UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 **Computer Security and Privacy** **Fall 2020**

ASSIGNMENT 3

- Milestone deadline: **November 23rd 2020 at 3:00 pm**
- Assignment deadline: **December 4th 2020 at 3:00 pm**
- The usual 48-hour automatic extension applies to ~~both deadlines~~ the final assignment deadline above.

Total Marks: 76 plus 4 possible bonus marks

Written Response TAs:
- Thomas Humphries: t3humphr@uwaterloo.ca

Programming Response TA:
- Lindsey Tulloch: ltulloch@uwaterloo.ca
- Sina Faraji: sina.faraji@uwaterloo.ca

TA Office Hours: Wednesdays 11:00 - 12:00 EDT

Please use Piazza for questions and clarifications. We will be using Big Blue Button for TA office hours this term; we have separate online rooms for the written and programming parts. To attend office hours, access the corresponding URL for that assignment part and use the corresponding access code when prompted:
Written: https://bbb.crysp.org/b/you-9uu-vgr - Access code: 452674
Programming: https://bbb.crysp.org/b/you-ggq-k6d - Access code: 450936
When asked for your name please enter both your first and last name as they appear in LEARN.

What to hand-in

For **programming questions**, the assignment website automatically updates your unofficial marks and records the completion timestamps whenever you successfully address a task. You can see your unofficial mark breakdown in the dashboard of the assignment website. The grade becomes official once your code submission has been examined. Please make sure you complete the tasks on the website and submit your code before the deadlines.

Using the "submit" facility on the student.cs machines (**not** the ugster machines or the UML virtual environment), hand in the following files for the appropriate deadlines:

1. **Milestone:**
   a3code-q1.tar: an uncompressed tar file containing your code for the first programming question (Q1). While we will not run your code, it should be clear to see how your code can issue web requests to the API. If it is not obvious to see how you solved a question, then you will not receive the marks for that question, even if points were awarded on the assignment website. **Note that you will not be able to submit the tarball for the first question after the milestone deadline.**
2. **Rest of assignment:**
   a3-responses.pdf: A PDF file containing your answers for all written questions. It must contain, at the top of the first page, your name, UW userid, and student number. *-3 marks if it doesn't!* Be sure to "embed all fonts" into your PDF file. Some students' files were unreadable in the past; if we can't read it, we can't mark it.

   Q1 Written Binary Files: For written question 1(a), you should submit a binary file `xor` as your solution and for written question 1(b) you are expected to submit two 300-byte files called `plaintext1` and `plaintext2` as part of your solution.

   a3code.tar an uncompressed tar file containing your code for all parts of the programming question except for Assignment 1-Q1 (the milestone).
   Arrange your file in two folders named **A1** and **A2**. Put all your code for Assignment 1 in **A1** and your code and description file for Assignment 2 in **A2**.
   While we will not run your code for Assignment 1, it should be clear to see how your code can issue web requests to the API to solve each question. If it is not obvious to see how you solved a question, then you will not receive the marks for that question, even if points were awarded on the assignment website.
   Your code for Assignment 2 will be marked according to the assignment description.

Written Response Questions [33 marks]

Note: Please ensure that written questions are answered using complete and grammatically correct sentences. You will be marked on the presentation and clarity of your answers as well as on the correctness of your answers. Also please remember to show your work for any calculations.

Question 1: Diffie-Hellman Key Exchange [9 marks]

Suppose Alice and Bob would like to exchange a secret key over an insecure channel. One way they could do this is using the Diffie-Hellman (DH) key exchange. See http://mathworld.wolfram.com/Diffie-HellmanProtocol.html for more information about this protocol.

Note: The parameters used in this question are very small as this is a toy example. Real Diffie-Hellman would use much larger numbers.

(a) [3 marks] Assume Alice and Bob have already agreed to use modulus $p = 107$ and base $g = 17$. Then Alice chooses secret parameter $a = 12$ and Bob chooses secret parameter $b = 34$. What is the public value A that Alice gives to Bob? What is the public value B that Bob gives to Alice? What is the resulting secret key that is generated as a result of DH protocol?

(b) [2 marks] During the key exchange, Eve observes g, p, $A = g^a \pmod{p}$, and $B = g^b \pmod{p}$. Can Eve recover the original secret values a or b using this information? Explain why or why not.

(c) [2 marks] What if Mallory comes along and behaves as an active Man-In-The-Middle (MITM) attacker, how can she manipulate the Diffie-Hellman protocol to obtain all of the plaintext communications between Alice and Bob? Explain.

(d) [2 marks] Provide a brief explanation of how Alice and Bob could prevent such an attack by Mallory.

Question 2: Textbook RSA [7 marks]

In RSA, the public key is a pair of integers (n, e), where $n = pq$ for large primes p and q. The private key is the triple (p, q, d) where $de \equiv 1 \mod (p-1)(q-1)$. In a simplified form of RSA, called "textbook RSA", the encryption of a message m to yield the ciphertext c is $c = m^e \mod n$ and the decryption is $m \equiv c^d \mod n$.

(a) [2 marks] Does textbook RSA provide semantic security? If so explain why, if not provide a simple countermeasure.

A key part of making RSA secure is to choose appropriate primes numbers. It is often stated that textbook RSA is weak because there are no restrictions on how to choose these primes (and for a number of other reasons as well). In this problem, we will investigate a variation of textbook RSA that puts restrictions on the primes p and q.

Let us assume a genius security expert, Frieda, thinks that the following way of generating primes would be a good idea: choose the numbers g, b, and a such that $p = 2ga - 1$ and $q = 2gb + 5$ are prime. Much like in textbook RSA, she then sets $n = pq$. During a round of beer, she boasts to her arch enemy Jimmy that her scheme is secure (that it is impossible to factor n into primes p and q in polynomial time).

Jimmy never took this (or any security) course and blindly believes that Frieda has devised an "unbreakable" encryption scheme. To impress another security expert, John, Jimmy gets involved in a bet that this scheme is unbreakable, even if given the values of a and b. John is intelligent, so after looking at the encryption scheme closely, he asks Jimmy to give him any two values of a and b that fit the scheme. Blinded by incomplete knowledge, Jimmy gives John the values of a and b.

We will see how John can factor n in time that is polynomial in the bit length of n and win the bet.

(b) [2 marks] Write the number n in terms of g, a and b.

(c) [2 marks] Use the above relation to give a closed-form expression for g.

(d) [1 marks] Write in one sentence, how the above information helps you in finding the factors of n.

Question 3: Tracker Attacks [7 marks]

Suppose a local police department stores a table of the salaries of all staff members for a given precinct. As a result, the department stores a table of size N, `Employee`, in its database. **Below is an excerpt (not the entire table)**:

Name	Type	Position	Salary
Fatima	Full Time	Traffic Officer	64,500
William	Full Time	Detective	90,315
Lucas	Full Time	Receptionist	50,751
Harper	Part Time	Police Officer	49,426
Olivia	Full Time	Police Officer	69,908
Manuel	Part Time	Police Clerk	42,923
Mariam	Part Time	Police Officer	48,129

Table 1: Part of `Employee` Table

To deter employees from learning other people's salary, the database is set up to suppress the `Salary` field in the output of queries. However, users can execute queries of the form `SELECT SUM(Salary) FROM Employee WHERE ...` where queries that match fewer than k or more than $N - k$ (but not all N) records are rejected. We will use $k = \frac{N}{8}$.

(a) [4 marks] Use a tracker attack, as defined in class, to design a tracker and a set of three queries based on this tracker that will let you infer Olivia's expected salary. Both the tracker and the three queries need to be of the form `SELECT SUM(Salary) FROM Employee...` Assume that names are unique and not known to the attacker (apart from Olivia's) and that the attacker has no additional information about Olivia. For the tracker attack to succeed, the attacker does need to make an assumption about the distribution of (some of) the values stored in the database. This assumption should be realistic so that your tracker attack (likely) would also work on a different set of records, not only on the set shown above. In your solution, you should give

i) Your assumption.

ii) Your Tracker.

iii) The set of 3 queries you will use.

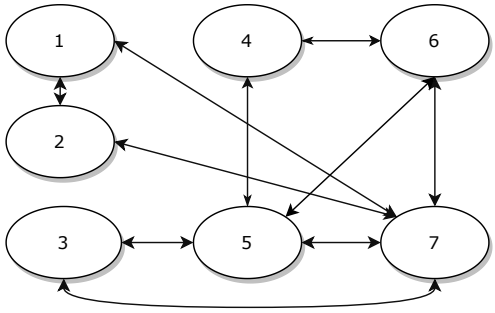
iv) How you can use the results of the three queries to obtain Olivia's Salary.

(b) [3 marks] The department becomes aware of the tracker attack and forbids `SUM(Salary)` queries. Instead the department allows only queries of the form `SELECT COUNT(*) FROM Employee WHERE Q`. Note that `Q` may include testing the value of the Salary field. (Again, queries that match fewer than k or more than $N - k$, with the exception of N, records are ruled out.) How would you modify your attack from part (a) to learn Sarah's expected salary(Not

shown in the table)? Explain the new attack. You may assume that no one in the database has an expected salary more than $200,000 and that all expected salaries are non-negative if that simplifies your attack.

Question 4: Naive Anonymization [10 marks]

The Acme Corporation decides to publish an email communication graph to show the how various departments are interacting with each other. To preserve anonymity they make sure to remove all employee names and instead assign an ID to each node. Below is the graph they released.



As we have seen from examples in class, naive anonymization techniques such at this are not robust to background information.

(a) [2 marks] Assume it is well known around the office that Cathy emails *everyone* except Ed (because of that one incident at the Christmas party). You also know that Alice only ever emails Bob, Cathy, and Ed. Using this knowledge can you reveal the identities of Alice, Bob, Cathy, and Ed? If so list their identities and if not explain why.

The Acme Corporation is all about publishing data. This time they wish to publish the number of emails sent by each person in the office. Once again they use the same IDs as part (a) instead of peoples names. However, they also apply a fixed secret linear perturbation function with secret parameters to the number of emails first. Below is the published table.

ID	Number of Emails
1	50
2	26
3	62
4	10
5	40
6	14
7	64

(b) [3 marks] After having their identities revealed in the last data release Alice and Cathy team up to see if they can break this anonymization. Suppose Alice knows her email count is 32 and Cathy knows hers is 57. Using this information can they obtain the true number of emails table? If so explain how and if not, explain why not.

Acme Corporation decides they are going to use k-anonymity to publish data about the manufacturing of their products. Specifically they decide the would like to modify the following data to be 3-anonymous.

Product Name	Manufacturing Facility	Net Income	Number of units sold
ACME Cactus Costume	Boston	-1	>100
ACME Smoke Bomb	Chicago	+16	>100
ACME Carrot	Detroit	-13	>100
ACME Patented Hair Grower	Boston	+5	<100
ACME Breakfast Cereal	Detroit	+2	<100
ACME Matches	Chicago	-3	>100
ACME Artificial Rock disguise	Boston	+25	>100
ACME Dynamite	Chicago	+12	<100
ACME Bird Seed	Detroit	+14	>100

They declare that Net Income (thousands), Number of units sold, and Product Name are identifiers. The Manufacturing Facility is considered sensitive information. The constraints placed on the data to be anonymized, while still allowing some utility, are the following:

- Product Name is masked.
- Number of units sold cannot be masked/modified.
- Net Income can only be generalized to at least 3 **equally sized** ranges.

As a result of the constraints, Acme Corporation chooses the following three net income ranges: [-14 - 0], [1- 14], [15-29]. The anonymized table is as follows

Product Name	Manufacturing Facility	Net Income	Number of units sold
*	Boston	[-14 - 0]	>100
*	Chicago	[15-29]	>100
*	Detroit	[-14 - 0]	>100
*	Boston	[1- 14]	<100
*	Detroit	[1- 14]	<100
*	Chicago	[-14 - 0]	>100
*	Boston	[15-29]	>100
*	Chicago	[1- 14]	<100
*	Detroit	[1- 14]	>100

(c) [3 marks] Is the anonymized table 3-anonymous? If yes, please explain and give the value of ℓ for which the table is ℓ-diverse. If no, please present a correct solution and give the value of ℓ

for which your table is ℓ-diverse.

Now we have a 3-anonymous table (either from Acme or your corrected table). However, the next day the following table is released that is 4-anonymous.

Product Name	Manufacturing Facility	Net Income	Number of units sold
*	Boston	[-14 - 0]	>100
*	Toronto	[-14 - 0]	>100
*	Vancouver	[-14 - 0]	>100
*	Vancouver	[-14 - 0]	>100
*	Vancouver	[1- 14]	<100
*	Austin	[1- 14]	<100
*	Austin	[1- 14]	<100
*	Austin	[1- 14]	<100

(d) [1 mark] Suppose Cathy knows that the Net Income for a ACME Cactus Costume is -1 and that Cactus Costumes are definitely in both tables. What can Cathy learn about Cactus Costumes from these two anonymized tables that she did not know before?

(e) [1 mark] It seems background knowledge can be a real problem for the Acme Corporation. Can you suggest a popular database privacy notion that is resilient to background knowledge?

Programming Question [43 marks + 4 bonus marks]

NOTE: There are seven questions. Questions 1—5 are mandatory questions and account for 24 marks. **Question 1 is the milestone question, and you should finish it before the milestone question deadline.** You are only required to answer **one** of Question 6 and Question 7 to get another 4 marks for the full 28 marks. If you attempt both Question 6 and 7 correctly, you can earn 4 bonus marks and access the "freedom" environment.

Note: This assignment is long because it includes a lot of text to help you through it.

> The use of strong encryption in personal communications may itself be a red flag. Still, the U.S. must recognize that encryption is bringing the golden age of technology-driven surveillance to a close.
>
> MIKE POMPEO
> *CIA director*

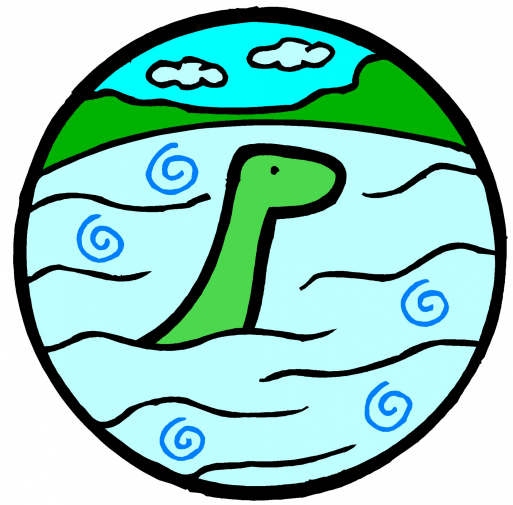
> Encryption works. Properly implemented strong crypto systems are one of the few things that you can rely on.
>
> EDWARD SNOWDEN

Assignment 1: Secure Messaging [28 marks + 4 bonus marks]

In this assignment, you will use "strong encryption" to send secure messages. Each question specifies a protocol for sending secure messages over the network. For each question, you will use the `libsodium`[1] cryptography library in a language of your choice to send a message through a web API.

For the assignment questions, you will send messages to and receive messages from a fake user named Nessie in order to confirm that your code is correct.

[1] `https://libsodium.org/`



Assignment website: https://hash-browns.cs.uwaterloo.ca

The assignment website shows you your unofficial grade for the programming part; the grade becomes official once your final code submission has been examined. Your unofficial grade will update as you complete each question, so you will effectively know your grade by converting it to your final grade with our formula (see the top note frame of Programming Question) before the deadline. The assignment website also allows you to debug interactions between your code and the web API.

`libsodium` documentation

The official documentation for the `libsodium` C library is available at this website:

https://libsodium.org/doc/

You should primarily use the documentation available for the `libsodium` binding in your language of choice. However, even if you are not using C, it is occasionally useful to refer to the C documentation to get a better understanding of the high-level concepts, or when the documentation for your specific language is incomplete.

In the past, the website has been unavailable for extended periods of time. If the documentation site is down, you can access an offline mirror of the PDF documentation on LEARN.

Choosing a programming language

Since we will not be executing your code (although we will read it to verify your solution), you may theoretically choose any language that works on your computer.

You will need to use `libsodium` to complete the assignment. While `libsodium` is available for dozens of programming languages (check the list of language bindings[2] to find an interface for your language), you will need to limit your language choice as not all `libsodium` language bindings support all of the features needed for this assignment. Specifically, you should quickly check the `libsodium` documentation for your language to ensure that it gives you access to the following features:

- "Secret box": secret-key authenticated encryption using XSalsa20 and Poly1305 MAC
- "Box": public-key authenticated encryption using X25519, XSalsa20, and Poly1305 MAC
- "Signing": public-key digital signatures using Ed25519
- "Password hashing": key derivation function (KDF) using Scrypt with Salsa20/8 and SHA-256
- "Generic hashing": using BLAKE2b

You should also choose a language that makes the following tasks easy:

- Encoding and decoding `base64` strings
- Encoding and decoding hexadecimal strings
- Encoding and decoding JSON data
- Sending `POST` requests to websites using HTTPS

While you are not required to use a single language for all solutions, it is best to avoid the need to switch languages in the middle of the assignment.

You may use any third-party libraries and code. However, if you copy code from somewhere else, be sure to include prominent attribution with clear demarcations to avoid plagiarism.

We have specific advice for the following languages, which we have used for sample solutions:

- **Python**: This language works very well. Use the `nacl` module (`https://github.com/pyca/pynacl`) to wrap `libsodium`. The box and secret box implementations include nonces in the ciphertexts, so you do not need to manually concatenate them. The `base64`, `json`, and `requests` modules from the standard library work well for interacting with the web API.

[2] `https://download.libsodium.org/doc/bindings_for_other_languages/`

- **PHP**: This language works well if you are already familiar with it. Use the `libsodium` extension (https://github.com/jedisct1/libsodium-php) for cryptography. The `libsodium-php` extension is included in PHP 7.2. Otherwise, you may need to install the `php-dev` package and install the `libsodium-php` extension through `PECL`. In that case, you must manually include the compiled `sodium.so` extension in your CLI `php.ini`. Interacting with the web API is easy using global functions included in the standard library: `pack` and `unpack` for hexadecimal conversions, `base64_encode` and `base64_decode`, `json_encode` and `json_decode`, and either the `curl` module or HTTP context options for submitting HTTPS requests.
- **Java**: This language is a reasonable choice if you are comfortable using it. The `libsodium-jna` binding (https://github.com/muquit/libsodium-jna) contains all of the functions that you will need. Please follow the installation instructions listed in the website. The `java.net.HttpURLConnection` class works for submitting web requests. Base64 and hexadecimal encoding functions are available in `java.util`, and JSON encoding functions are available in `org.json`. Note that the ugsters may not contain updated packages for Java.
- **JavaScript**: The simplest way to solve the assignment in JavaScript is to use Node.JS with the `libsodium.js` binding (https://github.com/jedisct1/libsodium.js). Unfortunately, the method signatures in `libsodium.js` are slightly different than the C library, and the wrapper is poorly documented; you may need to look at the prototypes in `wrapper/symbols/` to identify the inputs and outputs. You should use the `dist/modules-sumo` package (not the default), since it includes the required hashing functions. `libsodium.js` includes helper functions for converting between hexadecimal and binary. The standard library includes the other encoding functions you will need: `atob` and `btoa` for `base64`, and the `JSON` object for JSON processing. Be wary of string encoding: you may need to use the `from_string` and `to_string` functions in certain situations.
- **C**: While C has the best `libsodium` documentation, all of the other tasks are more difficult than other languages. The assignment is also much more challenging if you use good C programming practices like error handling and cleaning up memory. If you choose C, you will spend a significant amount of time solving Question 1 before receiving any marks. We recommend `libcurl` (https://curl.haxx.se/libcurl/) for submitting API requests, `Jansson` (http://www.digip.org/jansson/) for processing JSON, and `libb64` (http://libb64.sourceforge.net/) for `base64` handling. Note that you will need to search for the proper usage of the `cencode.h` and `cdecode.h` headers for `base64` processing. You will need to provide your own code for hexadecimal conversions; it is acceptable to copy code from the web for this purpose, but be sure to attribute its author using a code comment.

You may use any other language, but then we cannot provide informed advice for language-specific problems. We also cannot guarantee that bindings for other languages contain all required features.

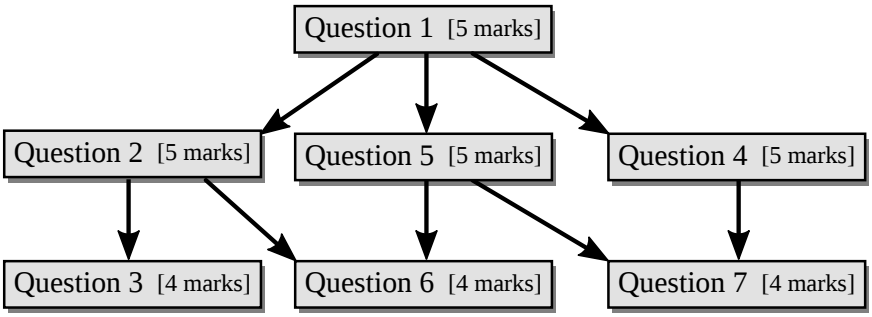
Ugster availability

Some of the aforementioned programming languages (C, Python, PHP) and libraries for libsodium will be made available on the Ugsters in case you do not have access to a personal development computer. Note that we do not have updated packages for Java or Javascript on the ugsters. We do support Ruby on the ugsters though we will not be able to provide informed advice for language-specific problems for Ruby.

Question Dependencies

Every question in the programming part takes place in an isolated environment; no data that you send or receive in one question will appear in another question. Some of the later questions use techniques that are introduced in earlier questions, so you may find that it is useful to copy and/or reuse code. However, when submitting your assignment, **you must submit code for all questions**, so do not overwrite your code for previous questions!

It is generally advisable to solve the questions in order. However, if you get stuck and want to move on to a later question, you should know that there are several "dependencies" between questions (i.e., solving some questions essentially involves solving previous ones as a component). The following graph indicates question dependencies:



Question 1: Using the API [5 marks]

In this first question, we will completely ignore cryptography and instead focus on getting your code to communicate with the server.

Begin by visiting the assignment website and logging in with your WatIAM credentials. You will be presented with an overview of your progress for the assignment. While you can simply use a web browser to view the assignment website, your code will need to communicate with the web

API. The web API does not use WatIAM for authentication. Instead, you will need an "API token" so that your code is associated with you.

Click on the "Show API Token" button on the assignment website to retrieve your API token. **Do not share your API token with anyone else**; if you suspect that someone else has access to your token, use the "Change API Token" button to generate a new one, and then inform the TA. Your code will need to use this API token to send and receive messages.

Question 1 part 1: send a message [3 marks]

Your first task is to send an unencrypted message to Nessie using the web API. To do this, submit a web request with the following information:

- URL: `https://hash-browns.cs.uwaterloo.ca/api/plain/send`
- HTTP request type: `POST`
- `Accept` header: `application/json`
- `Content-Type` header: `application/json`

For every question in this assignment, the request body should be a JSON object. The JSON object must always contain an `api_token` key with your API token in hexadecimal format.

To send a message to Nessie, your JSON object should also contain `to` and `message` keys. The `to` key specifies the username for the recipient of your message; this should be set to `nessie`. The `message` key specifies the message to send, encoded using `base64`.

You will receive marks for sending any non-empty message to Nessie (sadly, Nessie is a script that lacks the ability to understand the messages that you send). For example, to send a message containing "Hello, World!" to Nessie, your request would contain a request body similar to this:

```
{"api_token": "3158a1a33bbc...9bc9f76f",
"to": "nessie", "message": "SGVsbG8sIHdvcmxkIQ=="}
```

Consult the documentation for your programming language of choice to determine how to construct these requests.

The web API always returns JSON data in its response. If your request completed successfully, the response will have an HTTP status code of `200` and you will receive an empty object; check the assignment website to verify that you have been granted marks for completing the question. If an error occurs, the response will have an HTTP status code that is **not** `200`, and the JSON response will contain an `error` key in the object that describes the error.

If you are having difficulty determining why a request is failing, you can enable debugging on the assignment website. When debugging is enabled, all requests that you submit to the web API will be displayed on the assignment website, along with the details of any errors that occur. If debugging is enabled and you are not seeing requests in the debug log after running your code, then your code is not connecting to the web API correctly.

Question 1 part 2: receive a message [2 marks]

Next, you will use the web API to receive a message that Nessie has sent to you. To do this, submit a `POST` request to the following URL:

```
https://hash-browns.cs.uwaterloo.ca/api/plain/inbox
```

All requests to the web API are `POST` requests with the `Accept` and `Content-Type` headers set to `application/json`; only the URL and the request body changes between questions. The JSON object in the request body for your `inbox` request should contain only your `api_token`.

The response to your request is a JSON-encoded array with all of the messages that have been sent to you. Each array element is an object with `id`, `from`, and `message` keys. The `id` is a unique number that identifies the message. The `from` value is the username that sent the message to you. The `message` value contains the `base64`-encoded message.

Decode the message that Nessie sent you. The message should contain recognizable English words. **The messages from Nessie are meaningless and randomly generated.** We use English words so that it is obvious when your code is correct, but the words themselves are completely random.

To receive the marks for this part, go to the assignment website and open the "Question Data" page. This page contains question-specific values for the assignment, and also allows you to submit answers to certain questions. Enter the decoded message that Nessie sent to you in the "Question 1" section to receive your mark.

Question 2: Pre-shared Key Encryption [5 marks]

In this part, you will extend your code from question 1 to encrypt messages using secret-key encryption. For now, we will assume that you and Nessie have somehow securely shared a secret key at some point in the past. You will now exchange messages using that secret key.

Begin by importing an appropriate language binding for `libsodium`. Since every language uses slightly different notations for the `libsodium` functionality, you will need to consult the

documentation for your language to find the appropriate functions to call.

Question 2 part 1: send a message [3 marks]

Send a request to the following web API page:

`https://hash-browns.cs.uwaterloo.ca/api/psk/send`

Here, `psk` stands for "pre-shared key". The format of this `send` request is the same as in question 1 part 1, except that the `message` value that you include in the request body JSON will now be a ciphertext that is then `base64` encoded.

To encrypt your message, you should use the "secret box" functionality of `libsodium` to perform secret-key authenticated encryption. This type of encryption uses the secret key to encrypt the message using a stream cipher (XSalsa20), and to attach a message authentication code (Poly1305) for integrity. `libsodium` makes this process transparent; simply calling `crypto_secretbox_easy` (or the equivalent in non-C languages) will produce both the ciphertext and the MAC, which the library refers to as "combined mode".

You will need to generate a "nonce" ("number used once") in order to encrypt the message. The nonce should contain randomly generated bytes of the appropriate length. The size of the nonce is constant and included in most libsodium bindings (the `libsodium` documentation contains examples). To generate the message, you should `base64` encode a concatenation of the nonce followed by the output of `crypto_secretbox_easy`. Some language bindings will automatically do this for you, so check to see if the output of the function contains the nonce that you passed into it.

Abstractly, your request body should look something like this:

```
{"api_token": "3158a1a33bbc...9bc9f76f", "to": "nessie", "message":
  base64encode(concat(nonce, secretbox(plaintext, nonce, key)))}
```

To receive marks for this part, send an encrypted message to `nessie` using the secret key found in the "Question 2" section of the "Question Data" page. Note that the secret key is given in hexadecimal notation; you will need to decode it into a binary string of the appropriate length before passing it to the `libsodium` library.

Question 2 part 2: receive a message [2 marks]

Nessie has sent an encrypted message to you using the same format and key. Check your inbox by

requesting the following web API page in the usual manner:

`https://hash-browns.cs.uwaterloo.ca/api/psk/inbox`

You will need to decrypt this message by decoding the `base64` data, extracting the nonce (unless your language binding does this for you), and calling the equivalent of `crypto_secretbox_easy_open`. Enter the decrypted message in the "Question 2" section of the "Question Data" page to receive your marks. The decrypted message contains recognizable English words.

Question 3: Pre-shared Password Encryption [4 marks]

Securely sharing 32-byte keys is not very convenient. It is slightly more convenient to securely share passwords, but passwords themselves cannot be used for secret-key encryption. However, we can derive secret keys from reasonably secure passwords by using iterated hash functions, as discussed in class in the context of web applications.

Question 3 part 1: send a message [3 marks]

Using the exact same format as in question 2 part 1, send a message to Nessie using the following web API page:

`https://hash-browns.cs.uwaterloo.ca/api/psp/send`

Here, `psp` stands for "pre-shared password". The only thing to change is that, instead of passing in a secret key directly, you must instead iteratively hash a pre-shared password to derive the key.

Visit the "Question Data" page and retrieve the password and salt from the "Question 3" section. Use the Scrypt password hashing functionality of `libsodium` to derive the secret key from this password and salt. In the C library, the function that accomplishes this task is called `crypto_pwhash_scryptsalsa208sha256`. This function performs a large number of cryptographic hashes and memory-hard operations[3] to derive the secret key; this procedure greatly increases the amount of time required to guess the password by brute force.

The iterative hashing function also takes as input the number of computations to perform and the maximum amount of RAM to use. You should use the "`INTERACTIVE`" constants provided by `libsodium` to configure these values. If your language binding does not expose these constants

[3] These operations are intentionally designed to be difficult to perform on devices with small amounts of memory, such as custom password-cracking hardware.

(e.g., the `nacl` module for Python), then you will find the values to use in the "Question 3" section of the "Question Data" page.

Question 3 part 2: receive a message [1 mark]

Check your inbox using this web API page:

`https://hash-browns.cs.uwaterloo.ca/api/psp/inbox`

Derive the secret key from the password, decrypt the message, and enter the plaintext that Nessie sent you in the "Question 3" section of the "Question Data" page.

Question 4: Digital Signatures [5 marks]

In most common conversations, the communication partners do not have a pre-shared secret key. For this reason, public-key cryptography (also known as asymmetric cryptography) is very useful. The remaining questions focus on public-key cryptography.

In this question, you will send an unencrypted, but digitally signed, message to Nessie.

Question 4 part 1: upload a public verification key [2 marks]

The first step in public-key communications is key distribution. Everyone must generate a secret signature key and an associated public verification key. These verification keys must then be distributed somehow. For this assignment, the web API will act as a public verification key directory: everyone can upload a verification key, and request the verification keys that have been uploaded by other users.

`libsodium` implements public-key cryptography for digital signatures as part of its `sign` functions. Before sending a message to Nessie, you will need to generate a signature and verification key (together called a key pair). Generate this pair using the equivalent of the C function `crypto_sign_keypair` in your language. You should save the secret signature key somewhere (e.g., a file), because you will need it for the next part. To receive marks for this part, upload the verification key to the server by sending a `POST` request with the usual headers to the following web API page:

`https://hash-browns.cs.uwaterloo.ca/api/signed/set-key`

The request body should contain a JSON object with a `public_key` value containing the `base64` encoding of the verification key. For example, your request body might look like this:

```
{"api_token": "3158a1a33bbc...9bc9f76f",
 "public_key": "CazwYZnnnYqMI6...wTWk="}
```

Upon success, the server will return a `200` HTTP status code with an empty JSON object in the body. If you submit another `set-key` request, it will overwrite your existing verification key.

Question 4 part 2: send a message [3 marks]

Now that you have uploaded a verification key, others can use it to verify that signed messages really were authenticated by you (or someone else with your secret key). Send an unencrypted and signed message to Nessie by sending a request to the following web API page in the usual way:

`https://hash-browns.cs.uwaterloo.ca/api/signed/send`

The `message` value in your request body should contain the `base64` encoding of the plaintext and signature in "combined mode". In the C library, you can generate the "combined mode" signature using the `crypto_sign` function. Nessie will be able to verify the authenticity of your message using your previously uploaded verification key.

Question 5: Public-Key Authenticated Encryption [5 marks]

While authentication is an important security feature, the approach in question 4 does not provide confidentiality. Ideally, we would like both properties. `libsodium` supports authenticated public-key encryption, which allows you to encrypt a message using the recipient's public key, and authenticate the message using your secret key. Note, that for authenticated encryption the public key is used for both encryption and for verification while the secret key is used for both decryption and for signing.

The `libsodium` library refers to an authenticated public-key ciphertext as a "box" (in contrast to the "secret box" used in questions 2 and 3). Internally, `libsodium` performs a key exchange between your secret key and Nessie's public key to derive a shared secret key. This key is then used internally to encrypt the message with a stream cipher and authenticate it using a message authentication code.

Question 5 part 1: verify a public key [2 marks]

One of the weaknesses of public key directories like the one implemented by the web API in this assignment is that the server can lie. If Nessie uploads a public key and then you request it from the server, the server could send you its public key instead. If you then sent a message encrypted with that key, then the server would be able to decrypt it; it could even re-encrypt it under Nessie's actual public key, and then forward it along (acting as a "man in the middle").

To defend against these attacks, "end-to-end authentication" requires that you somehow verify that you received Nessie's actual public key. This is a very difficult problem to solve in a usable way, and is the subject of current academic research. One of the most basic approaches is to exchange a "fingerprint" of the real public keys through some other channel that an adversary is unlikely to control (e.g., on business cards, or through social media accounts).

For this part, you must download Nessie's public key from the web API, and then verify that you were given the correct one. Submit a `POST` request in the usual way to the following web API page:

`https://hash-browns.cs.uwaterloo.ca/api/pke/get-key`

Here, `pke` means "public-key encryption". Your request body should contain a JSON object with a `user` key containing the username associated with the public key you're requesting (in this case, `nessie`). The server's response will be a JSON object containing `user` (the requested username) and `public_key`, a `base64` encoding of the user's public encryption key.

To verify that you received the correct public key, you should derive a "fingerprint" by passing the key through a cryptographic hash function. Use the BLAKE2b hash function provided by `libsodium` for this purpose. The C library implements this as `crypto_generichash`, but other languages might name it differently. Do not use a key for this hash (it needs to be unkeyed so that everyone gets the same fingerprint). Remember to `base64` decode the public key before hashing it! The resulting hash is what you would compare to the one that Nessie securely gave to you. To get the marks for this part, enter the hash of the public encryption key, in hexadecimal encoding, into the "Question 5" section of the "Question Data" page. Keep in mind this hash/fingerprint is not the same as the public key itself.

Question 5 part 2: send a message [2 marks]

Before sending a message to Nessie, you will first need to generate and upload a public key. While the key pairs generated in question 4 were generated with the `sign` functions of `libsodium`, the key pairs for this question must be generated with the `box` functions. This difference is because the public encryption keys for this question will be used for authenticated encryption rather than digital signatures, and so different cryptography is involved.

Generate a public and secret key using the equivalent of the C function `crypto_box_keypair` in your language. Then, using the same request structure as in question 4 part 1, upload your public key to the following web API page:

`https://hash-browns.cs.uwaterloo.ca/api/pke/set-key`

Once you have successfully uploaded a key (indicated by a `200` HTTP status code and an empty JSON response), you can send a message to Nessie. Encrypt your message using the equivalent of the C function `crypto_box_easy` in your language. This function takes as input Nessie's public key (which you downloaded in the previous part, the value retrieved from a request to api/pke/get-key after it is base64-decoded), your secret key, and a nonce. The function outputs the combination of a ciphertext and a message authentication code.

You should generate the nonce randomly and prepend it to the start of the ciphertext, in the same way that you did for question 2 part 1. Encode the resulting data with `base64` encoding. Abstractly, your request body should look something like this:

```
{"api_token": "3158a1a33bbc...9bc9f76f", "to": "nessie",
                "message": base64encode(
concat(nonce, box(plaintext, nonce, nessie_public, your_secret))
                )}
```

Finally, send the message to Nessie in the usual way using the following web API page:

`https://hash-browns.cs.uwaterloo.ca/api/pke/send`

Question 5 part 3: receive a message [1 mark]

To receive the mark for this part, you will need to decrypt a message that Nessie has sent to you. Check your inbox in the usual way with a request to the following web API page:

`https://hash-browns.cs.uwaterloo.ca/api/pke/inbox`

To decrypt the message from Nessie, you will need your secret key and Nessie's public key. After `base64` decoding the message, decrypt it using the equivalent of the C function `crypto_box_open_easy` in your language. Provide the decrypted message in the "Question 5" section of the "Question Data" page.

Question 6: Government Surveillance [4 marks]

The government has decided that they must be able to decrypt all secure messages sent between you and Nessie through the web server. They have devised a new protocol that will protect the contents of your message from everyone except you, Nessie, and them. In the new protocol, you will use hybrid encryption: the message will be encrypted with secret-key encryption, and then the secret key will be encrypted using public-key encryption. Normally, you would encrypt the secret key using only Nessie's public encryption key. In this new protocol, you will also encrypt the secret key using the government's public encryption key. This way, both Nessie and the government will be able to use their secret key to decrypt the message.

Question 6 part 1: send a message [3 marks]

Begin by generating and uploading a public encryption key in the exact same way as for question 5 part 2, except using the following web API page:

```
https://hash-browns.cs.uwaterloo.ca/api/surveil/set-key
```

Next, download Nessie's public encryption key in the exact same way as for question 5 part 1, except using the following web API page:

```
https://hash-browns.cs.uwaterloo.ca/api/surveil/get-key
```

Visit the "Question Data" page and obtain the government's public encryption key (in `base64` encoding) from the "Question 6" section.

Now it is time to create your encrypted message for Nessie. Do this by following these steps using the appropriate functions for your language:

1. Generate a random key, called the message key, for "secret box" encryption.
2. Encrypt the plaintext with the message key using the same technique as question 2. The resulting ciphertext is called the message ciphertext. The nonce for this ciphertext is called the message nonce.
3. Encrypt the message key with Nessie's public encryption key and your secret key using the same technique as question 5. The resulting ciphertext is called the recipient ciphertext, and its nonce is called the recipient nonce.
4. Encrypt the message key with the government's public encryption key and your secret key using the same technique as question 5. The resulting ciphertext is called the government ciphertext, and its nonce is called the government nonce.

Now construct the message that you will send to the web API. The message should be the concatenation of these values, in order:

1. The recipient nonce [24 bytes]
2. The recipient ciphertext (including encrypted message key and a MAC) [16 bytes + 32 bytes]
3. The government nonce [24 bytes]
4. The government ciphertext (including encrypted message key and a MAC) [16 bytes + 32 bytes]
5. The message nonce [24 bytes]
6. The message ciphertext (including encrypted plaintext and a MAC) [16 bytes + variable length of the plaintext]

Send this message using `base64` encoding to Nessie in the usual way with a request to the following web API page:

`https://hash-browns.cs.uwaterloo.ca/api/surveil/send`

Question 6 part 2: receive a message [1 mark]

To receive the mark for this part, you will need to decrypt a message that Nessie has sent to you. Check your inbox in the usual way with a request to the following web API page:

`https://hash-browns.cs.uwaterloo.ca/api/surveil/inbox`

To decrypt the message from Nessie, use Nessie's public encryption key and your secret key to decrypt the ciphertext produced for you (i.e., the recipient ciphertext, not the government ciphertext). This will give you the message key. You can completely ignore the government ciphertext. Use the message key to decrypt the message ciphertext and recover the plaintext. Provide the decrypted plaintext in the "Question 6" section of the "Question Data" page.

Question 7: Forward Secrecy [4 marks]

The protocols in all of the previous questions share a problem: if an eavesdropper passively records all of the encrypted messages and later steals one of the secret keys, then they can retroactively decrypt any messages that they previously stored. If we stop this from happening, then we achieve forward secrecy (sometimes called perfect forward secrecy).

The "trick" for forward secrecy is to encrypt messages using temporary secret keys and authenticate

them using long-term secret signature keys. Stealing long-term secret signature keys that are only used for authentication does not affect previous conversations, since they have already concluded. It is also generally more difficult to steal secret keys that are erased quickly than it is to steal secret keys that must be kept around for a long time.

The protocol for this question shares aspects of the signed messages protocol in question 4, and the public-key encryption in question 5. However, in this question, every user will upload two public keys to the server: a public identity verification key, and a signed prekey.[4] The identity verification key is used for authentication via digital signatures. It is produced in the same way as in question 4. The signed prekey is used for encryption. It is produced in the same way as in question 5, with the exception that it is also signed by the identity verification key.

Unlike identity verification keys, signed prekeys are meant to be changed regularly. Once both the sender and receiver of a message have deleted their old signed prekeys, the message cannot be retroactively decrypted by stealing secret keys.

When sending a message to Nessie in question 5, the "box" was created using Nessie's public encryption key and your secret signature key, both of which are long-lived. Here, the "box" will be created using Nessie's signed prekey and the secret decryption key for your signed prekey.

Question 7 part 1: upload a signed prekey [1 mark]

In this first part, you will generate and upload a signed prekey so that others can send messages to you.

The first step is to generate and upload an identity verification key in the same manner as question 4 part 1. Produce a signature and verification key using the equivalent of the C function `crypto_sign_keypair` in your language. Upload the verification key in the usual way (encoded in `base64` within the `public_key` property of a JSON object in the `POST` request) through the following web API page:

```
https://hash-browns.cs.uwaterloo.ca/api/prekey/set-identity-key
```

If the upload was successful, you'll receive a `200` HTTP status code in response.

Next, generate a prekey using the same technique as question 5 part 2. Since prekeys will be used for public-key encryption, you should generate your prekey using the equivalent of the C function `crypto_box_keypair` in your language.

[4] A "prekey" is just an ordinary public encryption key with a short lifetime. This terminology was introduced by the secure messaging application called Signal.

To produce a signed prekey, use your identity signature key to sign the public encryption key in the same way as in question 4 part 2. Use the equivalent of the C function `crypto_sign` in "combined mode" to produce the signed prekey. Finally, `base64` encode the signed prekey and send it in the `public_key` property of a JSON object to the following web API page:

`https://hash-browns.cs.uwaterloo.ca/api/prekey/set-signed-prekey`

If you receive a `200` HTTP status code in response, then you have received the mark for this part.

Question 7 part 2: send a message [2 marks]

Now that you have uploaded a signed prekey, you are ready to send a message to Nessie. Download Nessie's identity verification key sending a request containing a JSON object with `nessie` in the `user` key to this web API page:

`https://hash-browns.cs.uwaterloo.ca/api/prekey/get-identity-key`

Your request body should look similar to this:

```
{"api_token": "3158a1a33bbc...9bc9f76f", "user": "nessie"}
```

You should receive the `base64`-encoded identity verification key in the `public_key` key of a JSON object in the response.

Using the exact same technique, request Nessie's signed prekey by sending a request to this web API page:

`https://hash-browns.cs.uwaterloo.ca/api/prekey/get-signed-prekey`

This time, you should receive Nessie's signed prekey, encoded using `base64`, in the `public_key` property of the JSON object. You should now verify the signature on this prekey, and extract the public encryption key, by using the equivalent of the C function `crypto_sign_open` in your language. If the signature is valid, this function will return the public key to use for encryption.

Finally, you can encrypt a message to send to Nessie. Encrypt the plaintext using Nessie's prekey as the public encryption key, and secret signature key associated with your prekey that you generated in part 1. Use these keys for public-key authentication using the equivalent of the C function `crypto_box_easy` in your language. This is the same public-key authenticated encryption technique that you used in question 5 part 2. As before, you should include a newly generated random nonce in your message.

Send the nonce and the ciphertext to Nessie by sending a request in the usual manner to the

following web API page:

`https://hash-browns.cs.uwaterloo.ca/api/prekey/send`

Abstractly, your request body should look something like this:

```
{"api_token": "3158a1a33bbc...9bc9f76f", "to": "nessie",
                    "message": base64encode(
concat(nonce, box(plaintext, nonce, nessie_prekey, your_prekey_secret))
                    )}
```

If Nessie is able to successfully decrypt your message, you will receive a `200` HTTP status code in the response indicating that you have been awarded the marks for this part.

Question 7 part 3: receive a message [1 mark]

Finally, you will need to receive a message sent to you by Nessie using the same encryption scheme as the previous part. Check your inbox in the usual manner using the following web API page:

`https://hash-browns.cs.uwaterloo.ca/api/prekey/inbox`

When you receive a message, you should look up the identity verification key and signed prekey of the sender (if you have not done so already). You can do this by submitting requests to `https://hash-browns.cs.uwaterloo.ca/api/prekey/get-identity-key` and `https://hash-browns.cs.uwaterloo.ca/api/prekey/get-signed-prekey` as described in part 2. Using Nessie's prekey and the secret signature key associated with your prekey, decrypt the ciphertext using the equivalent of the C function `crypto_box_open_easy` in your language. This is the same decryption function that was used in question 5 part 3.

Once you have decrypted the message sent to you by Nessie, enter the message in the "Question 7" section of the "Question Data" page to receive the mark for this part.

Freedom Environment

If you successfully complete every question in the programming part, then you will be given access to the "freedom" environment. Here, you can use the code that you wrote for question 7 to communicate with other students who have also completed all of the parts, assuming that you know their WatIAM username and they have uploaded keys in the environment. To communicate, use the following web API pages:

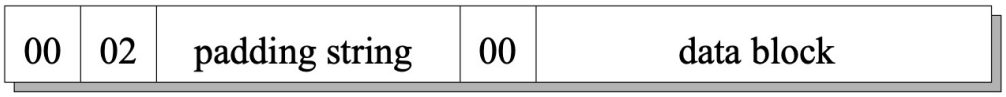
```
https://hash-browns.cs.uwaterloo.ca/api/freedom/get-identity-key
https://hash-browns.cs.uwaterloo.ca/api/freedom/set-identity-key
https://hash-browns.cs.uwaterloo.ca/api/freedom/get-signed-prekey
https://hash-browns.cs.uwaterloo.ca/api/freedom/set-signed-prekey
      https://hash-browns.cs.uwaterloo.ca/api/freedom/send
     https://hash-browns.cs.uwaterloo.ca/api/freedom/inbox
https://hash-browns.cs.uwaterloo.ca/api/freedom/message_id/delete
```

To delete messages from your inbox, send a `POST` request in the usual manner to the `delete` web API page listed above. The `message_id` in the URL is the `id` value of the message provided in the response to an `inbox` request. A `200` HTTP status code in the response indicates that the message has been removed from your inbox.

Assignment 2: Bleichenbacher's Attack [15 marks]

In this assignment you will implement the Bleichenbacher's oracle attack on PKCS v1.5 RSA. PKCS v1.5 is a Request For Proposal (RFC) that defines a RSA encryption standard. You can find more information about the RFC here.

In RSA crypto system, all messages are represented as integers. Given a public key (e, n) the message x is encrypted as $c = x^e \mod n$. Therefore, each message should fit within the bit-length of the modulus n. However, not all messages are exactly as long as the modulus. PKCS v1.5 defines a padding standard to solve this issue. In PKCS v1.5 every message (a sequence of bytes) is padded as follows:



The padding consists of a leading $0x00$ byte, a $0x02$ byte, a sequence of **non-zero** random bytes called the padding string, a $0x00$ byte indicating the start of the data block followed by the data itself. The length of the padding string is adjusted by the length of the data to be sent but is at least 8 bytes. In other words:

$$|padding\ string| = \min\ (key\ length - |data\ block| - 3, 8)$$

In his infamous paper, Bleichenbacher noticed that this padding can be used as an oracle to decrypt RSA messages without factoring the modulus or a key compromise. The attack uses the well-known fact that RSA is homomorphic with respect to multiplication. Given a cipher-text c corresponding to a message x, an attacker can construct $c' = a^e * c = a^e * x^e = (ax)^e \mod n$ which is an encryption of ax for arbitrary a.

With this knowledge, read **section 3.1** of Bleichenbacher's paper. It doesn't require much math beyond high school algebra to understand how it works. For this assignment, you only need to implement Step 2, 3 and 4 of the attack. You won't need "Binding" in Step 1 as the message is bound for you.

You are given 2 folders in `PA2.zip` which you can find on Learn:

- `Server` contains a python file `server.py`. Follow the instructions at the end of this assignment to install the required dependencies for running the server locally. It is important that your code works against this server as your assignment will be graded by running your code against it.

- `Test` contains 2 subfolders `t1` and `t2`. In each of these folders you will find 4 text files:
  1. `encryption_key.txt` contains a `base64` encoded RSA encryption key. The encryption key is 65537 following RSA conventions but this file is needed for grading.
  2. `decryption_key.txt` contains a `base64` encoded RSA decryption key.
  3. `modulus.txt` contains a `base64` encoded RSA modulus corresponding to the encryption and decryption keys in the folder.
  4. `cipher.txt` contains a `base64` encoded RSA cipher text encrypted with the encryption key and modulus above.

We are using 1024 bit RSA for all of our tests. But you probably don't need to rely on this fact.

Run the server with the following command:

```
python3 server.py -d [path to decryption_key.txt] -n [path to modulus.txt]
```

This brings up a HTTP server at `127.0.0.1:8080` that you will use as your oracle for the attack. The server accepts HTTP POST requests as the previous assignment. Your request body should contain a JSON object of the form:

{ "message" : base64encode (cipher) }

Upon receiving your request, the server will try to decrypt the cipher-text and send back a response with HTTP status code 200 and a JSON object of the form:

{ "message" : base64encode (True or False) }

If the decrypted plain-text is PKCS v1.5 compliant the message is `True` o.w. it will be `False`.

If at anytime during processing your request the server fails at decryption (for example due to bad encoding or bad JSON formatting), it will send back a response with HTTP status code 400 and a JSON object informing you of what might have gone wrong. These JSON objects are similar to the above except for different message contents:

{ "message" : base64encode (Error Message) }

[10 marks] You have to write your solutions such that both Tests `t1` and `t2` succeed with finding the plain texts. You can check whether you have the correct plain text by decrypting the cipher text with the decryption key given to you. However, for grading, we will use **new RSA keys and cipher texts**. If your code implements a successful Bleichenbacher attack on the tests, it should succeed against new tests. However, you won't get any marks if you don't pass our tests even if your code works with `t1` and `t2`. You can do additional testing locally by generating random data and keys and check if your attack is successful.

Your code must support a simple command to run any new cipher text, encryption and modulus files. Upon running the command, your code must only print the plain text and the number of times it has queried the server, each in a new line. For example if you use python as your language, We should be able to run it as:

```
python3 bleichenbacher.py -c [path to cipher.txt] -e [path to encryption_key.txt] -n [path to modulus.txt]
```

and the output must look like:

```
This is the plain text
34235
```

You are free to choose any language that you feel comfortable with for this assignment. You can use different cryptography libraries as long as they support plain RSA encryption. In the `Server` folder, there are some utility functions in `simple_RSA.py` that you can copy if you are using python. This will also give you an idea of how to structure your code generally.

[5 marks] Finally, you have to upload a file **bleichenbacher.pdf** that describes your attack along with the necessary steps to install the dependencies of your code in the chosen language on a UNIX system (similar to the instructions to run the server below). We will follow those instructions before running your code against the server.

How to run the Server

- You need to install `python3.8` on your system.
  - On Mac run: `brew install python3.8`
  - On Linux (Ubuntu) run: `sudo apt-get install python3.8`
- Check you have the right version. Run: `python3 -V`.
- Install the `virtualenv` module. Run: `pip3 install virtualenv`
- Create a new virtualenv. Run: `virtualenv venv`
- Activate your virtualenv. Run: `source venv/bin/activate`
- Fortunately, you don't need to install any modules as all the necessary modules are python built-ins. Run the server with the command given before. Check if your server is running by making POST requests to it or by visiting `127.0.0.1:8080` in your browser. If you see a **"You're all set!"** message, the server is running.
- To deactivate the virtualenv, run: `deactivate`