ElGamal Encryption and Diffie–Hellman Key Exchange

Sending and receiving information over the internet often comes with the assumption of sending and receiving over a private and secure channel. We will discuss two cryptographic algorithms which help to guarantee privacy and security for sending and receiving parties using number theoretic concepts.

Traditionally, sending and receiving parties are labeled alphabetically “Alice” and “Bob” to represent “sender” or “receiver” parties, with “Eve” sometimes being introduced to represent a potential “eavesdropping” party that reads public information sent between Alice and Bob [16]. Public key algorithms are a distinct category of cryptographic algorithm where all “keys” used (for encryption or decryption) need not be hidden from a potential “eavesdropper.” Obtaining the public keys will, assuming the algorithm is correctly implemented, not compromise the security of the information sent between the two sender and receiver parties.

The mathematical reason for why public key algorithms work hinges on the concept of “one-way” functions (also called “trapdoor” functions), which are functions that are assumed to be easy to compute given a set of input, but difficult to invert [3]. These are called “one-way” functions since the function’s outputs are thought to be simpler to calculate when given a set of inputs, whereas solving the “other way”—finding the inputs given the function’s outputs—is thought to be more difficult to compute. With this idea, the public keys of such algorithms are then inefficient tools for an eavesdropping party to use (as they would be using these keys to solve the one-way function in the more computationally inefficient and “difficult” direction) while useful and “easy” for intended parties.

Prior to the inception of public key cryptographic algorithms, there were instead private key algorithms (also known as *symmetric* algorithms, since decryption and encryption relied on the same private keys [15]), and it was recommended that sender and receiver parties send their keys physically, although this introduced a number of security concerns and was known as the *key-distribution problem* [15]. Finding a more secure system was an open problem at the time solved by public key encryption, also known as *asymmetric* cryptographic algorithms, where there exist both public and private keys. Today, applications of such protocols being used for encryption and decryption are most common over computer networks and on the internet where sender and receiver parties send and receive information over network connections.

In 1976, the Diffie–Hellman key exchange algorithm (commonly abbreviated “DHKE”) was the first *public key* cryptographic algorithm, attributed to computer scientist Ralph Merkle, who began working on the algorithm as an undergraduate at UC Berkley in 1974, alongside cryptographers Whitfield Diffie and Martin Hellman [14]. (The idea behind the DHKE algorithm is illustrated in Figure 1.) The DHKE algorithm later inspired cryptographer Taher Elgamal, who created a cryptographic algorithm for encryption and decryption in 1985 which he based on the DHKE’s public key system as well as the algorithm’s exploitation of the difficulty of computing *discrete logarithm* problems, which we will describe in Section 1 [4]. Elgamal’s algorithm is called the ElGamal algorithm, a public key algorithm which we will soon mathematically illustrate and describe alongside the DHKE algorithm. An important distinction to make between the DHKE algorithm and the ElGamal algorithm is that the DHKE algorithm guarantees the creation of a shared secret key integer value for sending and receiving parties whereas the ElGamal algorithm guarantees the encryption and decryption of an integer value [11].

Although also a public key algorithm, the Rivest–Shamir–Adleman (or RSA algorithm), which has today enjoyed relatively greater use in internet applications [15], exploits the difficulty of *prime factoring* (to find two specific large prime numbers p, q whose product is n, a public RSA key) while the difficulty of calculating *discrete logarithms* is used to create the one-way function for both the ElGamal algorithm and Diffie-Hellman key exchange algorithm. Additionally, the DSA algorithm (formally the "digital signature algorithm") and a class of public key cryptographic algorithms known as "elliptic curve cryptosystems" exploit the believed difficulty of solving discrete logarithms (to ensure the security of information transfer), as it has been conjectured (but not proved—it has not yet been proven that one-way functions exist) that solving discrete logarithm problems require nondeterministic polynomial time (and is thus difficult to efficiently solve, including with computers) [11, 12].

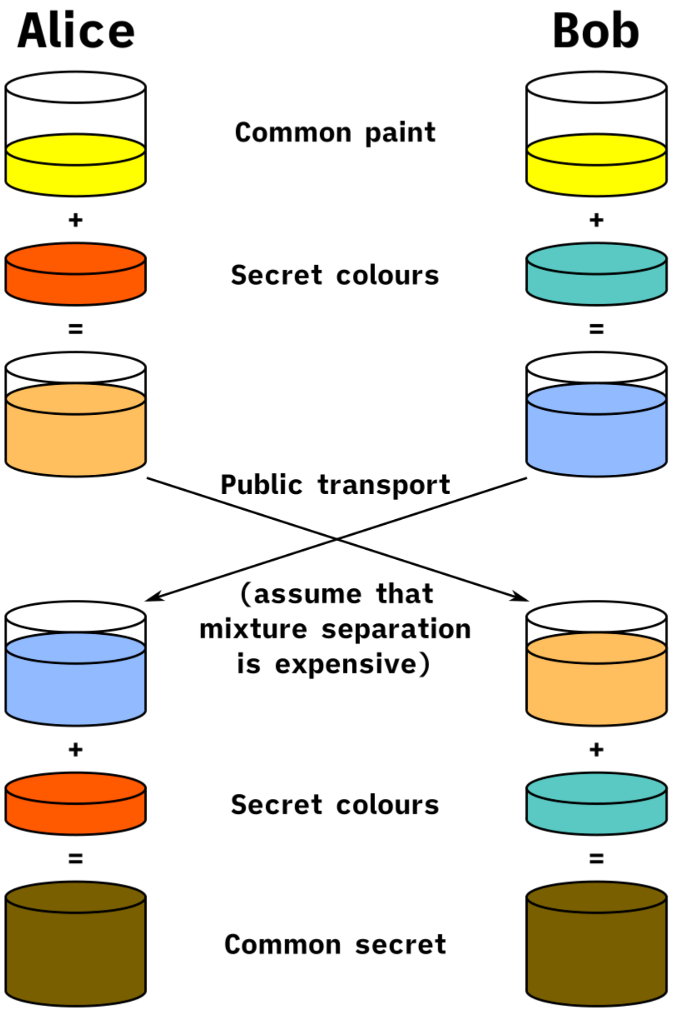


Figure 1. DHKE algorithm analogy using mixed paint [19].

Here, two parties “Alice” and “Bob” both use their secret key to send a publicly transported key (here shown as paint colors) to the opposite party. After combining their private and publicly exchanged keys, they swap their key combinations, finally using their respective private keys to solve for the “common secret.” This is a shared value between the two parties.
For a more detailed explanation of the DHKE algorithm, see Section 3.

Section 1. Primitive Roots and Discrete Logarithms

For the ElGamal public key cryptographic algorithm, the proof of correctness—that the receiver can find value *m* using the algorithm—can be proven using Fermat's Little Theorem [10], which we will see in Section 2. As for the difficulty of finding *m* given the public values exchanged by the sender and receiver, this requires solving a discrete logarithm problem.

Before describing a *discrete* logarithm problem, it is useful to begin by introducing a *continuous* logarithm problem for comparison. Importantly, continuous logarithm problems are not difficult to solve computationally because efficient algorithms exist to compute them. Given three integers *a*, *b*, *x*, solving a discrete logarithm problem means solving for *x* in the equation $a^x = b$, which is also written $\log_a b$.

Similarly, solving a *discrete* logarithm problem means solving for integer *x* in the equivalence relation $r^x \equiv a \pmod{p}$, which is also written $\text{ind}_r\, a \pmod{p}$, where $r, a \in \mathbb{Z}$. Notably, *p* is a prime number, *r* is a primitive root modulo *p*, and the $\gcd(a, p) = 1$ [21]. As we will soon see, this is because when *r* is a primitive root and *p* is a prime number, then there is guaranteed to be a solution to the discrete logarithm problem. However, solving for *x* is computationally inefficient for large values of *p*, which makes for a useful one-way function for cryptographic algorithms, as will be discussed in Sections 2 and 3.

The formal definition of a primitive root *a* (mod n) is that for $a \in \mathbb{Z}_n^*$, we say *a* is a generator (or primitive root) of $\mathbb{Z}_n^*$ if for all $c \in \mathbb{Z}_n^*$, then $c \equiv a^k \pmod{n}$ for some $k \in \mathbb{Z}$. [13]. For $\mathbb{Z}_p^*$, which is a set of congruence classes (mod p) where *p* is a prime number, then recall that $\mathbb{Z}_p^* = \{[1]_p, \dots [p-1]_p\}$. It has also been proved for all prime numbers *p*, $\mathbb{Z}_p^*$ has exactly $\phi(p-1)$ primitive roots [21], so it is instructive to try and find primitive roots (mod p) using small values of *p*.

An illustration of two examples of attempting to solve for primitive roots modulo a prime number *p* through a "brute force" approach is provided for primitive roots (mod 3) and for (mod 7), in Figures 2 and 3 respectively, using Mathematica code (alongside commented explanations).

```
(* Note #1: Primitive roots of 3 can generate the entire set
Zp*, where we have p=3 *)
(* Note #2: The first column shows the number we are checking
to see whether it is a primitive root (mod 3) *)
     l = Table[If[j==1,i,Mod[i^(j-1), 3]], {i,3},{j, 4}]
```

```
(* Note #3: Zp* is cyclic for prime p, since the set is {1, ...
,p-1}; note each value is a congruence class (mod p). We want a
generator that can generate this set. *)
(* We check each value from 1 to 3. The exponents we check go
from 0 to 3, so we subtract 1 from j,since Mathematica indexes
from 1... *)

     Grid[l,Frame -> All]
     1 1 1 1
     2 2 1 2
     3 0 0 0
(* We see 2 generates the set {2, 1, 2}; note each value is a
congruence class (mod p). For p=3, that is Zp*, so we have
found the correct set. *)
(* We also see that 1 and 3 do not generate Zp*, so these are
not primitive roots (mod 3). *)
```

Figure 2. Finding primitive roots modulo 3 through a “brute force” approach.

```
(* Note #1: Primitive roots of 7 can generate the entire set
Zp*, where we have p=7  *)
(* Note #2: The first column shows the number we are checking
to see whether it is a primitive root (mod 7) *)
     m = Table[If[j==1,i,Mod[i^(j-1), 7]], {i,7},{j, 8}]
(* Note #3: Zp* is cyclic for prime p, since the set is {1, ...
,p-1}; note each value is a congruence class (mod p). We want a
generator that can generate this set. *)
(* We check each value from 1 to 7. The exponents we check go
from 0 to 7, so we subtract 1 from j, since Mathematica indexes
from 1... *)
     Grid[m,Frame -> All]
```

1	1	1	1	1	1	1	1
2	2	4	1	2	4	1	2
3	3	2	6	4	5	1	3
4	4	2	1	4	2	1	4
5	5	4	6	2	3	1	5
6	6	1	6	1	6	1	6
7	0	0	0	0	0	0	0

```
(* We see 3,5 generates the set Zp*; note each value is a
congruence class (mod p). For p=7, that is Zp*, so we have
found the correct set. *)
(* We also see that 1,2,4,6, and 7 do not generate Zp*, so
these are not primitive roots (mod 7). *)
```

Figure 3. Finding primitive roots modulo 7 through a "brute force" approach.

Recall that it has been conjectured (although not proved) that discrete logarithms cannot be solved in polynomial time (making them computationally inefficient and time-consuming to solve), and no time-efficient algorithm has yet been found. So, although mathematically possible to solve, the difficulty of finding a solution this way makes the public key aspect of the algorithm secure so long as this problem remains difficult to compute. Analogously, the RSA algorithm would require an eavesdropping party to find primes p, q from public key n, where $n = pq$. The difficulty of prime factoring large values of n is the one-way function of the RSA algorithm just as the difficulty of solving discrete logarithms is the one-way function of the ElGamal algorithm [21].

Section 2. The ElGamal Algorithm

In order to use the ElGamal algorithm, the sender's original message is an integer m, and the receiver will decrypt this value to find m. Both the sender and receiver have one private value, integer values a and b, respectively, which are known only to themselves (it is assumed).

Exploiting the difficulty of solving discrete logarithms, ElGamal's algorithm finds a large prime p using an algorithm such as a prime sieve [17]. (This is often done using hundreds or thousands of digits to ensure that the sender's original message, m, is less than p so that the algorithm works properly. The larger the prime selected, the more time-consuming it is assumed that the discrete logarithm will be to compute, especially if trial and error methods are used.)

Next, ideally a random primitive root of p is used, but true random is difficult to generate using algorithms, so a "pseudo-random" number generator will be used. (These number generators take an input seed and output values which do not have an obvious pattern [22].) Then, a pseudo-random primitive root of p is found, which we denote g. Then, pseudo-random integer a, $1 \leq a < p - 1$ is chosen, and we find integer e, where $e \equiv g^a \pmod{p}$. Here, since p, g, e are public values, the value of a—which is private—could be found, but this would require solving for a in $g^{\mathbf{a}} \equiv e \pmod{p}$, also denoted $\text{ind}_g\, e \pmod{p}$, which would require solving a discrete logarithm problem, as mentioned previously.

Using $c_1, c_2, b \in \mathbb{Z}$, the "sender" will let b be a pseudo-random integer, $1 \leq b < p - 1$. Then, c_1 and c_2 will each form an equivalence relation with public values g, e respectively by letting $c_1 \equiv g^b \pmod{p}$ and $c_2 \equiv m * e^b \pmod{p}$. Similar to finding the value of a using $e \equiv g^a$ (mod p) above, the value of b could also be found (inefficiently) by solving the discrete logarithm problem of solving for b in $g^{\mathbf{b}} \equiv c_1 \pmod{p}$. Also note that c_1, c_2 will be public values, and b will be private. The equivalence relation $g^{\mathbf{b}} \equiv c_1 \pmod{p}$ can similarly be written $\text{ind}_g\, c_1 \pmod{p}$.

Since $e \equiv g^a \pmod{p}$, then it immediately follows that

$c_2 \equiv m * e^b \pmod{p} \equiv m * (g^a)^b \equiv mg^{ab} \pmod{p}$.

Recall that, by Fermat's Little Theorem, for $a, p \in \mathbb{Z}$, if p does not divide a, then it follows:

$a^{(p-1)} \equiv 1 \pmod{p}$.

Now, since the "receiver" has access to the value a, since $c_1 \equiv g^b \pmod{p}$, then to obtain $g^{ab} \pmod{p}$, we compute $c_1^a \pmod{p}$. We find that

$(c_1)^a \equiv (g^b)^a \equiv g^{ab} \pmod{p}$.

Now, we have $c_1^a \equiv g^{ab} \pmod{p}$ and $c_2 \equiv mg^{ab} \pmod{p}$. Here, the goal is to apply Fermat's Little Theorem using the values $g^{ab} \pmod{p}$ and $mg^{ab} \pmod{p}$ to find $m \pmod{p}$, where m (mod p) is the plaintext that must be decrypted. To begin, we must ensure that the condition "for $a, p \in \mathbb{Z}$, if p does not divide a" is satisfied. Importantly, in ElGamal's algorithm, we picked m so $m < p$ (by selecting a large prime; if this assumption is incorrect, then the proof of correctness fails) and $g < p$ since g is a pseudo-random primitive root of p, and all primitive roots of p are less than p.

So, we can apply Fermat's Little Theorem using the values c_1^a, c_2, and $p - 2$ to find

$(c_1^a)^{(p-2)} \equiv (g^{ab})^{(p-2)} \pmod{p}$, and since $c_2 \equiv m(g^{ab}) \pmod{p}$, then

$(c_2)(c_1^a)^{(p-2)} \equiv m(g^{ab})(g^{ab})^{(p-2)} \equiv m(g^{ab})^{(1)+(p-2)} \equiv m(g^{ab})^{(p-1)} \pmod{p}$.

Recall that Fermat's Little Theorem states:

$a^{(p-1)} \equiv 1 \pmod{p}$.

So, we substitute the value $(g^{ab})^{(p-1)}$ in (mod p) to find:

$m(g^{ab})^{(p-1)} \equiv m(1) \equiv m \pmod{p}$.

Therefore, a "sender" can have a "receiver" calculate m using the public key ElGamal algorithm. Assuming that solving discrete logarithm problems are too computationally difficult for computers to *efficiently* solve (and that "one-way functions" exist, as is conjectured), then it will be difficult to find *m* using only public values, assuming a large prime *p* was initially selected.

The ElGamal algorithm is further illustrated in Figure 4.

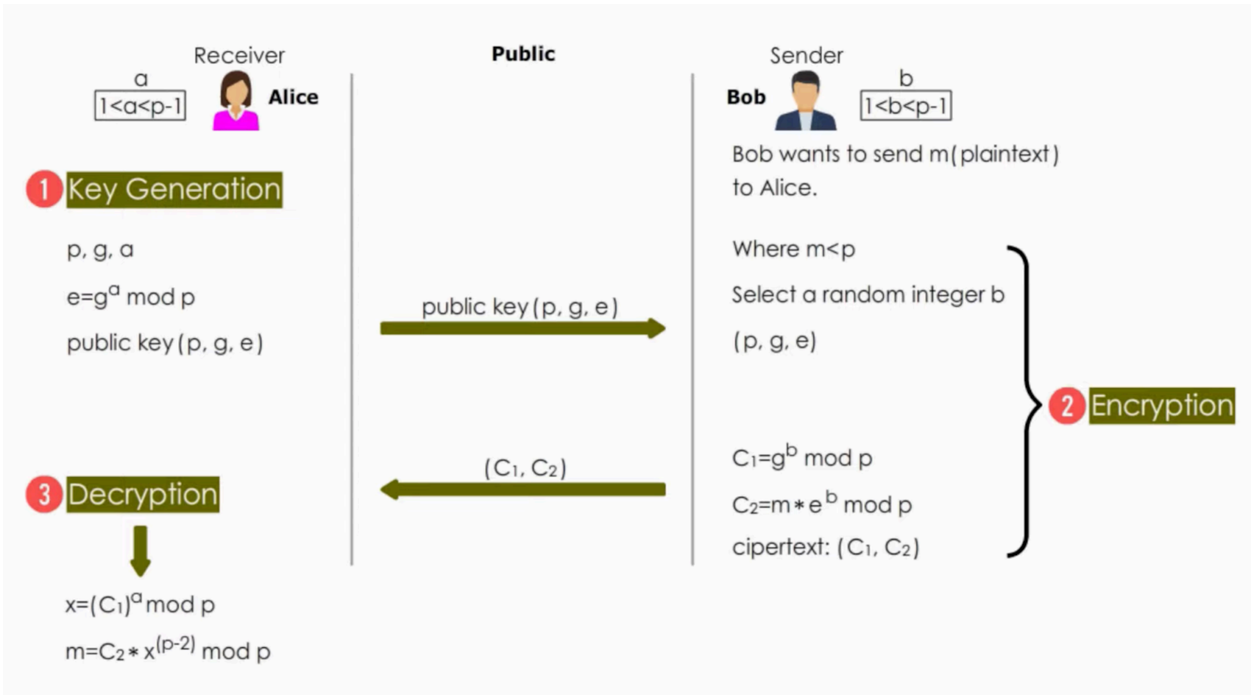


Figure 4. ElGamal cryptographic algorithm [18].

Here, two parties "Alice" and "Bob" have both a secret and private key. After combining their private and secret keys, they swap their key combinations in order to find the message sent by the sender to the receiver, decrypted by the receiver.

Section 3. The Diffie-Hellman Key Exchange Algorithm

We will now discuss the DHKE algorithm and its proof of correctness. While this algorithm is simpler than the ElGamal algorithm, it is still interesting to consider, as it was the first published public key algorithm, inspiring the creation of future public key algorithms. While the DHKE algorithm also uses discrete logarithms for its one-way function, unlike the ElGamal algorithm, the DHKE algorithm was developed for secure *key exchange* as opposed to encryption and decryption. So, rather than having a plaintext *m* integer to be encrypted and decrypted, the DHKE algorithm seeks to have both sender and receiver parties create a shared integer value congruent modulo p.

As with the ElGamal algorithm, there will be a large prime p selected followed by a pseudo-random primitive root of p, which we again denote g. These values are “public.” One party, which we will call the first party, will then generate a pseudo-random integer value a, $1 \leq a < p$, and the other party, which we will call the second party, will generate a pseudo-random integer value b, $1 \leq b < p$.

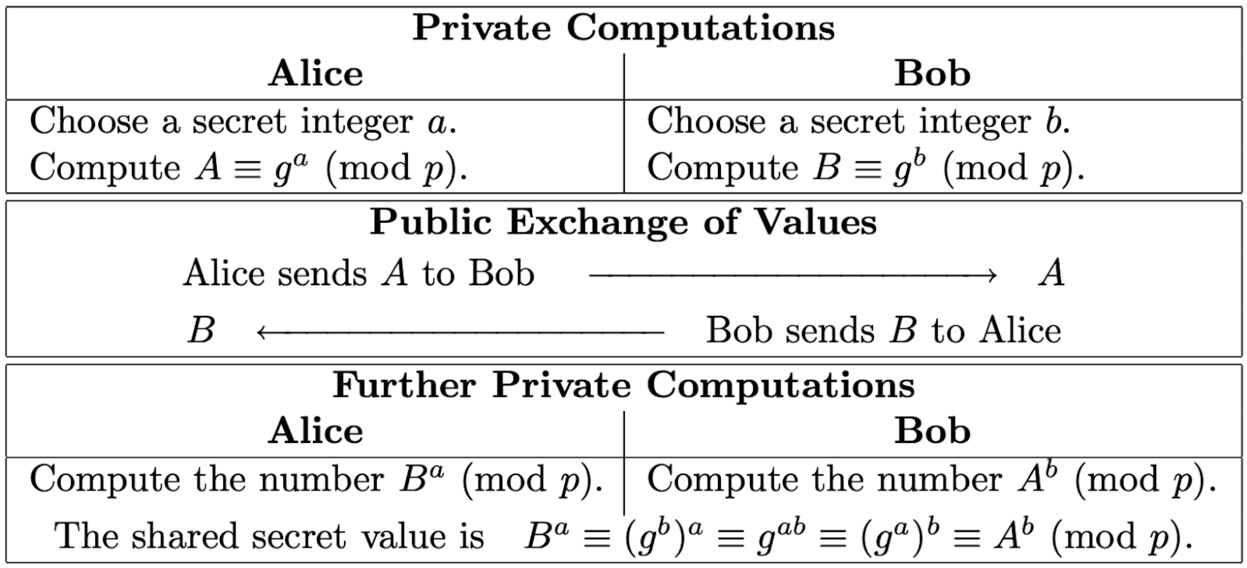
Now, with access to integer a, the first party can compute integer e, where $e \equiv g^a$ (mod p), and the second party can compute integer c_1, where $c_1 \equiv g^b$ (mod p) with access to b. Since e is public, the second party computes e^b (mod p) and the first party computes c_1^a (mod p).

We can show that $e^b \equiv c_1^a$ (mod p) with the following.

$e^b \equiv (g^a)^b \equiv g^{ab}$ (mod p), and $c_1^a \equiv (g^b)^a \equiv (g^a)^b \equiv g^{ab}$ (mod p), so we have shown:

$g^{ab} \equiv g^{ab}$ (mod p) is equivalent to $e^b \equiv c_1^a$ (mod p).

The DHKE algorithm is further illustrated in Figure 5.



Private Computations

Alice	Bob
Choose a secret integer a.	Choose a secret integer b.
Compute $A \equiv g^a \pmod{p}$.	Compute $B \equiv g^b \pmod{p}$.

Public Exchange of Values

Alice sends A to Bob $\xrightarrow{\hspace{6em}}$ A

B $\xleftarrow{\hspace{6em}}$ Bob sends B to Alice

Further Private Computations

Alice	Bob
Compute the number $B^a \pmod{p}$.	Compute the number $A^b \pmod{p}$.

The shared secret value is $B^a \equiv (g^b)^a \equiv g^{ab} \equiv (g^a)^b \equiv A^b \pmod{p}$.

Figure 5. Diffie–Hellman key exchange algorithm [17].
Using a private keys a and b, both parties Alice and Bob find from their respective private keys the value B^a (mod p) and A^b (mod p), both of which are congruent to one another. This occurs after a public key exchange between the two parties, and this public key is obtained using each party’s respective private keys.

Bibliography

[1] *The elgamal cryptosystem - ramanujan.math.trinity.edu*. (n.d.). Retrieved December 7, 2022, from http://ramanujan.math.trinity.edu/rdaileda/teach/f20/m3341/lectures/lecture23_slides.pdf

[2] Pauli, S. (n.d.). *MAT 112 integers and modern applications for the uninitiated*. ElGamal Encryption System. Retrieved December 7, 2022, from https://mathstats.uncg.edu/sites/pauli/112/HTML/secelgamal.html

[3] Libretexts. (2021, July 18). *5.6: The elgamal cryptosystem*. Mathematics LibreTexts. Retrieved December 7, 2022, from https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics/Yet_Another_Introductory_Number_Theory_Textbook_-_Cryptology_Emphasis_(Poritz)/05%3A_Indices__Discrete_Logarithms/5.06%3A_The_ElGamal_Cryptosystem

[4] *Elgamal:public-key cryptosystem - Indiana State University*. (n.d.). Retrieved December 7, 2022, from https://cs.indstate.edu/~jgrewal/steps.pdf

[5] *Practical private key cryptography - wwwmayr.informatik.tu-muenchen.de*. (n.d.). Retrieved December 7, 2022, from https://wwwmayr.informatik.tu-muenchen.de/konferenzen/Jass05/courses/1/presentations/gurevich_pres.pdf

[6] *ElGamal cryptosystem - Arizona State University*. (n.d.). Retrieved December 7, 2022, from https://math.asu.edu/sites/default/files/elgamal.pdf

[7] *El-Gamal - University of Illinois Chicago*. (n.d.). Retrieved December 7, 2022, from https://homepages.math.uic.edu/~leon/mcs425-s08/handouts/el-gamal.pdf

[8] *ElGamal Cryptosystem*. Math.umd.edu. (n.d.). Retrieved December 7, 2022, from https://www.math.umd.edu/~immortal/MATH406/

[9] Menezes, A. J., C., V. O. P., & Vanstone, S. A. (2001). *Handbook of Applied Cryptography*. CRC Press.
[10] *Lecture 10 number theoretic algorithms 1 El-Gamal Encryption*. University of Manchester. (n.d.). Retrieved December 7, 2022, from http://syllabus.cs.manchester.ac.uk/ugt/2019/COMP26120/2020slides/num1.pdf

[11] *Elgamal cryptosystem - University of Texas at Dallas*. (n.d.). Retrieved December 7, 2022, from https://personal.utdallas.edu/~muratk/courses/crypto07_files/elgamal.pdf

[12] *Open problem garden*. Discrete Logarithm Problem | Open Problem Garden. (n.d.). Retrieved December 7, 2022, from http://garden.irmacs.sfu.ca/op/discrete_logarithm_problem

[13] Anna NadenAnna Naden, & John OmielanJohn Omielan. (1967, October 1). *If r is a primitive root, then the residue of rt is also a primitive root if gcd(t,ϕ(m))=1 where ϕ is Euler's totient*. Mathematics Stack Exchange. Retrieved December 7, 2022, from https://math.stackexchange.com/a/3807701

[14]
NIHF inductee Ralph Merkle invented the public key cryptosystem. NIHF Inductee Ralph Merkle Invented the Public Key Cryptosystem. (n.d.). Retrieved December 7, 2022, from https://www.invent.org/inductees/ralph-merkle

[15] *Computer science | wellesley college*. The key-distribution problemA public-key solution. (n.d.). Retrieved December 7, 2022, from https://cs.wellesley.edu/~cs310/lectures/23_public_key_slides_handouts.pdf

[16] Bennett, C. H., Brassard, G., Robert, J.-M., Bennett and, C. H., Bennett and, C. H., Bennett, C. H., Berlekamp, E. R., Brassard, G., Carter and, J. L., Chor, B., Diffie and, W., Gallager, R. G., Goldreich, O., Helgert and, H. J., Lint, J. H. van, MacWilliams and, F. J., McEliece, R. J., Noble and, B., Ozarow and, L. H., … Bierbrauer, J. (n.d.). *Privacy amplification by public discussion*. SIAM Journal on Computing. Retrieved December 7, 2022, from https://epubs.siam.org/doi/10.1137/0217014

[17] *2.3. Diffie–Hellman key exchange*. Brown. (n.d.). Retrieved December 7, 2022, from https://www.math.brown.edu/johsilve/MathCrypto/SampleSections.pdf

[18] sunnylearning. (2022, September 18). *Intro to the elgamal cryptosystem*. YouTube. Retrieved December 7, 2022, from https://www.youtube.com/watch?v=oQqr8d5s3Uk

[19] File:Diffie-Hellman Key Exchange.svg. (2021, February 4). Wikimedia Commons, the free media repository. Retrieved 19:20, November 18, 2022 from https://commons.wikimedia.org/w/index.php?title=File:Diffie-Hellman_Key_Exchange.svg&oldid=530363048.

[20] Libretexts. (2021, July 7). *5.2: Primitive roots for primes*. Mathematics LibreTexts. Retrieved December 7, 2022, from https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics/Elementary_Number_Theory_(Raji)/05%3A_Primitive_Roots_and_Quadratic_Residues/5.02%3A_Primitive_Roots_for_Primes#:~:text=Every%20prime%20number%20has%20a,k%20for%20some%20integer%20k.

[21] Shoup, V. (2009). Finding generators and discrete logarithms. In *A computational introduction to number theory and Algebra*. essay, Cambridge University Press.

[22] https://www.sciencedirect.com/topics/mathematics/pseudo-random-number#:~:text=A%20pseudorandom%20number%20generator%20is,indistinguishable%20from%20a%20random%20string.