Pixel Mask 4.0; ElGamal AME with password protection

Bachelor Semester Project S5, University of Luxembourg

KIM TEREBA, 0232478543 University of Luxembourg, Luxembourg

This project presents a fully functional ElGamal-based anamorphic encryption scheme that allows two messages to be embedded in a single ciphertext. The system was developed by completely reworking the previous implementation from scratch, resulting in a cleaner, more efficient, and more robust design. Building on the ElGamal public-key system, which relies on the discrete logarithm problem in a subgroup of large prime order, the new implementation uses symmetric state parameters and a shared double key to generate pseudorandom values. This allows a hidden message to be embedded while preserving the structure of standard ElGamal ciphertexts. During decryption, the same pseudorandom value is regenerated from the shared state, so the remaining value contains the secret message as a discrete logarithm. The Baby-Step Giant-Step algorithm is then used to recover this message without large lookup tables, making the scheme faster, more memory-efficient, and capable of supporting larger secret messages than the previous version. Pixel Mask 4.0 also introduces password protection and an animated avatar, Echo, to combine improved cryptography with better usability.

ACM Reference Format:
Kim Tereba. 2026. Pixel Mask 4.0; ElGamal AME with password protection: Bachelor Semester Project S5, University of Luxembourg. 1, 1 (January 2026), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Secure communication often assumes that both sender and receiver can freely exchange messages without interference. In practice, this is not always the case: messages can be intercepted, altered, or observed by third parties such as service providers, network administrators, or more malicious parties. Standard encryption schemes protect the contents of the message but often do not hide the presence of the communication itself. Anamorphic encryption addresses this limitation by allowing one single ciphertext to encode two messages. Depending on which decryption key is used, a different message can be revealed. This allows for plausible deniability, since any intercepted key only reveals an innocent cover message, while the secret message remains hidden. ElGamal encryption is perfectly suitable for this, because its random-looking outputs allow a hidden message to be embedded without altering the appearance of the ciphertext. Therefore, by combining pseudorandom generation with careful key handling, multiple messages can be hidden in a single ciphertext. This paper presents a new ElGamal-based anamorphic encryption scheme implemented in Pixel Mask 4.0. The updated system is faster and more robust than earlier versions, making anamorphic cryptography easy to use for secure communication through Pixel Mask.

Author's Contact Information: Kim Tereba, kim.tereba.001@student.uni.lu0232478543, University of Luxembourg, Esch-sur-Alzette, Luxembourg.

2 Concepts

Discrete-logarithm problem is the problem of finding x if you know g and h such that $g^x = h$ [1]. It is very difficult to solve for large numbers because there is no efficient algorithm to solve x directly, compared to multiplication or exponentiation. This makes it useful for cryptosystems like ElGamal.
Safe prime is a number p such that (p-1)/2 is also prime [2]. For example, 23 is a safe prime because: (23-1)/2 = 11, which is also prime. They are used in cryptography because they make certain attacks on discrete-log-based systems much harder.
Brute forcing is a simple but inefficient method of solving a problem by trying all possible solutions until the correct one is found [3]. For example, guessing a password by trying every combination.
Group of order q is a mathematical group that contains exactly q elements, and the "order" tells you the size of the group.
Generator is a number that produces every element in a subgroup when you repeatedly exponentiate it. For example, if g is a generator of a group of size q, then $g^1, g^2, g^3, \dots, g^q (\mod p)$ produces all elements of the subgroup [4].
Cyclic group A group where all elements can be generated by repeatedly using the generator g, the powers of g produce every element in the group[5].

3 ElGamal

3.1 What Is ElGamal and Why Is It Best-Suited for Anamorphic Encryption?

ElGamal is a public-key encryption scheme that relies on the discrete logarithm problem. It works in a group whose order is a large prime number, which is often done using a safe prime p where $p = 2q + 1$ and both p and q are prime. A generator g is chosen for the subgroup of size q, then a user creates a secret key sk and a corresponding public key $pk = g^{sk} \mod p$. To encrypt messages (type: integer), a random value r is selected and the ciphertext is created: ct = (ct_0, ct_1) and $ct_0 = m * pk^r \mod p, ct_1 = g^r \mod p$. [6] ElGamal is well suited for anamorphic encryption because the r value looks completely random and that randomness makes it possible to hide a second message in the ciphertext without changing its structure. Therefore, the returned ciphertext looks normal, even when it contains two messages instead of one.

3.2 How Does the New ElGamal Implementation Work?

The updated ElGamal scheme follows the same concept of encrypting and decrypting two messages. Starting with a safe prime p and its subgroup of order q, generator g, and public/secret keys pk, sk. However, now the random value r is created differently and another secret key sk_a, also called the double key, is generated. Instead of brute-forcing it, the code generates a pseudorandom number r' in a controlled manner using AES in CBC mode. This pseudorandom generator takes as input the double key sk_a, an initialization vector

IV, and a state value st, and produces a number between 1 and q. The anamorphic message (type: integer) m_a is then added to this value: $r = r' + m^a \mod q$. Then standard ElGamal is performed using this r, the ciphertext becomes: $ct_0 = m * pk^r modp$, and $ct_1 = g^r modp$. During decryption, the same pseudorandom value r' is recreated using the shared double key sk_a. $g^{m_a} = g^r / g^{r'} \mod p$. Brute-forcing m_a works if the integer is small enough, but since we want to encrypt slightly larger integers m_a is recovered with the Baby-Step Giant-Step algorithm. Anamorphic communication example:
Bob – generates:

- pk, sk, sk_a

Bob – sends to Alice:

- pk, sk_a (via secure channel), IV (public), state st (public)

Alice – stores:

- pk, sk_a, IV, st

For better safety, it is recommended to share the double key sk_a via asymmetric encryption.

```
# Pseudo-random function - generate r'
def F(sk_a, IV, st, q):
    prg_input = st.to_bytes(16, 'little') #
        State encoded as AES block
    iv_bytes = IV.to_bytes(16, 'little') # IV
         for CBC mode

    aes = AES.new(sk_a, AES.MODE_CBC, iv=
        iv_bytes)
    encrypted = aes.encrypt(prg_input)

    # Map output into valid exponent range
    r_prime = (int.from_bytes(encrypted, '
        little') % (q - 1)) + 1
    return r_prime

# Generate generator g (mod p)
def g_generator(p):
    while True:
        h = secrets.randbelow(p-3) + 2
        g = pow(h, 2, p)
        if g != 1:
            return g

# Generate public, secret, and double key
def create_keys(p, q, g):
    sk_a = get_random_bytes(16) # Double key
    sk = secrets.randbelow(q) # Secret key
    pk = pow(g, sk, p) # Public key
    return pk, sk, sk_a

# Anamorphic encryption - Encrypt cover +
    anamorphic messages
def aEncrypt(p, q, g, m, m_a, pk, sk_a, IV,
    st):
    r_prime = F(sk_a, IV, st, q)
    r = (r_prime + m_a) % q # Embed secret
        message in r

    # Standard ElGamal encryption
    pk_r = pow(pk, r, p)
    ct0 = (m * pk_r) % p
    ct1 = pow(g, r, p)
    ct = (ct0, ct1)

    # Update state for next message
    st += 1
    return ct, st

# Anamorphic decryption - Recover anamorphic
    message
def aDecrypt(p, g, sk_a, ct, IV, st, q, bound
    ):
    r_prime = F(sk_a, IV, st, q)

    # Remove pseudorandom r' to get g^{m_a}
    g_rprime = pow(g, r_prime, p)
    g_rprime_inv = pow(g_rprime, p - 2, p)
    g_m_a = (ct[1] * g_rprime_inv) % p

    # Solve discrete log to recover m_a
    m_a = baby_step_giant_step(p, g, g_m_a,
        bound)
    return m_a


# Standard decryption - Recover cover message
def Decrypt(p, ct, sk):
    s = pow(ct[1], sk, p) # s = g^r^x = y^r
    s_inverse = pow(s, -1, p)
    cover_msg = (ct[0] * s_inverse) % p
    return cover_msg
```

3.3 Comparison to Older Version

The previous ElGamal anamorphic encryption scheme needed to pick random pairs of numbers x and y and store a large lookup table of powers of g. During encryption, the code repeatedly tried random pairs until they satisfied a certain condition. The decryption then had to check many possible x values against the table. Although the approach worked, it was slow, used a lot of memory, and limited the length of the secret message. The new scheme improves on it by removing random searching and large tables. It uses AES to generate a pseudorandom number r' and the secret anamorphic message is just added to it, making the encryption more deterministic and efficient. For decryption, Baby-Step Giant-Step replaces the lookup table, to recover m_a quickly without storing many values. This new version is faster, uses less memory, and can handle larger messages.

4 Baby-Step Giant-Step

4.1 Explanation

Baby-Step Giant-Step (BSGS) is an algorithm for solving the discrete-logarithm problem, that is, it can find x, given g and h such that $g^x = h$.
Now we assume that any exponent x can be rewritten as $x = i * n + j$ [7]. So instead of trying every possible x, BSGS speeds up the process by splitting the search into two parts:

- Baby-steps (j): Precompute and store many small powers of $g : g^0, g^1, g^2, \ldots, g^{n-1}$.
- Giant-steps (i): Repeatedly multiply h by g^{-n}, checking each result against the baby-step table. Each giant-step corresponds to which interval of size n the exponent is in.

So when a giant-step matches a baby-step, the algorithm knows both parts and can reconstruct $x = i * n + j$. This reduces the search time from O(q) to O($\sqrt{q}$).

Simple intuition example:
Consider x to be a two-digit number. The tens digit is the giant-step and the ones digit is the baby-step. Now BSGS finds x's tens digit by giant-stepping and its ones digit by baby-stepping. Then combine both into the actual number.

Second example:
If baby-steps give you "The last part of the exponent is 57" and giant-steps give you "It is in block 3". This means that the exponent is between $3n$ and $4n$. Therefore, $x = 3n + 57$

4.2 How BSGS Is Used in the Code?

BSGS is used during decryption to recover the secret message m_a. After recomputing the value of r', it gets removed from the exponent: $g^{m_a} = g^r * (g^{r'}) - 1 \mod p$. Now, the exponent of g is the secret message, which is computed by using a BSGS function and a known range (bound). The baby-step loop builds a table of small powers of g, and the giant-step loop iterates through decreasing powers of g^n, while checking for a match. Once a match is found, the exponent x (m_a in our case) is reconstructed to reveal the secret message. BSGS allows the scheme to recover the secret message quickly, without storing large tables and larger ranges for the message.

```
# Baby-Step Giant-Step algorithm for discrete
     logarithms
def baby_step_giant_step(p, g, h, bound):
    if h == 1:
        return 0  # g^0 == 1

    # Limit bound to at least 1
    if bound <= 1:
        return None

    # Block size n = sqrt(bound)
    n = math.isqrt(bound)
    if n * n < bound:
        n += 1

    # Baby-steps: store g^j for j in [0, n)
    baby = {}
    current = 1
    for j in range(n):
        if current not in baby:
            baby[current] = j
        current = (current * g) % p

    # Precompute inverse of g^n
    g_n = pow(g, n, p)
    g_n_inv = pow(g_n, p - 2, p)

    # Giant-steps: look for i, j s.t. x = i *
         n + j solves g^x = h
    step = h
    max_i = (bound + n - 1) // n  # Compute
        how many giant steps we need

    for i in range(max_i):
        # Check if current giant step matches
             any baby table value
        if step in baby:
            j = baby[step]
            x = i * n + j
            if x < bound:
                return x
            else:
                return None
        # Move to next giant step
        step = (step * g_n_inv) % p

    return None
```

5 AME Cryptography Initialization and Testing

Before testing the new ElGamal scheme on standard and anamorphic messages, several steps are needed. First, to initialize the parameters, a safe prime p is set up by using a well known 2048-bit safe prime defined in RFC 3526 (Group 14) [8]. This prime and its associated $q = (p - 1)/2$ ensure that the subgroup used for ElGamal has a large prime order to make the discrete-logarithm problem difficult, and then the generator g is chosen for this subgroup. Now, the system has a strong mathematical foundation for testing encryption.

- p = 179769313486231590770839156793787453197860296048756011706444423684197180216158519368947833795864925541502180565485980503646440548199239100050792877003355816639229553136239076508735759914822574862575007425302077447712589550957937778424442426617334727629299387668709205606050270810842907692932019128194467627007
- q = 89884656743115795385419578396893726598930148024378005853222211842098590108079259684473916897932462770751090282742990251823220274099619550025396438501677908319614776568119538254367879957411287431287503712651038723856294775478968889212221213308667363814649693834354602803025135405421453846466009564097233813503

The first test encrypts and decrypts simple integers: a decoy message m and an anamorphic message ma. The encryption function takes these parameters and returns the ciphertext (ct_0, ct_1) together with an updated state value st. Decryption is then performed with two functions: the standard Decrypt function recovers only the decoy message m using the secret key sk, while aDecrypt uses the double key sk_a and BSGS algorithm to recover the secret message ma. The size of m_a is limited by a bound parameter that defines the maximum value that BSGS will search for; if this bound becomes too large, the decryption time grows exponentially because BSGS searches up to $\sqrt{(bound)}$ steps. All of this confirms that anamorphic encryption and decryption work correctly and helps determine the optimal bound for integer sizes, before testing on messages.
The final step of testing applies the entire system to text messages. Each word in the input string is converted into an integer using a character-to-integer mapping function before being encrypted. For every word pair(m, ma) one ciphertext is created, allowing the messages to be processed word by word. Although only words of length < 7 work due to the optimal chosen $bound = 2^{36}$. During decoding, depending on which key is provided (secret key sk, double key sk_a) it decrypts the decoy text or recovers the secret text. After converting the integers back into words using an integer-to-character mapping function, both the decoy and the secret message are restored. This ensures that the system implements the safe primes, ElGamal, and BSGS to perform anamorphic encryption on messages.

6 Pixel Mask 4.0

6.1 Password Protection

Pixel Mask introduces password protection to improve security. In the first run of the application, the user is prompted to create a new password which is stored on their computer disk for long-term use. In all subsequent launches, the application opens a small login window, where the user has to enter this password to access the app.

```
# Check if entered password matches stored
    password
def verify_password(password: str):
    stored = load_password_data()
    if stored is None:
        return False # No password set

    salt = stored["salt"]
    stored_hash = stored["hash"]

    # Hash input password using stored salt
    new_hash = hash_password(password, salt)

    # Compare hashes
    return new_hash == stored_hash


# Display login window to verify password (or
     create password on first run)
def run_login_screen():
    stored_pw = load_password_data()
    first_run = stored_pw is None # No
        password saved yet

    login = tk.Tk()
    login.title("Create␣Password" if
        first_run else "Pixel␣Mask␣Login")
    login.geometry("300x180+600+300")
    login.resizable(False, False)

    # Prompt text
    label_text = "Create␣a␣password:" if
        first_run else "Enter␣password:"
    tk.Label(login, text=label_text).pack(
        pady=10)

    # Password entry
    pw_entry = tk.Entry(login, show="*",
        width=25)
    pw_entry.pack()
    pw_entry.focus()

    # Confirm password only on first run
    confirm_entry = None
    if first_run:
        tk.Label(login, text="Confirm␣
            password:").pack(pady=5)
        confirm_entry = tk.Entry(login, show=
            "*", width=25)
        confirm_entry.pack()

    # Error message label
    error_label = tk.Label(login, text="", fg
        ="red")
    error_label.pack(pady=5)

    result = {"success": False}

    # Handle login or password creation
    def submit(event=None):
        pw = pw_entry.get()

        if not pw:
            error_label.config(text="Password
                ␣cannot␣be␣empty")
            return

        if first_run:
            # Create and store password
            if pw != confirm_entry.get():
                error_label.config(text="
                    Passwords␣do␣not␣match")
                return

            set_password(pw)
            result["success"] = True
            login.destroy()
        else:
            # Verify existing password
            if verify_password(pw):
                result["success"] = True
                login.destroy()
            else:
                error_label.config(text="
                    Incorrect␣password")

    # Bind Enter key and button
    pw_entry.bind("<Return>", submit)
    if confirm_entry:
        confirm_entry.bind("<Return>", submit
            )

    tk.Button(
        login,
        text="Create␣Password" if first_run
            else "Login",
        command=submit
    ).pack(pady=10)

    login.mainloop()
    return result["success"]
```

6.2 Echo

Pixel Mask now also includes an avatar called Echo, who uses different facial expressions to make the application more engaging. Echo is not fully interactive yet, it simply changes expressions at random time intervals to appear animated. The avatar also adapts its appearance to match the current mode, including dark, light, red, blue, and ame, with the ame mode having a slightly different design from the default. The spritesheets for each mode are shown below.

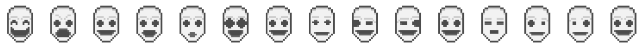

Fig. 1. Light Mode

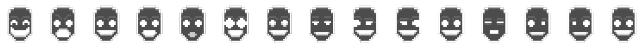

Fig. 2. Dark Mode

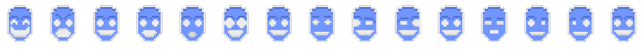

Fig. 3. Blue Mode

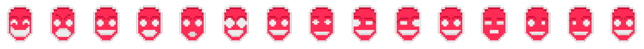

Fig. 4. Red Mode

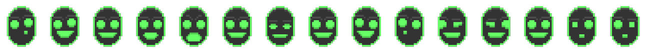

Fig. 5. AME Mode

```
def load_avatar_frames(self):
    # Determine avatar sprites based on
        current GUI mode
    mode = None
    match self.gui_mode:
        case 1: mode = "assets/
            light_mode_sprites"
        case 2: mode = "assets/
            dark_mode_sprites"
        case 3: mode = "assets/
            blue_mode_sprites"
        case 4: mode = "assets/
            red_mode_sprites"
        case 5: mode = "assets/
            ame_sprites"
        case _: mode = None

    # Clear previously loaded animation
        frames
    self.animation_frames.clear()

    try:
        # Iterate through all files in
            sprite folder
        for file in sorted(os.listdir(
            mode)):
            if file.lower().endswith(".
                png"): # Only process PNG
                img = Image.open(os.path.
                    join(mode, file))
                # Convert to Tkinter-
                    compatible image and
                    store it
                self.animation_frames.
                    append(ImageTk.
                    PhotoImage(img))

    except Exception as e:
        print(f"Failed␣loading␣animation␣
            frames:␣{e}")


def start_avatar_animation(self):
    # Update avatar frame if frames are
        loaded
    if hasattr(self, "avatar_label") and
        self.animation_frames:
            self.current_frame = (self.
                current_frame + 1) % len(self
                .animation_frames)
            frame = self.animation_frames[
                self.current_frame]
            self.avatar_label.config(image=
                frame)
            self.avatar_label.image = frame #
                 Avoid garbage collection

        # Random delay between frames
        delay = random.randint(100, 1200)
        self.after(delay, self.
            start_avatar_animation)
```

7 Assessment and Potential Improvements

The implemented anamorphic ElGamal scheme is fully functional and can be performed on multiple messages. One limitation, however, is the restricted character length per word in the secret message. The character-to-integer mapping function supports the most important characters, including uppercase and lowercase letters, numbers, and spaces. Though, only a small set of punctuation characters is currently supported, meaning that other symbols may not be encrypted correctly. At the moment, Pixel Mask includes only one anamorphic encryption method based on ElGamal. The original goal of the project was to implement an anamorphic scheme based on Paillier encryption. Although this was not completed, it could be added in the future. Password protection improves the security of the application, but if a user forgets their password, there is currently no way to reset it. This could be improved by adding a type of multi-factor authentication to allow users to verify their identity and reset their password. Another planned feature was a panic password that would delete all sensitive data (keys, states, ciphertexts, messages) before the application opens, providing extra protection if the user is forced to open Pixel Mask. Moreover, to support this, future versions would need to add basic file management features to the interface. Echo, Pixel Mask's avatar, adds simple animations to make the interface more engaging, but so far runs only in a loop and is not interactive. A future version could connect Echo to an Ollama-based text generator to answer cryptography-related questions and help guide users, to remove the need for hardcoded help commands. The animations could then be aligned with user actions and chat output to really make the interface interactive.

8 Conclusion

Pixel Mask 4.0 improves on previous versions by implementing a redesigned and stable ElGamal-based anamorphic encryption scheme. The new version is more efficient, reliable, and capable of handling larger hidden messages while staying consistent with standard ElGamal ciphertexts. With added password protection and interface improvements, users can perform anamorphic encryption more easily and securely. While there is still room for future work, such as additional encryption schemes and improved interface features, this version is a strong basis for practical anamorphic cryptography.

References

[1] S. Agramunt-Puig, "Discrete logarithm problem and Diffie-Hellman key exchange," Medium, Nov. 23, 2020. Available at: https://sebastiaagramunt.medium.com/discrete-logarithm-problem-and-diffie-hellman-key-exchange-821a45202d26.

[2] R. Code, "Safe primes and unsafe primes - Rosetta Code," Rosetta Code, Dec. 13, 2025. Available at: https://rosettacode.org/wiki/Safe$_{p}rimes_{a}nd_{u}nsafe_{p}rimes$

[3] GeeksforGeeks, "Brute Force Attack," GeeksforGeeks, Jan. 27, 2020. Available at: https://www.geeksforgeeks.org/computer-networks/brute-force-attack/

[4] What is a generator, "What is a generator?," Cryptography Stack Exchange, May 15, 2014. Available at: https://crypto.stackexchange.com/questions/16196/what-is-a-generator

[5] "Group Theory - Cyclic Groups," Stanford.edu, 2025. Available at: https://crypto.stanford.edu/pbc/notes/group/cyclic.html

[6] GeeksforGeeks, "ElGamal Encryption Algorithm," GeeksforGeeks, Nov. 11, 2018. Available at: https://www.geeksforgeeks.org/computer-networks/elgamal-encryption-algorithm/

[7] "The Baby-Step-Giant-Step algorithm", M2-HCMC2023. Available at: https://www.mat.uniroma2.it/ geatti/HCMC2023/Lecture4.pdf

[8] T. Kivinen and M. Kojo, "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). Available at: https://www.rfc-editor.org/rfc/rfc3526page-3