The Affine-Exponent Cipher

MATH 3CY3 - Cryptography Written Project

Student ID: 400144964

Name: Shakar Saeed

Due: April 1-8, 2025

Section 1 - Outline

Our Affine-Exponent Crypto system is a **private** key cryptographic system, it is based on the usual modular arithmetic, $\mathbb{Z}_p$, and with the exponent being in $\mathbb{Z}_{p-1}$. It is also closely related to solving Discrete Log Problem at certain steps.

This system is inspired by the usual Affine Cipher Crypto system, where the Encryption Algorithm is:

$$E(m) \equiv c \equiv a \cdot m + b \ (mod\ p) \qquad \text{for modular arithmetic p}$$

where its decryption algorithm is well known such that:

$$D(c) \equiv m \equiv a^{-1} \cdot (c - b) \ (mod\ p)$$

Instead in our Affine-Exponent Cipher System, we first apply an exponent to the message `m` we want to decrypt.

In the classical case of communication between Alice and Bob, Alice sets up the crypto system, Bob encrypts his plain text message `m` and he sends out his cipher text `c` to Alice where she uses her decryption algorithm to decrypt the cipher text `c` and receive the plain text message `m`.

Section 2 - Encryption

Our Encryption Algorithm is: $\boxed{E(m) \equiv (a \cdot m^e + b) \ \ (mod\ p)}$, where p prime ≥ 2, and we require:

$1 \leq m < p,\ gcd(a, p) = 1$ and $gcd(e, p-1) = 1$ for decryption to work later.

Also note that our plaintext messages `m` is not necessarily a generator $\pmod p$.

Eg.: Suppose p = 5, m = 2, e = 3. a = 4, b = 1.

Note that: $gcd(a, p) = gcd(4, 5) = 1$ and $gcd(e, p-1) = gcd(3, 4) = 1$

$$c \equiv E[2] \equiv 4 \cdot 2^3 + 1 \equiv 4 \cdot 8 + 1 \equiv 33 \equiv 3 \pmod 5$$

Section 3 - Decryption

Let $E(m) \equiv c \equiv a \cdot m^e + b \pmod p \implies c - b \equiv a \cdot m^e \implies m^e \equiv a^{-1} \cdot (c - b)$ such that $gcd(a, p) = 1 \implies (m^e)^d \equiv m^{e \cdot d} \equiv m \equiv (a^{-1} \cdot (c - b))^d$ such that: $d \cdot e \equiv 1 \pmod{p-1}$ where $gcd(e\ ,\ p-1) = gcd(d\ ,\ p-1) = 1$.

Therefore, the Decryption Algorithm is: $\boxed{D(c) \equiv (a^{-1} \cdot (c - b))^d \ \ (mod\ p)}$.

E.g.: Recall the previous example where p = 5, m = 2, e = 3. a = 4, b = 1.

Alice has previously computed her exponent d by Euclidean algorithm where $gcd(e, p-1) = gcd(3, 4) = 1$, such that $4 = 1 \cdot 3 + 1$.

Then by Back-substitution: $1 = 4 - 1 \cdot 3 \iff 1 \equiv -1 \cdot 3 \ (mod\ 4)$

In other words, $d \equiv -1 \equiv 3 \ (mod\ (p-1)) \equiv 3 \ (mod\ 4)$.

The multiplicative inverse of a = 4 can also be computed in a similar fashion except in $(mod\ 5)$. Note that by inspection, $a^{-1} \equiv -1 \equiv 4 \ (mod\ 5)$ since $a \cdot a^{-}1 \equiv 4 \cdot 4 \equiv 16 \equiv 1 \ (mod\ 5)$.

So, in this case Alice decrypts Bob's cipher text by computing:

$(a^{-1} \cdot (c-b))^d \equiv 4 \cdot (3-1)^3 \equiv 4(2^3) \equiv 32 \equiv 2 \ (mod\ 5)$ which is Bob's plain-text `m`.

(Click here to refer to Example of Encryption & Decryption Algorithm).

Section 4 - Security

Assuming that the encryption algorithm and decryption algorithm is public or well-known (Which even if it wasn't then an attacker, say Eve, could obtain both by deriving one if they have at least the other through social engineering).

Note that the private data are `(a, b, e, d, p)` the only public data being shared to the attacker is really the cipher text `c` if say Alice and Bob are communicating through an insecure channel of communications.

Aside from getting the private data including p by social engineering, if the attacker does not know p then they can make an educated guess of what p could be, although this is not always guaranteed.

Assuming that the attacker knows the system is in $\mathbb{Z}_p$, then since they know that the cipher text must be in the range $0 \leq c < p$ they need to just find the biggest number in the list of numbers of the cipher text that they have intercepted from Bob and their guess of p would be the next prime after that biggest number they found.

This method of finding p would be relatively quick especially if the cipher text that Bob shares to Alice is a long list of numbers depending on the length of the message he has sent by converting his English words mod 26 to numbers in mod p (for sake of simplicity we'll assume messages are automatically sent back and forth as numbers (mod p)). Although it may not always work especially if the cipher text is short. So the longer the cipher-text the more likely that the attacker would be able to guess what p is. For argument's sake, let's assume the attacker knows p from here on.

Also, note that since the plaintext message `m` is not necessarily a generator, it's possible that the order of m divides $p - 1$, which is not very ideal

for security as it would take the attacker less iterations to figure out how to decrypt (say by brute forcing) than if it was a generator. Although even if it was a generator, it doesn't add a lot to security, for reasons to be discussed soon.

Then, all they have left to figure out is `a, b, and e` (since d can be solved through Back-substitution Euclid's Algorithm $\pmod{p-1}$ if they know e).

Next, aside from brute forcing testing a, b, and e, to solve the rest, the attacker would first apply a statistical attack where they either figure out some easy or common words from sub-segments of the plain-text corresponding to their sub-segments of the cipher-text through Frequency Analysis or just by chance, such as the words `"Hello"` or `"the"` (converted into numbers $\pmod{p}$) and so on (in our code example, we separated plaintexts/ciphertexts by letter instead of by word, for sake of simplicity).

So:

$$\begin{array}{lcl}
c_1 \equiv a \cdot m_1^e + b \pmod{p} & & c_1 - c_2 \equiv a \cdot (m_1^e - m_2^e) \pmod{p} \quad ① \\
c_2 \equiv a \cdot m_2^e + b \pmod{p} & & c_2 - c_3 \equiv a \cdot (m_2^e - m_3^e) \pmod{p} \quad ② \\
c_3 \equiv a \cdot m_3^e + b \pmod{p} & \Longrightarrow & c_1 - c_3 \equiv a \cdot (m_1^e - m_3^e) \pmod{p} \quad ③ \quad \text{etc.} \\
\vdots & & \vdots \\
c_n \equiv a \cdot m_n^e + b \pmod{p} & & c_n - c_{n-1} \equiv a \cdot (m_n^e - m_{n-1}^e) \pmod{p} \quad ⓝ
\end{array}$$

for n many number of systems of equations.

Then, from ① and ② we initially guess $a = gcd(c_1 - c_2, c_2 - c_3)$. Note that:

$$gcd(c_1 - c_2, c_2 - c_3) = gcd(a \cdot (m_1^e - m_2^e),\ a \cdot (m_2^e - m_3^e))$$

$$= a \cdot gcd(m_1^e - m_2^e, m_2^e - m_3^e)$$

If $gcd(m_1^e - m_2^e, m_2^e - m_3^e) = 1$ then $gcd(c_1 - c_2, c_2 - c_3) = a$, otherwise $gcd(c_1 - c_2, c_2 - c_3) = a \cdot d$, where d some common greatest divisor such that $d = gcd(m_1^e - m_2^e, m_2^e - m_3^e)$.

So our initial guess **fails** if $gcd(m_1^e - m_2^e, m_2^e - m_3^e) \neq 1$. However, if the system of equation is large enough, one of the gcd's will give us `a` such that:

$$gcd(c_i - c_j, c_l - c_k) = a$$

So, we compute g the greatest common factor $g = a \cdot gcd(m_1^e - m_2^e, m_2^e - m_3^e)$ and then factor g into its prime factors, through various methods discussed in class such as iterating through $\sqrt{g}$ or Pollard's $p-1$ method, Quadratic Sieve etc. , such that we get g in the form of multiples of a, where $g = k \cdot a$, so $a|g$.

Since we're looking for a where g is a multiple of a, it's likely that a could be relatively small compared to g for the system to be practical. Then, we test combinations of small factors j = a of g where we have $gcd(j, p) = gcd(a, p) = 1$ and we must find an a that satisfies ① - ②:

$$c_1 - c_2 = a \cdot (m_1^e - m_2^e) \pmod{p} \implies X = a^{-1} \cdot (c_1 - c_2) \equiv m_1^e - m_2^e \pmod{p}$$

Once we find such a, to find b first suppose we know a cipher-text c' corresponds to a plain-text m' = 1, such that $c' \equiv a \cdot (1)^e + b \equiv a \cdot 1 + b$, then $b \equiv c' - a \pmod{p}$.

Therefore, so far we have found ways to solve (or guess) for a, and for b. Now, all we have left to do is solve for the exponent e such that:

$$c \equiv a \cdot m^e + b \implies a^{-1} \cdot (c - b) \equiv m^e \implies m^e \equiv h \pmod{p}$$

Then, for c_1 corresponding to $m_1 \implies m_1^e \equiv a^{-1} \cdot (c_1 - b) \equiv h_1 \pmod{p}$, we can then solve for e by using Baby-Step Giant-Step Algorithm, as discussed in class:

First find order of m = N, then compute $n = 1 + \lfloor N \rfloor$ then compute baby-steps $1, m, m^2, \cdots, m^{n-1} \pmod{p}$ and Giant step $G = m^{-n}$ with Giant Steps List: $h_1, h_1 \cdot G, \cdots, h_1 \cdot G^{n-1} \pmod{p}$ Then once we find equality in the lists such that $m^a = h_1 \cdot G^b$, s.t. $0 \leq a, b < n$, then we recover the exponent e by $h_1 \cdot G^b = h_1 \cdot (m^{-n})^b \equiv m^a \implies h_1 = m^{a+b \cdot n}$, where $e = a + b \cdot n$

Alternatively:

$$a \equiv \frac{c_1 - c_2}{m_1^e - m_2^e} \equiv (c_1 - c_2) \cdot (m_1^e - m_2^e)^{-1} \pmod{p}$$ if you first find e by iterating.

Thus, $b \equiv c_1 - a \cdot m_1^e \pmod{p}$ once we found a and e. To find e first by iteration,

$$①/② \implies R = \frac{c_1 - c_2}{c_2 - c_3} \equiv \frac{m_1^e - m_2^e}{m_2^e - m_3^e} \pmod{p}, \text{ where } R \equiv (c_1 - c_2) \cdot (c_2 - c_3)^{-1} \pmod{p}$$

Since m is not necessarily a generator, we likely have a sub-cyclic group of units (which should reduce number of computations) but not guaranteed, so we could test various e's to the ratio R that satisfies it.

Problem: number of computations in this ratio R is at worst $p - 1$ exponentiations, 3 exponentiations for the 3 m's and 2 subtractions. So at worst, there is $(p-1) * (3+2)$ number of computations for brute forcing by loops. Not to mention it's prone to error if the modular inverse of the denominator is not invertible where $m_2^e \equiv m_3^e \pmod{p}$.

We can manipulate the above ratio equation to find a root-finding function such that:
$R \cdot m_2^e - R \cdot m_3^e \equiv m_1^e - m_2^e \implies f(e) = m_1^e - (1+R) \cdot m_2^e + R \cdot m_3^e \equiv 0 \pmod{p}$
(Shown in attached file in Appendix (Example of Breaking the System)).
Iterate over all possible e's up from 1 to $p-1$ (instead of 0 to $p-2$) to find our e. However, it's likely that e will be relatively small compared to $p-1$. So, for sake of efficiency we can iterate through smaller e's first. This function is always defined and no modular inverses are needed.

Now, we know our e, and we previously found our b and our guesses for a.

Alternatively, since we have found e by iterating, we can then find a and b by: $a \equiv (c_1 - c_2) \cdot (m_1^e - m_2^e)^{-1} \pmod{p}$ and $b \equiv c_1 - a \cdot m_1^e \pmod{p}$ which is what we'll do in the coding example since this method is more computationally robust than having to guess for `a` first which may not always work or if the list of cipher text is not big enough to compute gcd of all of it to find `a` that satisfies our need.

So far, we have `a, b, e, p`. Last step is to find `d` by using Euclidean Algorithm on e to find $e^{-1} \equiv d \pmod{p-1}$ to compute $m \equiv (a^{-1} \cdot (c-b))^d \equiv m^{e \cdot d} \pmod{p}$.
Now we can easily decrypt the cipher texts using the Decryption Algorithm.

Therefore, clearly this system is not very secure, especially since I've just pointed out at least 2 ways to break it.
(Click here to refer to the Example of Breaking the System in the Appendix).

Appendix: Codes

Encryption & Decryption Functions:

```
def encrypt_affine_exponent(m, a, b, e, p):
    return (a * pow(m, e, p) + b) % p


def decrypt_affine_exponent(c, a, b, e, p):
    # Step 1: undo b and scale down by a
    a_inv = pow(a, -1, p)
    x = (a_inv * (c - b)) % p

    # Step 2: compute e-th modular root by raising to
       e^{-1} mod (p-1)
    e_inv = pow(e, -1, p - 1)
    m = pow(x, e_inv, p)
```

```
    return m
```

Example of Encrypting and Decrypting:

```
p = 5; m = 2; e = 3; a = 4; b = 1

cipher = encrypt_affine_exponent(m, a, b, e, p)
print("cipher_text , c = ", cipher)

plain = decrypt_affine_exponent(cipher, a, b, e, p)
print("plain_text , m = ", plain)
```

Output:

```
cipher_text, c = 3
plain_text, m = 2
```

Example of Breaking the System: (click on the pin for the full details)

Note that since the code is very long for breaking the system, we will only show the final output. For full results (OPTIONAL), feel free to see the attached document by clicking on the pin icon (NEED TO VIEW in Adobe):  .

So, if the plaintext was first converted into numbers, char by char, then the numbers was encrypted using our affine-exponent system. Suppose we have a very big list for the cipher text as an example of length 485 characters/numbers (shown in the attached document):

Ciphertext nums: [96, 110, 31, 31, 65, 149, ..., 98, 187, 5, 9, 110, 3, 98, 96]

First we guess p to be the next prime after the max value of ciphertext nums, then after some frequency analysis and matching respective characters to common english alphabets, we find e by using the iterative root finding function previously discussed and we find many possible e's that is a root. Then we find a and b with those e's, and then we decrypt the ciphertext nums using those combinations of data `(e, a, b, p)` and convert the decrypted numbers back into characters.

Finally, we scan which decrypted texts with corresponding data `(e, a, b, p)` contains as many english words as possible that make sense, by matching to most common english words. Then we use our data to decrypt the full ciphertext numbers into plain english text:

```
'HELLOWORLDTHEQUICKBROWNFOXJUMPSOVERTHELAZYDOGMANYTIMESINTHE
DARKNESSWHILESTARS .... MOMENTOFSUSPENSETHEWORLDHOLDSITSBREATH'
```