Project Cryptography

Project Files

You can download a zip file with all necessary project files here.

Goals of the Project For Students

- Will get an introduction to both symmetric and asymmetric cryptographic systems
- Will gain an understanding of how these systems are implemented through examples
- Will exploit systems that have certain vulnerabilities
- Will have the opportunity to engage in discussion about advanced topics in cryptography

Information

There is no required VM for this project. All that is required is a Python development environment. Make certain that you are using Python 3. To check your version of Python, open a command prompt and run the command:
`python --version`. (You may need to use the `python3` command instead.)

For the established algorithms that you may find it necessary to use, you are allowed to reference and implement pseudocode with citation (a comment in your code will suffice)**.** What is Pseudocode? https://en.wikipedia.org/wiki/Pseudocode

UNDER NO CIRCUMSTANCES should you copy/paste code into the project. Doing so is an honor code violation (not to mention a real world security concern) and will result in a zero (refer to the syllabus for more information).

The Final Deliverable

You will complete the provided Python file **project_cryptography.py** and submit it to the autograder in Gradescope.

Open Discussions

For each task we have provided prompts for further discussions. There will be threads created in Ed where students can discuss these topics. Participation is optional and will not be graded.

Task Vigenere Ciphers (35 points)

The Vigenere cipher is an example of a symmetric key cryptographic algorithm. In such a system, a single key is used to both encrypt and decrypt messages (this is what makes it symmetric). The first step for both encryption and decryption is to build a Vignere square like the one pictured below. In each row of the Vigenere square, the letters of the alphabet are shifted to the left by one. The second step is to extend the key to match the length of the cipher/message by repeating it, taking care to remove any spaces or punctuation from the original message. For example, if our message is **GEORGIA** and our key is **TECH** we would end up with a key **TECHTEC**.

To encrypt a message, we lookup each letter of the message as a row, and find its intersection with the column whose label contains the corresponding letter from the key. Using our **GEORGIA**/**TECH** example, the intersection of the first letter of the message **G** with the first letter of the key **T** is **Z**. Going letter-by-letter in this manner we build our cipher until we get **ZIQYZMC**.

To decrypt a cipher, we start with the first letter of the key and traverse that row until we reach the corresponding letter from the cipher. The label of this column is our decrypted letter. Using our same **GEORGIA**/**TECH** example, to decrypt the cipher we would start with the first letter of our key **T** and traverse that row until we reached the first letter of the cipher **Z**. The label of this column is **G** - the first letter in our message.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Vigenere Square [source: Wikipedia]

Armed with this information, the first two parts of this task will ask you to write the necessary code to handle the encryption and decryption functionality for a Vigenere cipher system.

For the third and final part, you will attempt to crack a Vigenere cipher using a dictionary attack. Ordinary words can make convenient keys because they are easy to remember, but this practice is far from secure. For this task, you are given a cipher and a list of some of the most common words in the English language. One of those words was used as the key to encrypt the cipher, and your job is to write the code to figure out which one it is. For simplicity, you can assume that all words in the original message are also chosen from the provided list of dictionary words.

```
def vigenere_decrypt_cipher(c: str, keyword: str) -> str:
    # TODO: Write the necessary code to get the message (m) from the cipher ©
    # using the keyword
    m = ‘’
    return m

def vigenere_encrypt_message(m: str, keyword: str) -> str:
```

```
    # TODO: Write the necessary code to create a Vigenere cipher (c) of the
    # message (m) using the provided keyword
    c = ''
    return c

def vigenere_dictionary_attack(c: str) -> str:
    # TODO: Write the necessary code to get the message (m) from the cipher (c)
    m = ''
    return m
```

Submission Details

You will write your code in the specified function stubs found in the provided **project_cryptography.py** file. When ready, submit this file to the **Project Cryptography** autograder in Gradescope.

Open Discussion

The Vigenere cipher improves upon the ancient Caesar cipher. Mathematically speaking, how much more complexity does it add and how does it accomplish this? What are some ways that one could add more complexity/security to the Vigenere cipher system? See Discussion Question 1 thread.

Task RSA Warmup (10 points)

Now that we've reviewed a symmetric key cryptographic algorithm, we can move on to the world of asymmetric key cryptography. RSA is perhaps the best known example of asymmetric cryptography. In RSA, the public key is a pair of integers (N, e), and the private key is an integer d.

To encrypt integer m with public key (N, e), we use the formula $c \equiv m^{e} \bmod N$.
To decrypt cipher integer c with private key d, we use the formula $m \equiv c^{d} \bmod N$.

In this task you will write the code to perform the encryption and decryption steps for the RSA cryptographic algorithm. Finally, you will write the code necessary to calculate the private key **d** when given the factors of the public key **N** (i.e. **p** and **q**).

```
def rsa_decrypt_cipher(n: int, d: int, c: int) -> int:
    # TODO: Write the necessary code to get the message (m) from the cipher (c)
    m = 0
    return m

def rsa_encrypt_message(m: int, e: int, n: int) -> int:
    # TODO: Write the necessary code to get the cipher (c) from the message (m)
    c = 0
    return c

def rsa_calculate_private_key(e: int, p: int, q: int) -> int:
    # TODO: Write the necessary code to get the private key d from
    # the public exponent e and the factors p and q
```

```
    d = 0
    return d
```

Submission Details

You will write your code in the specified function stubs found in the provided **project_cryptography.py** file. When ready, submit this file to the **Project Cryptography** autograder in Gradescope.

Open Discussion

Did you try to decrypt a cipher by using a line of Python code something like this: m = c ** d % n? Did it work? (Hint: It did not.) Why not? After all, the math is correct. See Discussion Question 2 thread.

Task RSA Factor A 64-Bit Key (10 points)

Modern day RSA keys are sufficiently large that it is impossible for attackers to traverse the entire key space with limited resources. But in this task, you're given a unique set of RSA public keys with a relatively small key size (**64 bits**).

Your goal is to get the factors (p and q) of each key. **You can use whatever methodology you want.** Your only deliverable is a formatted json file containing p and q. To get your unique set of keys, you must update the task.py file located in the task folder with your 9-digit GT ID, and then run it. Find the section below in the provided **task_factor_64_bit_key.py** file:

```
##############################################
# Change this to your 9-digit Georgia Tech ID!
STUDENT_ID = '123456789'
##############################################
```

Running the command "python task_factor_64_bit_key.py" should output your assigned keys. Once you've calculated your p and q values, enter them into the function stub for this task. **NOTE: It doesn't matter which value you specify as p and which value you specify as q.**

```
def rsa_factor_64_bit_key() -> typing.Dict[str, typing.Dict[str, int]]:
    return {
        'test_1': {
            'p': 0,
            'q': 1
        },
        'test_2': {
            'p': 0,
            'q': 1
        },
        'test_3': {
            'p': 0,
            'q': 1
        },
```

```
        'test_4': {
            'p': 0,
            'q': 1
        },
        'test_5': {
            'p': 0,
            'q': 1
        }
    }
```

Submission Details

You will write your code in the specified function stubs found in the provided **project_cryptography.py** file. When ready, submit this file to the **Project Cryptography** autograder in Gradescope.

Open Discussion

If 64-bit keys aren't safe, then what size is appropriate? Is there a trade-off between size and performance? See Discussion Question 3 thread.

Task RSA Weak Key Attack (15 Points)

Read the paper "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices", which can be found at: https://factorable.net/weakkeys12.extended.pdf. The essay is essential to understanding this task. Do not skip it, do not skim it, read the whole of it.

You are given a unique RSA public key, but the RNG (random number generator) used in the key generation suffers from a vulnerability described in the paper above. In addition, you are given a list of public keys that were generated by the same RNG on the same system. Your goal is to write the code to get the unique private key (d) from your given public key (N, e) using only this provided information.

```
def rsa_weak_key_attack(given_public_key_N: int, given_public_key_e: int, public_key_list: typing.List[int]) -> int:
    # TODO: Write the necessary code to retrieve the private key d from the given public
    # key (N, e) using only the list of public keys generated using the same flawed RNG
    d = 0
    return d
```

Submission Details

You will write your code in the specified function stubs found in the provided **project_cryptography.py** file. When ready, submit this file to the **Project Cryptography** autograder in Gradescope.

Open Discussion

Have you ever heard the saying, "Never roll your own crypto?" What are some ways (besides this particular attack - we don't want you to give too much away) that doing so can cause unintended problems? Can you point to any specific examples or known exploits? See Discussion Question 4 thread.

Task RSA Broadcast Attack (15 Points)

A message was encrypted with three different 1,024-bit RSA public keys (N_1, N_2, and N_3), resulting in three different ciphers (c_1, c_2, and c_3). All of them have the same public exponent $e = 3$.

You are given the three pairs of public keys and associated ciphers. Your job is to write the code to recover the original message.

```
def rsa_broadcast_attack(N_1: int, c_1: int, N_2: int, c_2: int, N_3: int, c_3: int) -> int:
    # TODO: Write the necessary code to retrieve the decrypted message (m) using three different
    # ciphers (c_1, c_2, and c_3) created using three different public key N's (N_1, N_2, and N_3)
    m = 0
    return m
```

Submission Details

You will write your code in the specified function stubs found in the provided **project_cryptography.py** file. When ready, submit this file to the **Project Cryptography** autograder in Gradescope.

Open Discussion

In addition to the low public exponent being used, this attack is possible because a textbook implementation of RSA is being used. In the real world, there are common mitigating tactics used. What are some examples? Why else are they important? See Discussion Question 5 thread.

Task RSA Parity Oracle Attack (15 Points)

By now you have seen that RSA treats messages and ciphers as ordinary integers. This means that you can perform arbitrary math with them. And in certain situations a resourceful hacker can use this to his or her advantage. This task demonstrates one of those situations.

Along with an encrypted message (c), you are given a special function that you can call - a parity oracle. This function will accept any integer value that you send to it and decrypt it with the private key corresponding to the public key that was used to encrypt the given cipher (c). The return value of the function will indicate whether this decrypted value is even (true) or odd (false). Armed with this function and a little modular arithmetic, it is possible to crack the encrypted message. Your goal is to write the code necessary to decrypt the original message (m) from the given cipher (c).

```
def rsa_parity_oracle_attack(c: int, N: int, e: int, oracle: Callable[[int], bool]) -> str:
    # TODO: Write the necessary code to get the plaintext message from the cipher (c) using
    # the public key (N, e) and an oracle function - oracle(chosen_c) that will give you
    # the parity of the decrypted value of a chosen cipher (chosen_c) value using the hidden private key (d)
    m = 42

   # Transform the integer value of the message into a human readable form
    message = bytes.fromhex(hex(int(m_int)).rstrip('L')[2:]).decode('utf-8')
```

```
    return message
```

Submission Details

You will write your code in the specified function stubs found in the provided **project_cryptography.py** file. When ready, submit this file to the **Project Cryptography** autograder in Gradescope.

Open Discussion

This task is a simplified example, but can you see how some potentially useful information may be inadvertently leaked by something (i.e. a protocol)? Can you find any examples? See Discussion Question 6 thread.

BONUS: Task Affine Ciphers (+2 points)

The affine cipher is a kind of monoalphabetic substitution cipher. In this case, the substitution rule is determined by a simple mathematical formula that relates each letter to its corresponding numerical value. This formula ensures that every letter is encrypted to a unique letter, and can be decrypted back to its original form. Because the affine cipher follows this rule for its substitution, it shares the same vulnerabilities as other substitution ciphers.

To encrypt a message using the given keyset(a, b) involves the following steps:

1. Determine the size of the alphabet (m). In most cases, the size of the alphabet is 26 for English letters a to z. However, it could be different if you are using a self-defined alphabet, or adding some special characters.
2. Map each letter in the alphabet to its corresponding value. Typically, 'a' is mapped to 0, 'b' to 1, and so on, resulting in a range of integers from 0 to m-1.In some cases, the mapping could start from 1, creating a range from 1 to m.
3. Select a pair of keys (a, b) to be used in the encryption process. The key 'a' must be coprime with the size of the alphabet 'm', meaning that their greatest common divisor is 1.
4. For each letter in the plaintext message, find its corresponding integer value x using table 1.
5. Apply the affine cipher encryption formula below to compute the encrypted integer value.

$$E(x) = (ax + b) \bmod m$$

6. Use the calculated integer value to find the letter for the cipher in the alphabet table.
7. Repeat each letter in the plain text until you get the full cipher text.

To decrypt a cipher using the given keyset(a, b) one reverses the encryption process::

1. Determine the m, size of the alphabet.
2. Map each letter in the cipher text to the integer value in the table.
3. Calculate the multiplicative inverse of 'a' modulo 'm'
4. Using the decryption formula, get the value for the decrypted letter.
5. Find the decrypted value corresponding to the letter in the alphabet table.

$$D(x) = a^{-1}(x - b) \bmod m$$

Decrypt a message by determining the keyset by yourself.

Decrypt a message by identifying the keyset independently. In this task, the keyset (a, b) is not provided, and you must determine it yourself. Return the plaintext message's SHA-256 hash for submission. In real-world scenarios, it is highly likely that you won't know the keyset. If you are aware that the message has been encrypted using the affine cipher, you need to devise a method to obtain the keyset and then decrypt the message.

Notes:
The alphabet in this task will be 29 characters - lowercase a to z, space, comma and period.
The index of a is 0, b is 1…and space is 26, comma is 27, period is 28.

a	0	k	10	u	20
b	1	l	11	v	21
c	2	m	12	w	22
d	3	n	13	x	23
e	4	o	14	y	24
f	5	p	15	z	25
g	6	q	16		26
h	7	r	17	,	27
i	8	s	18	.	28
j	9	t	19		

Table 1: Index value for each letter. The alphabet size m is 29 as seen in the table.

Example:
To encrypt “hello world.” using keyset (a=2, b=3), the alphabeta is the same as Table 1.

	h	e	l	l	o		w	o	r	l	d	.
index	7	4	11	11	14	26	22	14	17	11	3	28
(ax+b)%29	17	11	25	25	2	26	18	2	8	25	9	1
Encrypted	r	l	z	z	c		s	c	i	z	j	b

The ciphertext is “rlzzc scizjb”

Useful resources:
Affine cipher Wikipedia: https://en.wikipedia.org/wiki/Affine_cipher

```python
def affine_encrypt_message(message: str, keyset: tuple) -> str:
    # TODO: Write the necessary code to create a Affine cipher of the message using the provided keyset
    alphabet = "abcdefghijklmnopqrstuvwxyz ,."
    cipher = ''
    return cipher


def affine_decrypt_cipher(cipher: str, keyset: tuple) -> str:
    # TODO: Write the necessary code to get the message from the cipher using the keyset
    alphabet = "abcdefghijklmnopqrstuvwxyz ,."
    message = ''
    return message


def affine_decrypt_without_key(cipher: str) -> str:
    # TODO: Write the necessary code to get the message from the cipher by determining the keyset on your own
    alphabet = "abcdefghijklmnopqrstuvwxyz ,."
    message = ''
    return hashlib.sha256(message.encode()).hexdigest()
```

Submission Details

You will write your code in the specified function stubs found in the provided **project_cryptography.py** file. When ready, submit this file to the **Project Cryptography** autograder in Gradescope.

Open Discussion

What are some ways that one could add more complexity/security to the Affine cipher system? What are the vulnerabilities? See Discussion Question 7 thread.

Important Notes:

All necessary starter code and unit tests for each task is located in the corresponding folder in the provided zip file.

You may import any python packages that are part of the standard library. Some useful ones are already imported for you, and additional ones are not strictly necessary.

For each task you are also given a unit test file (it starts with `test_`)to help you develop and test your code. We encourage you to read up on Python unit tests, but in general, the syntax should resemble either:

```
python -m unittest test_task_rsa_encrypt_message
```

or:

```
python test_task_rsa_encrypt_message.py
```

However, keep in mind that passing the unit test(s) does NOT guarantee that your code will pass the autograder!

The autograder will timeout after 10 minutes. If your implementation is timing out, then there is very likely something wrong with your implementation. It is possible to solve each task in a few seconds. We encourage you to test locally to avoid unnecessary submissions.

Gradescope can get very busy and even potentially unavailable near submission deadlines. **Please do not wait until the last minute to make your submissions to the autograder. Submit early and often. There will be no late submissions accepted, as per the syllabus.**

Good luck!