UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 **Computer Security and Privacy** **Spring 2019**

Ian Goldberg
Navid Esfahani

ASSIGNMENT 3

Assignment due date: **Tuesday, July 30th, 2019 at 3:00 pm**

Total Marks: 62

Written Response TA: Dimcho Karakashev
Office hours: ~~Mondays 11:00 am–12:00 pm~~ Tuesdays 4:00 pm–5:00 pm, Jul 4–30, DC 3333

Programming TA: Miti Mazmudar
Office hours: ~~Mondays 1:00 pm–2:00 pm~~ Tuesdays 11:00 am–12:00 pm, Jul 4–30, DC 3333

TAs' office hours were updated 4 Jul 11:00 am so as not to conflict with class times.

Please use Piazza for all communication. Ask a private question if necessary. The TAs' office hours are also posted to Piazza for reference. You are expected to follow the expected Academic Integrity requirements for Assignments; you can find them here: https://uwaterloo.ca/library/get-assignment-and-research-help/academic-integrity/academic-integrity-tutorial. Strict penalties will be enforced on students for any Academic Integrity violations.

Written Response Questions [30 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.

1. [8 marks] **Two-time Pad**
   Two ciphertexts, called `ciphertext1` and `ciphertext2`, have been provided for you through infodist. The original plaintexts are fragments of text taken from the English-language Wikipedia. These fragments have had references (notations like "[4]") removed, and contain ASCII characters only in the range from 0x20 (space) to 0x7e (tilde). In particular, there are no control characters, such as newlines, in the plaintexts. The plaintexts were then truncated to exactly 300 bytes.
   To produce the ciphertexts, a random 300-byte pad was generated, and each plaintext was encrypted *using the same pad* by XORing the plaintext and the pad.
   The plaintexts, ciphertexts, and pad are unique to you (your classmates have been given different ciphertexts generated from different plaintexts using different pads).
   (a) [2 marks] What is the XOR of the two original plaintexts? Submit it as a 300-byte file called `xor`.
   (b) [6 marks] Determine the two original plaintexts. (Tips: "man ascii". You may search Wikipedia.) Submit them as two 300-byte files called `plaintext1` and `plaintext2`. Explain in detail how you got your answer. If you used or wrote any software to help you, describe how the software works.
2. [9 marks] **(Textbook) RSA**
   In first year you were taught how basic RSA public-key encryption works (MATH 135). This simplified version of RSA is referred to by cryptographers as "textbook RSA" or "naive RSA". This question investigates the practical (in)security of using this simple version. Recall that the public key is (n, e) where $n = pq$ for large primes p and q, and e is an integer. The private key is (p, q, d) where d is such that $de \equiv 1 \bmod (p-1)(q-1)$. The encryption of $m \in Z_n$ is $c = m^e \bmod n$, and decryption of the ciphertext c is $c^d \bmod n$. For more review, see Section 8.2 of the *Handbook of Applied Cryptography*, which is online: `http://www.cacr.math.uwaterloo.ca/hac/about/chap8.pdf`. For all of the following questions, assume n is too large to be factored in any reasonable amount of time. All questions will assume that (n, e) is Bob's public key, and that the other parties have an authentic copy of it.
   (a) [2 marks] Suppose that Alice sends $c = m^e \bmod n$ to Bob. Eve observes c, and suspects that m is one of $m_1 =$ `START PRODUCTION`, $m_2 =$ `HALT PRODUCTION` or $m_3 =$ `SELL INVENTORY`. Eve's suspicion is based on auxiliary information she has gathered about Alice and Bob, for instance she knows that Bob is Alice's warehouse manager. How can Eve confirm her guess, using only m_1, m_2, m_3, c and (n, e)?
   (b) [2 marks] Now, to counter the above attack, suppose Alice concatenates a random three-digit number between 000 and 999 before encrypting the message. Thus, Alice now sends $c' =$

$(m||r)^e \bmod n$, where the notation $m||r$ denotes the concatenation (as strings) of the values (numbers or strings) m and r. Describe how Eve can still check whether c' is an encryption of m_1, m_2 or m_3.

(c) [3 marks] Using the method you described in part (b), determine whether the given c' is an encryption of m_1, m_2 or m_3. The RSA public key is as follows:

n=28771083164278873253034411689186063378672971584656311337919722503551581159882315776087423908190059 83

and

$$e = 876970893102603883117458822861692717542922422215687.$$

The encrypted message is

c'=28739280528349179125431194948608507291687262557562857084252218658270542303835527598635798263774320 0.

We must also specify how the plaintext is converted to a decimal number. First, the 26 letters `A`, `B`, ... `Z` are represented by two-digit strings 01, 02, ... 26, respectively, and a "space" is represented by 00. Finally, three decimal digits are appended onto the end of the string as the random value r. For example, the plaintext message `I AM` with randomness $r = 78$ would be converted to the decimal number 09000113078 before applying RSA encryption. Note: In Maple, use the command "`&^`" for modular exponentiation. For example, in Maple:

```
> 143 &^ 15 mod 157;
118
```

You may of course use methods other than Maple to compute modular exponentiations, if you wish.
Determine which of m_1, m_2, or m_3 was encrypted to yield c'. Explain how you found your answer.

(d) [2 marks] Alice is a server operator who uses the following challenge-response protocol based on textbook RSA to authenticate users to Alice's website.

i. Alice encrypts a random number r to the user, say Bob, using Bob's public key.

ii. Bob decrypts the ciphertext from Alice to obtain the plaintext, say r_1.

iii. Bob returns the plaintext r_1 to Alice.

iv. Alice checks that $r_1 \stackrel{?}{=} r$. If this check is not successful, then Bob could not decrypt the message and thus does not have the right private key.

The challenge-response protocol is also shown in Figure 1.
Mallory is an evil server operator who also follows the same challenge-response protocol to "authenticate" Bob on Mallory's website. Suppose Mallory has observed the ciphertext $c = r^e \bmod n$ sent from Alice to Bob and wants to learn r by running the challenge-response protocol with Bob. Bob will abort the protocol if Mallory tries to use the same c (that would be too easy). How can Mallory learn r, using only c, e, n and random value(s) from Z_n? Specify the challenge Mallory should create and explain why it works.

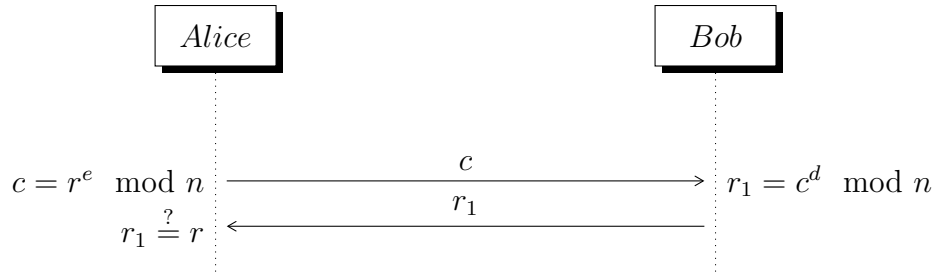


Figure 1: Challenge-response protocol

3. [5 marks] **Inference Attacks**
A bidding platform has a table of size N, `Bids`, in its database. Below is an excerpt (not the entire table):

Name	Birthday	Bid
Miguel	02/21	1,000
Matthew	03/15	3,451
Rachel	02/25	1,236
Nicholas	09/02	1,657
Stacey	03/02	2,765
Cori	09/02	3,444
James	03/23	2,786
Natalie	02/16	2,931
Christine	09/11	1,239

Table 1: Part of `Bids` Table

To deter people learning other people's bids, the database is set up to suppress the `Bid` field in the output of queries. However, users can execute queries of the form `SELECT SUM(Bid) FROM Bids 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) [3 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 Miguel's bid. Both the tracker and the three queries need to be of the form `SELECT SUM(Bid) FROM Bids ...`. Assume that people's names are unique and not known to the attacker (apart from Miguel's) and that the attacker has no additional information about Miguel (not even his birthday). 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 1) your assumption, 2) your tracker, and 3) the set of three queries.

Name	Gender	Age	Medical Item
Austin	Male	42	insulin pump
Chloe	Female	50	thermometer
Nicholas	Male	48	blood pressure monitor
Tony	Male	64	blood pressure monitor
Christine	Female	56	thermometer
Matthew	Male	59	insulin pump
Natalie	Female	49	insulin pump
Aidan	Male	43	thermometer
Henry	Male	62	insulin pump

Table 2: User data to be sold

(b) [2 marks] The bidding platform decides to sell user data to a vendor who is interested in entering the medical business. Medical items are not popular on the platform, so Table 2 represents all of the data to be sold.

To protect the privacy of the users, the bidding platfrom decides to modify the table such that it is 3-anonymous. The bidding platform declares that name, gender, and age fields are indentifiers. The constraints on the data to be anynomized are the following:

- Name is masked.
- Gender cannot be masked/modified.
- Age can only be generelized to at least 3 **equally sized** ranges.

As a result of the constraints, the bidding platform chooses the following three age ranges: [40-49], [50-59], [60-69]. Table 3 shows the anonymized table.

Is the anonymized table 3-anonymous? If yes, please explain and give the value of ℓ for which the table is ℓ-diverse. Otherwise, please present a correct solution and give the value of ℓ for which your table is ℓ-diverse.

Name	Gender	Age	Medical Item
*	Male	[40-49]	insulin pump
*	Female	[50-59]	thermometer
*	Male	[40-49]	blood pressure monitor
*	Male	[60-69]	blood pressure monitor
*	Female	[50-59]	thermometer
*	Male	[50-59]	insulin pump
*	Female	[40-49]	insulin pump
*	Male	[40-49]	thermometer
*	Male	[60-69]	insulin pump

Table 3: Anonymized user data

$$\begin{bmatrix}0 & 0 & 1 & 0 & 0\end{bmatrix} \begin{bmatrix}100 & 20 & 4\\ 200 & 30 & 3\\ 50 & 10 & 5\\ 70 & 15 & 3\\ 250 & 25 & 4\end{bmatrix} = \begin{bmatrix}50 & 10 & 5\end{bmatrix}$$

Figure 2: Example query vector ($\vec{q}$), inventory matrix (M) and response row ($\vec{r}$) for the original scheme. The user wants to retrieve row 3 in this example and so the query vector only includes a 1 on the third index of the query row vector.

4. [6 marks] **Private Information Retrieval (PIR)**
   Striker is a privacy-conscious individual who is fed up of targeted advertising and hyper-specific recommendations. DON.MAC is a *seemingly* privacy-friendly shopping company who wishes to cater to individuals like Striker. DON.MAC's inventory of items for sale is presented in the form of a matrix. Each row of the matrix contains information to further specify the product, such as its price, its weight and so on, and the row numbers correspond to the product ID.

   The original scheme for a user to retrieve information about a product, given by row c of the matrix, was as follows: the user sends the row number c to DON.MAC. DON.MAC returns row c of the matrix M to the user. Maliciously, DON.MAC would then log the number c that each user sent and build up a detailed profile of their searching history on DON.MAC's website.

   Note that sending a row number c to DON.MAC is mathematically equivalent to left-multiplying the matrix with a unit vector with a 1 in index c. (That is, a vector with 0s as its elements in all indices other than index c, which contains the element 1.) This vector is known as the query vector. So, from here on, we describe the original scheme as follows: the user sends a query vector $\vec{q}$ and DON.MAC computes $\vec{q} \cdot M$. That is, DON.MAC multiplies the query vector with the inventory matrix. DON.MAC then sends the resulting row, say $\vec{r}$, to the user. (The matrix M and the original query vector $\vec{q}$ for a specific query that Striker made are shown in Figure 2.)

   Privacy-conscious individuals like Striker have called for a revolution: companies like DON.MAC should be prevented from learning what the user queries, at *all* costs.

   (a) [2 marks] Briefly describe a *trivial* solution that would fulfill the revolution's goals. In other words, Striker should obtain the product $\vec{q} \cdot M$ such that DON.MAC does not learn $\vec{q}$. This solution may have a significant communication complexity and should not involve any cryptography.

   (b) [4 marks] DON.MAC has to spend a lot of money in internet bills to sustain the above trivial scheme. To solve this, DON.MAC convinces Striker to use a special public-key encryption

scheme, where the encryption function Enc_K under public key K has the following property:[1]

$$Enc_K(a + b) = Enc_K(a) + Enc_K(b) \tag{1}$$

where a and b are the plaintext integers. For the purposes of this question, assume the ciphertexts are also integers (in reality, the ciphertexts are a little more complicated). For $b = a$ we get:

$$Enc_K(a + a) = Enc_K(a) + Enc_K(a) \tag{2}$$

Note that the encryption scheme is *not* a deterministic one, but rather, *probabilistic*. In other words, two encryptions of the same plaintext under the *same* key may result in different ciphertexts. Therefore, the two ciphertexts on the right-hand side of equation 1 may not be the same. In that case, we obtain the following form of equation 2:

$$Enc_K(t \times a) = \sum_{i=1}^{t} Enc_K(a) \tag{3}$$

Otherwise, that is, when we have t identical encryptions of a plaintext a or a given encryption of a plaintext a multiplied by t, then we obtain the following:

$$Enc_K(t \times a) = t \times Enc_K(a) \tag{4}$$

The new scheme is as follows:

i. Striker generates a public-private encryption keypair, where K is the public encryption key and $\hat{K}$ is the private decryption key.

ii. Say that Striker wants to lookup index c in the matrix M. Striker constructs the *plaintext* query vector $\vec{q}$ to be a unit vector. That is, only the element q_c is 1 and the rest are 0.

iii. Striker encrypts the query vector with the encryption function Enc_K. That is, Striker encrypts each element of the query vector $\vec{q}$ under key K using the above function.

iv. Striker sends the ciphertext query vector, given by $[Enc_K(q_i)]_{1\times m}$, to DON.MAC.

v. As usual, upon obtaining the ciphertext query vector, DON.MAC proceeds to multiply it with the matrix to obtain the response row.

vi. DON.MAC sends the response row back to Striker.

vii. Striker has access to the private key $\hat{K}$ and a decryption function $Dec_{\hat{K}}$ that functions as usual. That is, $Dec_{\hat{K}}(Enc_K(x)) = x$. (Note that DON.MAC never obtains the key $\hat{K}$ during this protocol.) Striker thus decrypts row $\vec{r}$. That is, Striker computes $[Dec_{\hat{K}}(r_j)]_{1\times m}$.

Thus, now we have:

- M is an m-by-n matrix of integers, shown as $[M_{ij}]_{m\times n}$
- $[q_i]_{1\times m}$ and $[Enc_K(q_i)]_{1\times m}$ are 1-by-m rows

[1] An encryption scheme that satisfies this property is known as *homomorphic* encryption and the scheme in part b) is known as a computational PIR (or CPIR) scheme.

- $\vec{r}$ has the dimensions of a row of the matrix M and is thus 1-by-n. In compact notation, it is shown as $[r_j]_{1\times n}$ and it is the following product:

$$\vec{r} = \begin{bmatrix} Enc_K(q_1) & Enc_K(q_2) & \cdots & Enc_K(q_m) \end{bmatrix} \cdot \begin{bmatrix} M_{11} & \cdots & M_{1n} \\ \vdots & \ddots & \vdots \\ M_{m1} & \cdots & M_{mn} \end{bmatrix} \tag{5}$$

Question: Striker looks up row c in the matrix M using the above scheme. Using equations 1, 3, 4 and 5, show that Striker can reconstruct row c of the matrix M by decrypting the response row $\vec{r}$ that DON.MAC provides (that is, in step *vi* of the new protocol). Note that the plaintext query vector $\vec{q}$ that Striker constructs is a unit vector, as shown in the example in Figure 2. Your answer must include a *clear* explanation for all mathematical formulae in your answer.

5. [2 marks] **Government surveillance**
   A government may[2] force its citizens to encrypt any communication to itself along with the intended recipient, effectively eavesdropping on all communication that any citizens engage in. An example of such a scheme, which uses hybrid encryption, is described in the programming part in question 6. Megaluminous is a government agent who wishes to convince you two things:
   - The symmetric keys for each of the citizens are *only* made available to the police force of the country.
   - You should fear nothing from such a scheme unless you have something to hide. *But,* if you have something to hide, then you must be doing something illegal and thus must be punished for it (and so our system works!).

   You are to convince Megaluminous that both of their arguments are flawed.
   (a) [1 mark] Briefly describe why it is difficult to achieve secrecy of the symmetric decryption keys for each of the citizens throughout the country.
   (b) [1 mark] Briefly explain why a citizen may reasonably want to hide the content of their communication from the government. You may discuss *threats* to the individual or to the society at large.

[2] https://www.politico.com/story/2019/06/27/trump-officials-weigh-encryption-crackdown-1385306

Programming Question [32 marks]

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

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`[3] cryptography library in a language of your choice to send a message through a web API to your account on the assignment website. You will send messages to and receive messages from a fake user named *Jessie* in order to confirm that your code is correct. However, if you complete all of the programming part questions, you will be able to use your code to send secure messages to other students who have also completed all questions (or to the TA).

Assignment website: https://whoomp.cs.uwaterloo.ca/458a3

The assignment website shows you your unofficial grade for the programming part, which will update as you complete each question. So, you will effectively know your grade *before* the deadline. The grade becomes official once your final code submission has been examined. 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/

[3] https://libsodium.org/

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[4] 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,

[4] https://download.libsodium.org/doc/bindings_for_other_languages/

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.
- **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, but getting `libsodium` to work can be tricky. The `libsodium-jni` binding (https://github.com/joshjdevl/libsodium-jni) works, but it is incomplete. If you choose to use Java, check LEARN for modified bindings containing all of the functions that you will need. 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.

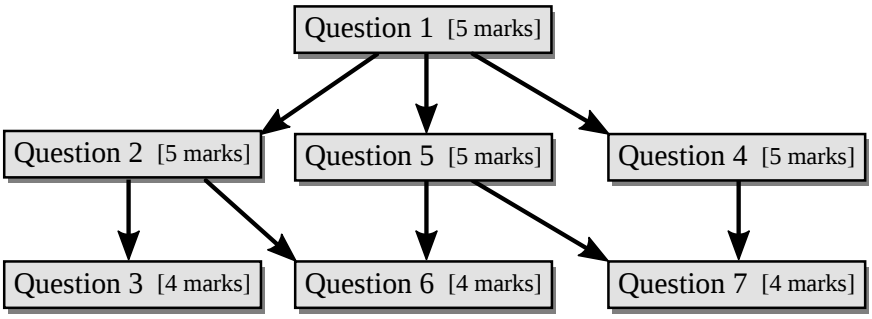
Ugster availability

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:



Questions

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.

Web requests: To send a message to Jessie or to upload content to the web server through the web API, you must send an HTTP `POST` request to the given URL for each question. *All* requests must contain the following **headers**:

- `Accept` header: `application/json`
- `Content-Type` header: `application/json`

The request *body* of any messages that you send to the web API must 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 Jessie, 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 `jessie`. The `message` key specifies the message to send, encoded using `base64`. For questions that require you to send requests to Jessie, you will receive marks for sending any non-empty message to Jessie.[5]

For example, to send a message containing "Hello, World!" to Jessie, your request would contain a request body similar to this:

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

[5] Sadly, Jessie is a script that lacks the ability to understand the messages that you send.

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

Web API Responses: If your request completed successfully, the response will have an HTTP status code of `200`. For requests to URLs that correspond to *sending* messages or uploading content, you will receive an empty object on a successful request. Check the assignment website to verify that you have been granted marks for completing the question. For requests to URLs that correspond to *receiving* messages, the response to your request will be 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. You *must* first perform `base64` decoding on the message that Jessie sent you to get an array of bytes. Following that, for question 2 onwards, you may decrypt the decoded message. Note that the message should contain recognizable English words. **The messages from Jessie 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.

For questions that ask you to send requests for downloading content, the question also contains a specification of the expected format of the JSON object in the response. For all types of requests, 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.

Question data: Go to the assignment website and open the "Question Data" page. This page contains question-specific values for the assignment. Additionally, for questions that ask you to receive messages from Jessie, you will need to submit the plaintext message that you received in a field on this page for that question.

Debugging: 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 1: send a message [3 marks]

Your first task is to send an unencrypted message to Jessie using the web API. To do this, submit a `POST` request, with the headers and request body as mentioned above, to the following URL:

`https://whoomp.cs.uwaterloo.ca/458a3/api/plain/send`

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

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

`https://whoomp.cs.uwaterloo.ca/458a3/api/plain/inbox`

Note that the JSON object in the request body for this request should contain only your `api_token`.

Submitting received message: After performing base64-decoding on the message that Jessie sent to you, enter the decoded plaintext message, which should contain English words, in the "Question 1" section on the "Question Data" pageto 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 Jessie have somehow securely shared a secret key at some point in the past. This secret key is found in the "Question 2" section of the "Question Data" page. You will now exchange messages using that secret key, which is known as the *pre-shared key*. (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.)

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]

For this question, you will first generate a secret-key encryption of a message using the `libsodium` library as follows. You should use the secret-key found in the "Question 2" section of the "Question Data" page to encrypt your message.

1. You will need a "nonce" ("number used once") in order to encrypt the message. The nonce should contain randomly generated bytes of the appropriate length. The `libsodium` documentation contains examples to generate a nonce. Some language bindings will allow calling the encryption function, mentioned below, with just the secret key and may internally generate a nonce for you.

2. 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 and the nonce 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".
3. For Jessie to be able to correctly decrypt and authenticate your message, you will need to include the nonce along with the ciphertext and the MAC. Some language bindings will automatically include the nonce with the ciphertext and the MAC, so check to see if the output of the function contains the nonce that you passed into it. Otherwise, you should *pre-pend* the nonce to the output of `crypto_secretbox_easy`.
4. Finally, you should `base64` encode the above concatenation (of the nonce with the output of `crypto_secretbox_easy`).

Abstractly, your request body should look something like this:

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

Next, send a `POST` request to the following web API page, with identical headers and body as in question 1 part 1 *except* that the key of the `message` value in the request body will now be the above `base64`-encoded message.

`https://whoomp.cs.uwaterloo.ca/458a3/api/psk/send`

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

Jessie 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 same manner as in question 1 part 2:

`https://whoomp.cs.uwaterloo.ca/458a3/api/psk/inbox`

Process the ciphertext message as follows and then enter the plaintext English words in the "Question 2" section of the "Question Data" page to receive your marks.

1. First decode the `base64` data to obtain the nonce and the ciphertext as a binary string.
2. You can now perform secret-key decryption on the above binary string using the equivalent of the `crypto_secretbox_easy_open` function in your language to obtain the plaintext message. Some language bindings may accept the entire decoded binary string as a single

input, whereas others may accept the nonce and the $(ciphertext \| MAC)$ as separate inputs. For the latter case, you will need to extract the nonce from the binary string. Keep in mind that 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]

In this question, you will again generate a secret-key encryption of a message using the `libsodium` library, but instead of passing in a secret key directly, you will instead *derive* the key by iteratively hashing a pre-shared password and using a pre-shared salt. You should use the password and salt from the "Question 3" section of the "Question Data" page.

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[6] 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 (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.

Once you derive the key, you should use it to encrypt your message, as in question 2 part 1. Then, you should send this ciphertext message to Jessie by sending a request of the same format as in that question at the following web API page.

`https://whoomp.cs.uwaterloo.ca/458a3/api/psp/send`

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

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

You have received a ciphertext message from Jessie. Check your inbox using this web API page to receive the ciphertext message for this question.

`https://whoomp.cs.uwaterloo.ca/458a3/api/psp/inbox`

You should derive the secret key for your communication with Jessiefrom the pre-shared password and salt in the same way as you did for question 3 part 1. Decode and then decrypt the ciphertext message that you obtained from the above URL, using the key that you just derived. These two operations should be done in the same way as you did for question 2 part 2, You should enter the plaintext that Jessie 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 Jessie.

Question 4 part 1: upload a public 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 public keys must then be distributed somehow. For this assignment, the web API will act as a *public key directory*: everyone can upload a public key, and request the public keys that have been uploaded by other users.

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

`https://whoomp.cs.uwaterloo.ca/458a3/api/signed/set-key`

The request body should contain a JSON object with a `public_key` value containing the `base64` encoding of the public 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 public key.

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

Now that you have uploaded a public verification key, others can use it to verify that signed messages really were authenticated by you (or someone else with your secret key). In this question, you will first generate an unencrypted and signed message, using the secret signing key that you generated in the previous question. In the C library, you can generate the "combined mode" signature using the `crypto_sign` function. You may find an equivalent function for your language. Perform a `base64`-encoding of the output of this function.

Next, send this message to Jessie by sending a request to the following web API page in the usual way (same as for question 2 part 1):

`https://whoomp.cs.uwaterloo.ca/458a3/api/signed/send`

The `message` value in your request body should contain the above `base64` encoding. Jessie will be able to verify the authenticity of your message using your previously uploaded public 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.

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 Jessie'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 Jessie 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 for that key, then the server would be able to decrypt it; it could even re-encrypt it under Jessie'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 Jessie'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). The "fingerprint" of the public key is obtained by passing the key through a cryptographic hash function. Thus, when you obtain a public-key fingerprint from another person, you should obtain the associated public key from a public-key directory, compute the expected fingerprint and compare it to the one that was given to you. For this part, you must download Jessie's public key from the web API, and then compute its fingerprint, as follows.

1. Submit a `POST` request in the usual way to the following web API page:

   `https://whoomp.cs.uwaterloo.ca/458a3/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, `jessie`). 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 key.
2. Perform `base64` decoding of the public key to obtain a binary string.
3. Compute a fingerprint of the public key binary string using the BLAKE2b hash function provided by the `libsodium` implementation for your language. 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). The resulting hash is what you would compare to the one that Jessie securely gave to you.
4. To get the marks for this part, enter the hash of the public key, in hexadecimal encoding, into the "Question 5" section of the "Question Data" page.

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

In this question, you will first generate and upload a public key and then send an authenticated, encrypted message to Jessie. 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 keys for this question will be used for authenticated encryption rather than digital signatures, and so different cryptography is involved.

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

   `https://whoomp.cs.uwaterloo.ca/458a3/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 an authenticated, encrypted message to Jessie.
3. Encrypt your message using the equivalent of the C function `crypto_box_easy` in your language. You will need Jessie's public key in the form of a binary string, which you obtained and *decoded* in steps 1 and 2 in the previous part of this question. This function takes as input Jessie's public key, your secret key, and a nonce. 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. The function outputs the combination of a ciphertext and a message authentication code (MAC).
4. Perform `base64` encoding on the ciphertext-and-MAC binary string. Abstractly, your request body should look something like this:

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

5. Finally, send the message to Jessie in the usual way using the following web API page:

   `https://whoomp.cs.uwaterloo.ca/458a3/api/pke/send`

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

In this question, you will be decrypting a message that Jessie has sent to you. Check your inbox in the usual way with a request to the following web API page:

`https://whoomp.cs.uwaterloo.ca/458a3/api/pke/inbox`

Process the ciphertext message as follows to obtain the plaintext message.

1. Perform `base64` decoding on the message to obtain a ciphertext binary string.
2. To decrypt the above string from Jessie, you will need your secret key and Jessie's public key, which you obtained through steps 1 and 2 in question 5 part 1. Pass these three strings as input to the equivalent of the C function `crypto_box_open_easy` in your language to obtain the plaintext message.

3. Enter the plaintext English message in the "Question 5" section of the "Question Data" page to receive marks for this question.

Question 6: Government Surveillance [4 marks]

The government has decided that they must be able to decrypt all secure messages sent between you and Jessie through the web server. They have devised a new protocol that will protect the contents of your message from everyone except you, Jessie, 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 Jessie's public key. In this new protocol, you will *also* encrypt the secret key using the government's public key. This way, both Jessie and the government will be able to use their secret key to decrypt the message.

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

In this question, you will encrypt a message to Jessiesuch that the government may also view the message.

1. Begin by generating a public key and a secret key. *Save* the secret key. Following this, upload a public key in the exact same way as for question 5 part 2, except using the following web API page:

   `https://whoomp.cs.uwaterloo.ca/458a3/api/surveil/set-key`

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

   `https://whoomp.cs.uwaterloo.ca/458a3/api/surveil/get-key`

3. Visit the "Question Data" page and obtain the government's public key (in `base64` encoding) from the "Question 6" section. Now it is time to create your encrypted message for Jessie. Do this by following the next steps using the appropriate functions for your language.
4. Generate a random key, called the *message key*, for "secret box" encryption.
5. Encrypt the plaintext with the message key using the same technique as question 2 part 1. The resulting ciphertext is called the *message ciphertext*. The nonce for this ciphertext is called the *message nonce*.
6. Encrypt the message key with Jessie's public key and your secret key using the same technique as question 5 part 2. The resulting ciphertext is called the *recipient ciphertext*, and its nonce is called the *recipient nonce*.

7. Encrypt the message key with the government's public key and your secret key using the same technique as question 5 part 2. 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 the following values, in the *given* order. (The length of each item is shown beside it, following the names of constants in the C `libsodium` library.)

1. The recipient nonce - first `crypto_box_NONCEBYTES` bytes
2. The recipient ciphertext - includes encrypted message key (first `crypto_secretbox_KEYBYTES` bytes) and a MAC (next `crypto_box_MACBYTES` bytes).
3. The government nonce - `crypto_box_NONCEBYTES` bytes
4. The government ciphertext - includes encrypted message key (first `crypto_secretbox_KEYBYTES` bytes) and a MAC (next `crypto_box_MACBYTES` bytes).
5. The message nonce - `crypto_box_NONCEBYTES` bytes
6. The message ciphertext - includes encrypted plaintext (of message length) and a MAC (last `crypto_box_MACBYTES` bytes).

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

`https://whoomp.cs.uwaterloo.ca/458a3/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 Jessie has sent to you. Check your inbox in the usual way with a request to the following web API page:

`https://whoomp.cs.uwaterloo.ca/458a3/api/surveil/inbox`

1. First, you will need to obtain the recipient ciphertext and nonce. You will then decrypt this ciphertext in order to obtain the message key, which can then be used to decrypt the message. Note that the message from Jessie is of the same format as the one that you sent to Jessie in the previous part of this question. Knowing this, you can therefore obtain a substring of the message corresponding to the recipient nonce and the ciphertext. (You can completely ignore the government ciphertext.)
2. To decrypt the recipient ciphertext from Jessie, use Jessie's public key and your secret key, which were obtained in steps 2 and 1 respectively of the previous part of this question. This will give you the message key.
3. To extract the message ciphertext, note that as the format of the entire message is identical to

the one you generated in the previous part of this question. Again, as the entire message has the same format as the one that you sent in the previous part, you can compute the expected indices of the message nonce and message ciphertext and extract the corresponding substrings.

4. Use the message key to decrypt the message ciphertext and recover the plaintext.
5. Provide the plaintext in the "Question 6" section of the "Question Data" page. Keep in mind that the plaintext should consist of random English words.

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 keys. Stealing long-term secret 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*.

- The identity verification key is used for authentication via digital signatures. It is produced in the same way as in question 4.
- A "prekey" is just an ordinary public key with a short lifetime.[7] 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 Jessie in question 5, the "box" was created using Jessie's public key and your secret key, both of which are long-lived. Here, the "box" will be created using Jessie's *signed prekey* and the secret for your *signed prekey*.

[7] This terminology was introduced by the secure messaging application called Signal.

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. To do so, go through the following steps:

1. The first step is to generate and upload an identity verification key in the same manner as question 4 part 1. As mentioned in that question, *save* the signing key for step 3 below. The only difference is that the `base64` encoded verification key will be sent to the URL shown below.

   `https://whoomp.cs.uwaterloo.ca/458a3/api/prekey/set-identity-key`

2. 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. *Save* the secret key.
3. To produce a signed prekey, use your identity signing key, which was generated in step 1 above, to sign the public 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.
4. 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://whoomp.cs.uwaterloo.ca/458a3/api/prekey/set-signed-prekey`

Note that for both steps 1 and 4 above, if you received a `200` HTTP status code in response, then the request has been successful (and you should receive the mark for this part for successful completion of step 4).

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

Now that you have uploaded a signed prekey, go through the following steps to send a message to Jessie.

1. Download Jessie's identity verification key sending a request containing a JSON object with `jessie` in the `user` key to this web API page:

   `https://whoomp.cs.uwaterloo.ca/458a3/api/prekey/get-identity-key`

   Your request body should look similar to this:

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

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

2. Perform `base64` decoding on the key that you obtained in the previous step to obtain the binary string form of the identity verification key.
3. Using the exact same technique as in step 1, obtain Jessie's signed prekey by sending a request to this web API page:

   `https://whoomp.cs.uwaterloo.ca/458a3/api/prekey/get-signed-prekey`

   This time, you should receive Jessie's signed prekey, encoded using `base64`, in the `public_key` property of the JSON object.
4. Perform `base64` decoding on the key that you obtained in the previous step to obtain the binary string form of the signed prekey.
5. Verify the signature on this prekey, using the identity verification key binary string that you obtained in step 2. To do so, use the equivalent of the C function `crypto_sign_open` in your language. If the signature is valid, this function will return the content that was signed, which is Jessie's public key to use for encryption.
6. Finally, you can encrypt a message to send to Jessie. Encrypt the plaintext using Jessie's prekey (which you would have obtained at the end of the previous step) as the public key, and the secret key associated with your prekey (which was generated in step 2 of the previous part of this question). Pass to the equivalent of the C function `crypto_box_easy` in your language to perform public-key authentication, as you did in step 3 of question 5 part 2. As before, you should include a newly generated random nonce in your message.
7. Send the nonce and the ciphertext to Jessie by sending a request in the usual manner to the following web API page:

   `https://whoomp.cs.uwaterloo.ca/458a3/api/prekey/send`

   Abstractly, your request body should look something like this:

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

   If Jessie 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 Jessie using the same encryption scheme as the previous part. Check your inbox in the usual manner using the following web API page:

`https://whoomp.cs.uwaterloo.ca/458a3/api/prekey/inbox`

When you receive a message, you should look up Jessie's identity verification key and signed prekey, and obtain Jessie's public key, as in steps 1–5 of the previous part of this question (if you have not done so already). Using Jessie's prekey and the secret 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 Jessie, 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. The programming part TA is also available in this environment, with username `m2mazmud`. To communicate, use the following web API pages:

`https://whoomp.cs.uwaterloo.ca/458a3/api/freedom/get-identity-key`
`https://whoomp.cs.uwaterloo.ca/458a3/api/freedom/set-identity-key`
`https://whoomp.cs.uwaterloo.ca/458a3/api/freedom/get-signed-prekey`
`https://whoomp.cs.uwaterloo.ca/458a3/api/freedom/set-signed-prekey`
`https://whoomp.cs.uwaterloo.ca/458a3/api/freedom/send`
`https://whoomp.cs.uwaterloo.ca/458a3/api/freedom/inbox`
`https://whoomp.cs.uwaterloo.ca/458a3/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.

What to hand in

All assignment submission takes place on the `student.cs` machines (not ugster or the virtual environments), using the `submit` facility. In particular, log in to the Linux student environment (`linux.student.cs.uwaterloo.ca`), go to the directory that contains your solution, and submit using the following command: `submit cs458 3 .` (dot included). CS 658 students should also use this command and ignore the warning message. Hand in the following files:

a3-responses.pdf: A PDF file that contains your answers to all written response questions. You must include your name, your uWaterloo userid, and your student number at the top of the first page of a3.pdf. **-5 marks if it doesn't!** Also, be sure to "embed all fonts" into your PDF files. Some students' files were unreadable in the past; if we can't read it, we can't mark it.

xor, plaintext1, plaintext2: For the first question of the written part of the assignment.

a3code.tar an uncompressed tar file containing suitably named source code files for all parts of the programming question. While we will not run your code for the programming part, it should be evident 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 they were shown on the assignment website.