibilities/combinations. There are two main ideas that will be covered: combinations and permutations.

Before dealing with these, we need to define $n!$. We have

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1$$

for any integer n.

Combinations, written ${}^nC_k$ or $\binom{n}{k}$ (n choose k) counts the number of ways to choose k objects given n objects total, given that order does not matter. This is equal to $\frac{n!}{k!(n-k)!}$, so we can write

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Permutations are similar to combinations, except that order of selection does matter. Permutations, written ${}^nP_k$ are equal to $\frac{n!}{(n-k)!}$, so we have

$${}^nP_k = \frac{n!}{(n-k)!}.$$

This summarizes commonly used notation.

2.1.2 Probability

Probability can be written as

$$\frac{\text{number of successful combinations}}{\text{number of total combinations}},$$

where the number of successful combinations and the number of total combinations will be computed using the combinatorics detailed in section 2.1.1. Note that probability will always fall in the range of $[0, 1]$.

2.2 Background Number Theory

Certain number theoretic concepts are critical to the learning of cryptography. This section will list and briefly explain their importance.

2.2.1 Euclidean Algorithm

The Euclidean Algorithm states that if $d|a$ and $d|b$ for integers a, b, d, then $d|a - b$. By extension, we get that

$$ma + nb = d. \tag{1}$$

where $d = \gcd(a, b)$ and m, n are integers.

This is very useful if b is a prime. We replace b with p for ease of notation. We have $\gcd(a, p) = 1$, so $ma + np = 1$. Taking mod p, we get $ma \equiv 1 \bmod p$. Therefore,

$$m \equiv a^{-1} \bmod p. \tag{2}$$

This provides us an easy way to find multiplicative inverses in mod p.

2.2.2 Fermat Little Theorem/Euler Totient Function

Fermat Little Theorem states that

$$a^{p-1} \equiv 1 \bmod p.$$

Notably, this can be used as a primality test on a number n. Plugging some integer a, we have $a^{n-1} \bmod n$. If $a^{n-1} \equiv 1 \bmod n$, then n is a *probable* prime.

The Euler Totient function is an extension of the Fermat Little Theorem (sometimes called the Extended Fermat Little Theorem), which states that

$$a^{\phi(n)} \equiv 1 \bmod n.$$

$\phi(n)$ is defined as the number of integers less than n that are coprime to n. If $p_1^{k_1} \cdot p_2^{k_2} \dots p_n^{k_m}$ is the prime factorization of n, then

$$\phi(n) = \frac{p_1 - 1}{p_1} \cdot \frac{p_2 - 1}{p_2} \dots \frac{p_n - 1}{p_m}.$$

2.2.3 Chinese Remainder Theorem

The basis of the Chinese Remainder Theorem is finding a number that satisfies multiple modulo equivalences. We will consider the basic case with only two equivalences.

Assume that $\gcd(a, b) = 1$. We have

$$j \equiv a \bmod m, \tag{1}$$

$$k \equiv b \bmod n. \tag{2}$$

We seek to find an integer x that satisfies both of these relations.

From the Euclidean Algorithm, we know that

$$ma + nb = 1.$$

Then, we get

$$jma + jnb = j.$$

Taking mod a, we get

$$jnb \equiv j.$$

Letting $f = jn$, we get

$$fb \equiv j \bmod a. \tag{1}$$

Likewise, we have

$$ga \equiv k \bmod b, \tag{2}$$

where g is an integer.

We then add ga and fb to find x. We have

$$ga + fb = x. \tag{3}$$

Notice how ga impacts the value of x mod b without impacting the value of x mod a, and fb impacts the value of x mod a without impacting the value of x mod b. This is the key to the Chinese Remainder Theorem.

Note that the Chinese Remainder Theorem can be used on any number of modulo-remainder pairs (j, a and k, b above), provided that the modulos are co-prime. This demonstration only used two modulo-remainder pairs for the sake of simplicity.

2.2.4 Fast Power Algorithm

The fast power algorithm is a highly useful tool in coding, as it allows us to quickly and efficiently find

$$b^p \bmod m,$$

for some integer base b, power p, and mod m.

The fast power algorithm works by computing and storing the values of $b, b^2, b^4, \ldots, b^{2^k}$ mod m by successively squaring b^{2^k} and taking mod m.

Next, the algorithm finds the binary expansion of p, and then multiplies the stored values $b, b^2, \ldots, b^{2^k}$ and takes mod m to find the final result. This algorithm is extremely efficient, as unnecessary computations are minimized.

2.3 Complexity Theory

In its most basic form, Complexity Theory estimates the amount of time a computer would spend solving a problem. This section will provide some basic background information to contextualize later claims.

2.3.1 P Complexity

We define a problem to be **P** complexity if and only if there exists a Turing machine (some machine $M(x)$) and a polynomial $Q(y)$ so that on an input string x, we have

1. M must accept the input x
2. Machine M terminates after (at most) $Q(|x|)$ steps.

In other words, there must exist a machine which is able to solve the problem within a polynomial number of steps. We call this polynomial-time, or **P** complexity.

2.3.2 NP Complexity

We define a problem to be **NP** complexity if and only if there exists a Turing machine $M(x, y)$ and polynomial p so that on input x, we have

1. M accepts all inputs x, and generates a guess y. $M(x, y)$ terminates after (at most) $p(|x|)$ steps.
2. M accepts some inputs x and returns 1.
3. If x is not accepted, then the Turing machine returns 0.

This definition of **NP** essentially states that a non-deterministic machine M can compute the problem in polynomial time. However, this relies on a guess y given x, which makes the machine non-deterministic. From this, we can get that **NP** problems are not solvable in polynomial time by a deterministic Turing machine M. This second definition will be more useful to us.

2.3.3 Time Complexity

Time Complexity is a crucial tool in the study of cryptography, as it estimates the running time of algorithms. More precisely, it estimates the number of calculations needed, therefore estimating run time. Here, we introduce the $O(n)$ notation that is commonly used.

We start by defining $O(n)$ notation. Simply put, $O(n)$ notation estimates the number of operations a computer must do. For instance, consider the addition of two n-digit binary numbers. For each digit, the computer must perform one operation (adding the digits). Thus, we say addition is $O(n)$ time complexity, as n operations are needed for an n-digit input.

This definition allows a rough estimate of running time for any algorithm; we can say $O(n)$ (linear time), $O(n^2)$ (quadratic time), etc. In general, we say $O(n^a)$ is polynomial time, for any $a > 1$.

Some algorithms and their corresponding time complexities are given below:

- Addition/subtraction: $O(n)$
- Multiplication/division: $O(n^2)$
- Euclidean Algorithm: $O(n^2)$
- Modular exponentiation of n^k (by repeated multiplication and reduction): $O(n^2 \cdot 2^k)$

Note that these may not be optimized; in fact, optimization of functions to reduce their time complexity is a key aspect of cryptography.

2.4 Cryptographic Goals

It is important to define the goals of cryptography, as the further design needs to meet these goals. Additionally, it is important to define security mathematically, in order to compare different cryptographic systems.

2.4.1 Goals of Cryptography

In cryptography, we define the two parties who wish to communicate (usually named Alice and Bob), A and B. We also name an eavesdropper (Eve), E. The key goal of cryptography is allowing A to send B a message M without E being able to read the message. This is usually done so by encrypting M using a secret key S and sending a ciphertext c.

2.4.2 Adversary Model

With the basics defined, we can now model an adversary. We say that an adversary

- Has limited (non-infinite) computing power
- Has access to the ciphertext c
- Has access to the encryption and decryption algorithms (known as oracles)
- Is modeled as a polynomial-time algorithm

The first two are relatively easy, but models 3 and 4 need to be explained. We start with the latter:

When we assume the adversary has access to the ciphertext c, we also assume

they know and have access to the encryption and decryption algorithms. However, this does not allow them to decrypt the ciphertext: these encryption and decryption algorithms rely on the secret key S, which is not known.

Finally, we explore the 4th model. When we define an adversary as a polynomial-time algorithm, we model their algorithm to break the encryption scheme as polynomial-time. This is the key aspect of cryptography: if we can design the encryption scheme to be infeasible to break, we can prevent the adversary from learning the message m. This is discussed more in the next section.

2.4.3 Feasibility

Modern cryptography relies on the infeasibility of algorithms that can break an encryption scheme, while the encryption scheme is easy to compute. In general, we say an algorithm is infeasible if its running time is too long: usually, this is defined as requiring

- At least 2^{50} calculations (usually more), or
- The algorithm's $O(n)$ notation is of the form $O(b^n)$, for some $b > 1$

In short, we require any algorithm to break the encryption scheme take an unreasonable amount of time (e.g. 1000 years). This can be done by requiring a massive amount of computing power, or by making the problem an **NP** problem.

This brings us to the end of cryptographic modeling and prerequisite knowledge. Next, we will begin discussing some commonly used encryption schemes, as defined by Goldwasser and Bellare.

3 Topics Covered

This section will aim to rewrite, summarize, and reformat the topics in *Lecture Notes on Cryptography* for ease of readability. It should be noted that *Lecture Notes on Cryptography* may be referred to as *gb* or *gb.pdf* for ease of notation.

3.1 One Way Functions

One way functions form the basis of cryptography. Simply put, one way functions take an input (usually the plaintext m) and output a different value, c. These functions are designed so converting m to c is easy (polynomial time), but finding m given c is hard. In short, one way functions are hard to invert.

In cryptography, we define three variants of one way functions: strong one way functions, weak one way functions, and trapdoor functions. We start with the definition of strong one way functions, and build the definition of weak one way functions and trapdoor functions from this first definition.

3.1.1 Strong One Way Functions

As stated above and in *gb.pdf*, one way functions can be informally defined as functions which are "easy" to compute but "hard" to invert. More formally, we can say that these functions are computable in polynomial time, but any probabilistic polynomial time (PPT) algorithm will only succeed with negligible probability.

We now state and explain a formal definition of a strong one way function.

We say a probability is negligible if it vanishes faster than the inverse of any polynomial. More formally, we state that a function v is negligible if

$$v(k) < k^{-c}$$

for all k greater than a certain value k_c and for all constants $c > 0$. If this definition is graphed, we can imagine that $v(k)$ is bounded by the inverse of any polynomial and the x-axis. In short, $v(k)$ must decrease faster than the inverse of any polynomial, so we can also write

$$v(k) < \frac{1}{p(x)},$$

or

$$v(k) < \frac{1}{x^n}$$

for all n.

Inversely, we say a probability is non-negligible if

$$v(k) > \frac{1}{p(x)}$$

for some polynomial $p(x)$. Note, however, that a probability may be either negligible or non-negligible.

With negligible probability defined, we can now define a one way function.

We define a function f which operates from $\{0,1\} \rightarrow \{0,1\}$. We can say f is strong one way if:

1. There exists a polynomial time algorithm to output $f(x)$ given input x.
2. Assume that $f(x) = y$ for some input x on $\{0,1\}^k$. A polynomial time algorithm A given y as an input has a negligible probability, $v_A(k)$, of outputting a z so that $f(z) = y$, for a sufficiently large k.

   Goldwasser and Bellare state this as the following equation in section 2.2.1 of *gb.pdf*. Note that $\stackrel{\$}{\leftarrow}$ means random selection from a range.

$$\Pr[f(z) = y : x \stackrel{\$}{\leftarrow} \{0,1\}^k; y \leftarrow f(x); z \leftarrow A(1^k, y)] \leq v_A(k)$$

Note that the adversary is not asked to find x, but rather find a z so that $f(z) = f(x)$.

3.1.2 Weak One Way Functions

The previous definition defined the requirements of a strong one way function. However, with slight modification, we can define a weak one way function.

We define a function f which operates from $\{0,1\} \rightarrow \{0,1\}$. We can say f is weak one way if:

1. There exists a polynomial time algorithm that outputs $f(x)$ given input x
2. Assume that $f(x) = y$ for some input x on $\{0,1\}^k$. A polynomial time algorithm A given y as an input has a probability greater $\frac{1}{Q(k)}$ of outputting a z so that $f(z) \neq y$, for a polynomial function $Q(x)$ and a sufficiently large k.

   Goldwasser and Bellare state this in the form of the following equation:

$$\Pr[f(z) \neq y : x \stackrel{\$}{\leftarrow} \{0,1\}^k; y \leftarrow f(x); z \leftarrow A(1^k, y)] \geq \frac{1}{Q(k)}.$$

The difference between these two definitions is that weak one way functions only require some non-negligible fraction of the inputs to be hard to invert, while a strong one way function must be hard to input on all but a negligible fraction of inputs.

From these definitions, it is clear that strong one way functions are preferred. However, we have not yet shown the existence of such functions. However, Goldwasser and Bellare prove a theorem which states that "weak one way functions exist if and only if strong one way functions exist." Unfortunately, this theorem was beyond the scope of this directed study, but it can be found in section 2.2.2 of *gb.pdf*, titled Theorem 2.10.

3.1.3 Trapdoor Functions

These definitions bring us to the final variant of one way functions: trapdoor functions. Trapdoor functions are essentially a one way function with an extra property, which is a secret inverse function (a "trapdoor") which allows for the easy inversion of f. However, without knowledge of this function, f is still hard to invert. Goldwasser and Bellare proposed the following definition, which we have adapted and modified for ease of notation.

A trapdoor function is a one way function f which operates on $\{0,1\} \rightarrow \{0,1\}$. There must exist a polynomial p and polynomial time algorithm I so that for some t_k, $I(f(x), t_k) = y$ and $f(y) = f(x)$. In short, an inversion function I must exist which can provide a y which is the inverse of x on the function f.

These trapdoor functions are crucial in cryptography, as they form the basis of secure encryption and decryption. Some examples of trapdoor functions are the Euler Totient Function, RSA, and the discrete logarithm problem.

3.2 Pseudo-random Bit Generators

Pseudo-random bit generators are crucial to the area of cryptography, as they allow for the create of unbiased, uncorrelated random bits. These random bits are usually used for key generation.

3.2.1 Definitions

We need to start by defining pseudo-randomness. We start by defining probability distributions: Let X_n be a probability distribution from $\{0,1\}^n$. Then, we say X_n is polynomial-time indistinguishable from Y_n if

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

This definition was proposed by Goldwasser and Bellare in *gb.pdf*, in section 3.1. This simply means that the difference in probability of an algorithm $A(t)$ returning 1 when ran on X_n and Y_n must be bounded by an $\frac{1}{Q(n)}$, where $Q(n)$ is an increasing polynomial. Thus, this difference approaches 0.

From here, we can now define pseudo-randomness as being indistinguishable

from the uniform distribution. We write

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

Where U_n is the uniform distribution, and X_n is the probability distribution of a pseudo-random bit generator.

3.2.2 Generation and usage of Random Bits

Note that computers cannot generate pseudo-random bits on their own, as they would rely on an algorithm. Thus, the generation of random bits must come from a physical source. Some examples can be seen in *About Random Bits* (Geisler, Krøigård, Danielsen), which provides a method of obtaining random bits from air turbulence in hard drives. Another example would be Cloudflare's famous wall of lava lamps, whose unpredictable movement is captured by a camera and used for random bits. Other examples would be radioactive particles, coins/dice, and noise diodes.

However, we are not focused on the generation of random bits. Rather, we are interested in using these random bits. While physical sources are impossible to predict, they may be biased (in which a certain outcome is favored) or correlated. Fortunately, this can be remedied by suitable processing.

For instance, von Neumann proposed grouping the bits into pairs, and then treating 10 pairs as a 1, 01 pairs as 0, and discarding all other pairs. It can be shows that this equalized the probability of generating a 0 or 1.

Unfortunately, correlated pairs are more difficult to process. Blum, Santha and Varzirani, and Chor and Goldreich have published possible methods of correcting correlation. However, we were unable to cover this material.

Regardless, this provides a way to generate non-correlated, non-biased random bits given a hardware source, which are pseudo-random bits. Thus, we have created a true pseudo-random bit generator.

3.3 Public Key Encryption

A Public Key Encryption Scheme allows secure message exchange between sender(s) and receiver(s) without a secret key. There are multiple schemes, which will be covered.

3.3.1 Definitions

Before discussing Public Key Encryption, we need to define the encryption scheme. This paper will use a modified version of the definition provided in *gb.pdf* by Goldwasser and Bellare in Chapter 7.

Goldwasser and Bellare define Public Key Encryption as a triplet (G, E, D) of probabalistic algorithms. G is a key-generation algorithm, E is an encryption algorithm, and D is a decryption algorithm.

G is a key-generation algorithm that generates a random pair of keys (p, r), where p is the public key and r is a user-specific private key. In short,

$$G \to (p, r)$$

Note that p is published, while r remains secret.

E is a probabilistic encryption function that inputs a string m and outputs a ciphertext c using some algorithm with the public key p.. In other words,

$$E(m) \to c.$$

Notably, anyone can send encrypted messages, but only the receiver may decrypt them with the private key r.

Finally, D is a probabilistic decryption algorithm which inputs a string c and a private key r. D outputs the decrypted message, which should be m.

$$D(c, r) \to m.$$

Before finishing the definitions, note that G, E, D are all probabilistic. This is required for security, so that similar/identical messages have different ciphertexts.

3.3.2 RSA

One of the first public-key encryption schemes was RSA, which was proposed by ivest, Shamir, and Adleman in 1977. This scheme is built on the RSA function, which relies on the difficulty of factoring integers to be secure.

We start by defining a key-generation function. Let G be the key-generation function which randomly generates two (large) primes, p, q. It also generates an exponent e which is coprime to $(p-1), (q-1)$. Then, the public key $N = pq$ is generated. Finally, N, e are published as the public key. This can be written as

$$G \to N, e$$

which simply means that G generates (and publishes) N, e.

Next, we define an encryption algorithm E. Assume we have a plaintext m. Then, we find

$$m^e \equiv c \bmod N.$$

Here, c is our ciphertext. We can use the following notation for decryption:

$$E(m) \equiv m^e \bmod N \to c,$$

which simply means that E computes $m^e \bmod N$ and outputs c.

Finally, we define a decryption algorithm D. We first generate the private key d, which will be used for decryption. We find d so that

$$e \cdot d \equiv 1 \bmod (p-1)(q-1).$$

Note that this is possible because e was defined to be coprime to $(p-1), (q-1)$. Then, we take $c^d \bmod N$ and generate the plaintext m. We write

$$D(c) \equiv c^d \bmod N \to m.$$

Now that RSA is defined, we need to explore the underlying mathematics and logic to see why it works. In encryption and decryption, we take m^e to find c, and then we find c^d to find m without justification. Now, we wish to justify it and show that

$$(m^e)^d \equiv m \bmod N$$

is true. Notice that we defined e, d so that

$$e \cdot d \equiv 1 \bmod (p-1)(q-1).$$

Then, we get

$$m^{e \cdot d} \equiv m \bmod N.$$

From the Euler Totient Function, we know that

$$m^{(p-1)(q-1)} \equiv 1 \bmod N.$$

Because $e \cdot d \equiv 1 \bmod (p-1)(q-1)$, we know that

$$m^{e \cdot d} = m^{k(p-1)(q-1)+1},$$

where k is a positive integer. Taking mod N and applying Euler Totient function, we get

$$m^{e \cdot d} \equiv m^1 \equiv 1 \bmod N,$$

as desired.

Finally, we wish to show that RSA is feasible for use, but infeasible to break (secure). Notice that N is published, but p, q are not. It is infeasible for an adversary to factor N and find p, q, as integer factorization is a **NP** problem. On the other hand, it is easy to find N given p, q.

Next, notice that RSA relies on the fact that it is hard to find m only given $m^e \bmod N$, which is assumed to be an **NP** problem. On the other hand, it is easy to compute $m^e \bmod N$, using the fast power algorithm.

This finishes our brief overview of RSA. For more formal and in-depth learning, we recommend reading Chapter 7 in *gb.pdf* by Goldwasser and Bellare.

3.3.3 ElGamal

The ElGamal public key cryptosystem was developed shortly after RSA, and it relies on the discrete log problem for security. Additionally, the ElGamal public key cryptosystem is closely related to the Diffie-Hellman key exchange. In this section, we describe the algorithms of ElGamal, and briefly discuss its security and weaknesses.

We start by detailing the key generation of ElGamal. Similar to RSA, we begin by generating a large prime p. Then, we select a generator g, where $1 < g < p - 1$. This defines the initial public key generation. For brevity, we can write

$$G \to p, g.$$

Next, Alice randomly selects a private key A from the range $(1, p)$. She computes

$$A = g^a \bmod p.$$

A is then Alice's public key, and A is published but a is kept secret. The publishing of A completes the key generation step of ElGamal.

After key generation, Bob can encrypt his message m. He does so by choosing a random key k, which is a temporary, one-time use key. Then, using the public key A, Bob computes

$$c_1 = g^k \bmod p$$
$$c_2 = m \cdot A^k \bmod p.$$

Bob then sends (c_1, c_2) to Alice. In short, this encryption can be written as

$$E \to g^k \bmod p = c_1, m \cdot A^k \bmod p = c_2.$$

Finally, Alice decrypts (c_1, c_2) by computing

$$(c_1^a)^{-1} \cdot c_2 \bmod p.$$

This decryption is successful because

$$(c_1^a)^{-1} = (g^k)^{-a} \bmod p = g^{-a \cdot k} \bmod p,$$

so

$$(c_1^a)^{-1} \cdot c_2 = g^{-a \cdot k} \cdot m \cdot A^k,$$

and substituting $A = g^a$, we find that

$$(c_1^a)^{-1} \cdot c_2 = g^{-a \cdot k} \cdot m \cdot g^{a \cdot k} = m.$$

Thus, this decryption will always work, and this completes our definition of the ElGamal cryptosystem.

As mentioned before, ElGamal relies on the hardness of the Diffie-Hellman problem. Simply stated, the Diffie-Hellman problem is

$$\text{Given } g^x, g^y; \text{compute } g^{xy}.$$

This is assumed to be hard. However, the ElGamal cryptosystem has a backdoor weakness to a chosen-ciphertext attack, detailed as follows:

Eve has access to the values $c_1 = g^k$, c_2, and $A = g^a$. Also assume that Eve has access to an ElGamal oracle, which will perform the computations of ElGamal given proper inputs. With this oracle and information, Eve can solve the Diffie-Hellman problem:

Eve chooses an arbitrary c_2'. Then, she sends the oracle public key A and ciphertexts (B, c_2'). The oracle outputs

$$m' = (c_1^a)^{-1} \cdot c_2 = (g^{ab})^{-1} \cdot c_2.$$

From here, Eve computes

$$(m')^{-1} \cdot c_2 \equiv g^{ab} \bmod p.$$

Thus, Eve has solved the Diffie-Hellman problem with the aid of an ElGamal oracle, and can subsequently break the ElGamal encryption. This demonstrates that despite a "hard" underlying mathematical problem, backdoor methods can be found with poorly designed encryption.

This completes our brief discussion of public key encryption, as RSA and ElGamal have been covered and defined. Additionally, following Goldwasser and Bellare's definitions, we have defined a set of algorithms for any public key encryption scheme. Further reading can be found in chapter 7 of *gb.pdf*.

3.4 Block Ciphers

Block ciphers are essential tools for symmetric cryptographic schemes. On their own they don't do useful things for the end-users, but when used in conjunction with other tools they can be incredibly powerful.

A block cipher is a function $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$. This means that E takes two inputs, a k-bit string and an n-bit string, and maps them to another n-bit string. The first input is the key. Commonly, the second input is the plaintext (M) and the output is the ciphertext (E), so we write $E(K, M)$. The *key-length* k and *block-length* n are parameters associated with a specific block cipher. They need not be uniform across block ciphers, nor is the block cipher design itself.

Let's define a few other terms before we delve into how they all work together. For each key K, let $E_K = \{0,1\}^n \to \{0,1\}^n$ be a function defined

on $E_K(M) = E(K, M)$. By requirement, E_K is a permutation on $\{0,1\}^n$. Because permutations are invertible, E_K has an inverse ${E_K}^{-1}$, which also maps $\{0,1\}$ to $\{0,1\}$. Thus we have $E^{-1} : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ defined by ${E_K}^{-1}(C) = E(K, C)$, which is the inverse block cipher to E. These are the basic working pieces of a block cipher, but they say nothing about the strength of the cipher.

The block cipher E is some publicly and fully specified algorithm. Both E and E^{-1} should be easily computable, so given K, M we can readily compute $E(K, M)$, and given K, C we can readily compute $E^{-1}(K, C)$. (By "readily compute", we mean that there are enough publicly available and computationally feasible programs that can accomplish these tasks.) Typically, a random key K is chosen and kept secret between a pair of users. Then the function E_K is used by the users to process data before they send it to each other. We assume that the adversary can intercept some input-output examples of E_K, so they can see some plaintext and ciphertext messages. Thus, the adversary's goal is key recovery. The block cipher should be designed so that that task is computationally difficult, but this is not a sufficient condition for the security of a block cipher.

So far we've outlined the components of a block cipher, but we've said nothing about what properties it should have in its design. That topic is beyond the scope of this paper, but we give an example of a famously secure block cipher, DES.

3.4.1 DES

Unfortunately, due to time constraints, this section was not completed.

3.5 Hash Functions

At their core, hash functions are functions which compress an input, returning a shorter output. These functions have applications throughout cryptography. This section will delve into the details of SHA-1 and briefly dicuss Goldwasser and Bellare's definition(s) of hash function security.

3.5.1 SHA-1

We start our discussion of hash functions with SHA-1. Developed in 1995, SHA-1 remained secure for over two decades, until Google broke (definition discussed later) SHA-1 in 2017.

At its core, SHA-1 compresses strings up to length 2^64 into strings of length 160. It does so using a family of functions, which are collectively grouped under the name SHA-1. Goldwasser and Bellare provide a formal definition of the SHA-1 family, which has been copied below.

SHA-1(M) // $|M| < 2^{64}$
    $V \leftarrow$ SHF1(5A827999|6ED9EBA1|8F1BBCDC|CA62C1D6, M)
return V

SHF1(K, M) // where $|K| = 128$
    $y \leftarrow$ shapad(M)
    Parse y as $M_1|M_2|\ldots|M_n$ where $|M_i| = 512$
    $V \leftarrow$ 67452301|EFCDAB89|98BADCFE|10325476|C3D2E1F0
    for $i = 1, 2, \ldots n$, do
        $V \rightarrow$ shf1($K, M_i|V$)
return V

shapad(M); $|M| < 2^{64}$
    $d \leftarrow (447 - |M|) \bmod 512$
    Let l be the 64-bit binary representation of M
    $y \leftarrow M|1|0^d|l$ // $|y|$ is a multiple of 512
return y

shf1($K, B|V$) //$|K| = 128, |B| = 512, |V| = 160$
    Parse B as $W_0|W_1|\ldots|W_15$ where $|W_i| = 32$ and $(0 \leq i \leq 15)$.
    Parse V as $V_0|V_1|V_2|V_3|V_4$ where $|V_i| = 32$
    Parse K as $K_0|K_1|K_2|K_3$ where $|K_i| = 32$
    For

At its core, SHA1 simply inputs a message M, pads the message using shapad, and then iterates sha1, a compression function to get an output.

Notably, SHA1 can input any M with $|M| < 2^{64}$ and map it to a 160-bit string. By pigeonhole principle, we know that at least two of these messages must hash to the same string. In short, we know that

$$\text{SHA1}(M_1) = \text{SHA1}(M_2)$$

Yet, no such collision was found for over 20 years. This property is known as collision resistance, and is a key factor in hash function security.

3.5.2 Collision Resistance

Collision resistance is a key aspect of secure hash functions, and we will lay out some formal definitions in this section.

We start by defining a hash function as a family of functions $H : k \times D \rightarrow R$, where k is a key, D is the domain of H, and R is the range of H.

4 Code

Code was written as homework assignments throughout the course. The goal of this code was to reinforce understanding of underlying mathematical concepts.

This code was written in Python 3. Note that we use the random library, which **should not be used** for generating cryptographic keys, as mersenne twisters are not cryptographically secure.

The following code will be presented in **chronological order** of when the code was written.

4.1 RSA

This code for RSA is relatively simple and well-commented. See the following code:

```
import math

def buffer():
    print("--------------------------------------------------------------------")
p = 101 #this is private
q = 79 #this is private
e = 11 #then generate a d
d = 3191 #inverse of e mod totient function, it is assumed hard to find d given n, e. th

n = p*q #public key, this relies on n being hard to factor (p, q stay private)

message = int(input("enter an integer to be encrypted: "))
buffer()
plaintext = message % n

print("public key: ", n, e) #publish e and n
buffer()
def encrypt(plaintext):
    ciph = pow(plaintext, e, n) #take plaintext^e, then take mod n
    print("ciphertext: ", ciph)
    return int(ciph)

print("plaintext: ", plaintext)
ciph = encrypt(plaintext)

#maybe make a separate thing for decryption??

decrypt = input("decrypt? y/n: ")
buffer()
```

```
if (decrypt == "y"):
    decrypted = pow(ciph, d, n) #take ciph^d, then take mod n
    result = decrypted % n
    print("decrypted: ", decrypted)
else :
    pass
```

Note that p, q, e should be inputs, and d should be generated using the Euclidean Algorithm. However, this implementation relied on a pre-determined p, q, e, d. However, this could be fixed in a future implementation/edit.

4.2 ElGamal

The following code simulates both the sender (Alice) and receiver (Bob). Note that the random() function should not be used for cryptographic generation due to security issues. Additionally, note that this code does not generate a key, but uses a preset key (499).

```
import math
import random

p = 499 #p should be a large prime
g = random.randint(2, p-2)

print("Public P =", p)
print("Public G =", g)

secA = input("Secret A: ")
secA = int(secA)

pubA = pow(g, secA, p)
print("Public A =", pubA)

print("We now pretend to be Bob")

plaintext = input("Plaintext (insert int): ")
plaintext = int(plaintext)

tempKey = random.randint(1, p-1)
#print(tempKey)
cipher1 = pow(g, tempKey, p)
cipher2 = (plaintext * pow(pubA, tempKey)) % p
print(cipher1, cipher2)

print("Back to Alice")
cipher1A = pow(cipher1, p - 1 - secA, p)
#print(cipher1A)
```

```
decrypted = (cipher1A * cipher2) % p
print(decrypted)
```

This code could be improved by implementing a prime generation function and creating a more convenient input system. Otherwise, the code is already simplified and optimized.

4.3 Prime Generation

This code generates a random (probable) prime of size 10^n, given an input n. It searches for random numbers within the range of 10^n, and runs the Fermat Little primality test.

It should be noted that this randomness **should not** be used for cryptographic purposes, as the random library is insecure (mersenne twisters are insecure). Regardless, here is the code:

```
import math
import random

def primeGen(x):
attempts = 1
primeGenerated = False
while(primeGenerated == False):
gen = random.randint(10**(x-1), 10**x)
a = random.randint(1, gen)
if (pow(a, gen - 1, gen) == 1): #fastPowAlg
print("Trial passed")
print(gen)
primeGenerated = True
else:
print("Trial failed", attempts)
attempts = attempts + 1

primeSize = input("How big (pow 10) is prime?")
primeSize = int(primeSize)

primeGen(primeSize)
```

In testing, this code was able to generate primes of size 1000 within 2 minutes.

Note: in the line with the comment "fastPowAlg", the usage of the pow() function (which relies on the fast power algorithm) increases the efficiency thousandfold. This was an interesting demonstration of how important the fast power algorithm is.

4.4 Chinese Remainder Theorem

The following code on Chinese Remainder Theorem relies on creating lists of the modulos and remainders. The code does not take advantage of the Euclidean Algorithm, listing multiples of possible values instead.

```
import math

def CRT(x): #x is the number of modulo
    remainderList = []
    moduloList = []
    miniSolutions = []
    for i in range(x): #this for loop just takes inputs, this works perfectly
        remainder = input("remainder: ")
        remainder = int(remainder)
        remainderList.append(remainder)
        print("RemainderList", remainderList) #printBestDebug
        modulo = input("modulo: ")
        modulo = int(modulo)
        moduloList.append(modulo)
        print("ModuloList", moduloList) #printBestDebug

    for j in moduloList:
        tempM = moduloList.pop(0)
        tempbigMod = 1
        for m in moduloList:
            tempbigMod = tempbigMod*m
        print("tempbigMod", tempbigMod) #printBestDebug
        generated = False
        i = 0
        tempRemainder = remainderList.pop(0)
        while(generated == False):
            num = tempRemainder + i*tempM
            if(num % tempbigMod == 0):
                generated = True
                miniSolutions.append(num)
            else:
                i = i + 1
        moduloList.append(tempM) #add back the modulo that was popped out
    print("miniSolutions", miniSolutions)
    solution = 0
    for S in miniSolutions:
        solution = solution + S
    LCM = 1
    for n in moduloList:
        LCM = LCM*n
    print("Solution:", solution % LCM)
```

```
InPairs = input("How many modulo-remainder pairs? ")
InPairs = int(InPairs)
CRT(InPairs)
```

This code could potentially be improved, but its quality is severely limited by my poor understanding of Python.

4.5 Meet-in-the-middle Attack against the Discrete Log Problem

To briefly review, a meet-in-the-middle attack searches for a corresponding pair of input-outputs. The code relies on the following mathematical logic:

Let r be the remainder, b be the base, and p be a power. We are searching for a number $r \equiv b^p \mod m$, where r, b, m are given. We search for a pair $x \equiv r \cdot b^j \mod m$ and $x \equiv b^k \mod m$. Then, we have $r \cdot b^j \equiv b^k \mod m$, so we find $r \equiv b^{k-j} \mod m$.

Here is the code which follows this mathematical logic:

```
import math
import random

x = 0
y = 0

xPows = []
yPows = []
yList = []

base = 101
remainder = 18349
mod = 954002421469

iRange = 0
i = 0

while(True):
    powX = random.randint(0, mod)
    x = pow(base, powX, mod)
    xPows.append(powX)

    powY = random.randint(0, mod)
    y = remainder * pow(base, powY, mod) % mod

    yList.append(y)
```

```
yPows.append(powY)

iRange = iRange + 1
print(iRange)

for i in range(0, iRange):
    if(x == yList[i]):
        print(powX, yPows[i])
        power = powX - yPows[i]
        print(power)
        print(pow(base, power, mod))
        quit()
```

This code stores values of $r \cdot b^k$ and b^k (k is randomly chosen) in two lists. It then checks the lists to search for a match, and then outputs the power k that satisfies $r \equiv b^k \bmod m$.

5 Closing Remarks