UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 **Computer Security and Privacy** **Winter 2023**
Meng Xu
Yousra Aafer

ASSIGNMENT 3

Assignment release: **Fri, March 17, 2023**
Milestone due date: **Fri, March 24, 2023 at *3:00 pm***
Assignment due date: **Mon, April 10, 2023 at *3:00 pm***

Total Marks: 92 (44 written + 48 programming)
Bonus Marks: 0 (no bonus marks)

Written Response TA: Shufan Zhang
Programming TA: Xiaohe Duan

TA Office hours: Fridays 1:00 – 2:00 pm
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-ec2-aec-tyb - Access code 821815

Programming: https://bbb.crysp.org/b/you-pqv-8rg-xsq - Access code 255717

Please use Piazza for all communication. Ask a private question if necessary.

Note that the Milestone is MANDATORY and includes Q1 and Q2 of the programming questions

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.

For **written questions** and **code for programming questions**, the assignment submission takes place on the `linux.student.cs` machines (**not** ugster or the virtual environments), using the submit utility. Log in to the linux student environment, go to the directory that contains your solution, and submit using the following command:

`submit cs458 3 .` (dot at the end)

CS 658 students should also use this command and ignore the warning message.

By the **milestone deadline**, you should hand in:

- **a3-milestone.tar**: Your source files for the **Questions 1 and 2 of the programming questions**, in your supported language of choice, inside a tarball (created as outlined below). **Note that this milestone is mandatory!!** You will not be able to submit the tarball for these questions after the milestone deadline.

By the **assignment deadline**, you are required to hand in:

- **a3-written.pdf**: A PDF file containing your answers for the written-response questions.

- **a3-code.tar**: Your source files for the programming assignment, in your supported language of choice, inside a tarball. While we will not run your code, 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 marks were awarded on the assignment website.

  To create the tarball, `cd` to the directory containing your code, and run the command
  ```
  tar cvf filename.tar .
  ```
  (pay attention to the `.` at the end).

Written Response Questions [44 Marks]

Q1: Two-Time Pad [10 marks]

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.

In class, you have studied the usage of one-time pads using the XOR operation. However, another way to generate a ciphertext from a plaintext and a one-time pad is to use mod 128 arithmetic. To produce the ciphertexts you have been provided with for this question, we began by generating a random 300-byte pad and then encrypted each plaintext *using the same pad* by adding the pad to the plaintext, followed by modulo 128. **Note that the addition is done byte-by-byte; i.e., the first byte of the plaintext is added to the first byte of the pad, followed by modulo 128, and this is repeated for all bytes.**

$$\texttt{ciphertext} = (\texttt{plaintext} + \texttt{pad}) \mod 128$$

The following algorithm shows how this is done:

Algorithm 1 Encrypt

```
Require: plaintext, pad
  ciphertext = bytearray()
  for i from 0 to 299 do
    cipherbyte = (plaintext[i] + pad[i]) mod 128
    ciphertext.append(cipherbyte)
  end for
  return ciphertext
```

The plaintexts, ciphertexts, and pad are unique to you (your classmates have been given different ciphertexts generated from different plaintexts using different pads).

1. [3 marks] What is $(\texttt{plaintext2} - \texttt{plaintext1}) \mod 128$? Submit it as a 300-byte binary file called `p2p1`. **Note:** This subtraction is done the same way as the addition; i.e., it is done byte-by-byte.
2. [7 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.

Q2. Cryptography Applications: TLS certificates [10 Marks]

Alice is the CEO of a company that provides all its employees with a laptop. Mallory, an employee in the company, is trying to learn Alice's login information for Alice's account on the company website: `www.comp.com`. For each of the following attacks, say whether Mallory will be able to learn Alice's login information. In addition, either **explain** the attack in more detail or describe why the attack fails. Assume that Alice's browser communicates only over TLS-protected connections, that the browser will terminate if it detects any TLS-related problems, and that Alice fails to detect phishing attacks.

(a) [2 marks] Mallory sets up a phishing website, `https://www.comp.mallory.com`. She gets a certificate for her website from a CA whose verification key is embedded in Alices's browser. She then uses a phishing email to fool her victim to click on the link `https://www.comp.mallory.com`, and Alice goes ahead and clicks.

(b) [2 marks] This scenario is identical to the first one with the following modification: During the TLS connection setup process, Mallory has her website send the original company certificate, which she retrieved earlier from the real company website, to the victim's browser.

(c) [2 mark] Mallory manages to execute a DNS cache poisoning attack on the victim's host, and `www.comp.com` now maps to the IP address of the machine hosting Mallory's website. In other words, when accessing `https://www.comp.com`, Alice's browser will talk to Mallory's website, which will give the original company certificate to the browser. Mallory also modifies her phishing email to contain the link `https://www.comp.com`, and Alice clicks on the link.

(c) [4 Marks] Suppose that Bob is a terrible webmaster and chooses an invalid TLS certificate for his online store website. Alice trusts her friend Bob and chooses to ignore her browser's warning and proceeds to Bob's website to buy some stuff. An attacker, Mallory, can perform a man-in-the-middle attack in this scenario to steal Alice's credentials and banking information. You know that man-in-the-middle attacks involve the attacker establishing a channel with both ends — the server (Bob) and the client (Alice). Note that Mallory might need to perform some preparation even before Alice's attempt to connect to Bob's website.

  (a) [2 marks] Specify what cryptographic information Mallory needs to generate or modify so that Alice's vulnerable browser (the client) will establish a secure channel with Mallory.

  (b) [2 marks] Mallory needs to convince Alice that she is communicating with Bob's server, to reassure her that her purchase was successful. Present a *clear and concise* protocol that Mallory executes in order to convince Alice of a successful purchase and to steal her banking credentials. Mallory will need to communicate with the server and relay (repeat) information back to Alice and vice versa. So, in steps of your protocol, include what Mallory encrypts or decrypts, under a key shared with one of Alice or Bob, and sends to/receives from them.

Q3: Privacy Enhancing Technologies [8 Marks]

Mr. Goose is a Rogers ISP client and is suspected of watching pirated streams of NHL games on the Streams "R" Us website. To prevent its customers from watching such pirated streams, as of May 27th, Rogers has deployed an Internet filtering mechanism based on IP blocking (`https://twitter.com/kaplanmyrth/status/1534662222810468352`). To get around such blocking, Mr. Goose now leverages Tor to access the Streams "R" Us website.

You have been called to help Rogers in collecting evidence that Mr. Goose is still watching pirated streams on the Streams "R" Us website (now via Tor). Assume that 1) you know the IP addresses of both Mr. Goose's desktop and the streaming website, and 2) you have unlimited access to the packet logs of the Rogers ISP traffic. How can you link Mr. Goose's accesses to the streaming website in the following scenarios (if at all)?

1. [2 Marks] Mr. Goose's desktop and the streaming website are connected to the Internet through the Rogers ISP.
2. [2 Marks] Mr. Goose's desktop is not connected through Rogers, but the streaming website and the entry node of the Tor circuit are.
3. [2 Marks] Mr. Goose's desktop and the entry node are not connected through Rogers, but the middle and exit nodes are.
4. [2 Marks] Mr. Goose's desktop is connected through Rogers, but neither the streaming website or the Tor relays are connected to Rogers.

Q4: Database Privacy [16 Marks]

Inference Control [7 Marks]

Alice, from Cryptopia, was sent to execute a secret mission to gather information about the competing company, Cryptomania's next marketing event in their headquarters. This mission has been recorded in Cryptopia's database which is protected by access control policies. The record has the following fields, for example,

This table is protected by mandatory access control. The *Destination* and *Mission* fields are marked as *Top Secret* and the other fields are marked as *Confidential*. Therefore, Bob (with privilege

Executor	Destination	Location Code	Mission	Salary
Alice	Cryptomania Headquarters	N2L 233	Spying	5000 Dollars

Secret) only knows Alice is executing a mission but does not have access to the exact information about this secret mission.

Bob cares about Alice. He wants to know *what the mission is and where the mission is executed.* And soon, he finds some additional side information...

(a) Each position is associated with distinct Location codes (i.e., zipcode).

(b) There are some past mission records marked as *Confidential* in the same database.

Executor	Destination	Location Code	Mission	Salary
Eve	East Lake Center	W1E 514	Marketing	1000 Dollars
Charlie	Informatica Headquarters	123 G7A	Spying	5000 Dollars

Note: this is related to the inference (control) problem in access control research. Interested student may check out a recent VLDB'22 paper (Don't be a tattle-tale: Preventing Leakages through Data Dependencies on Access Control Protected Data) for reference.

1. [3 Marks] Could you explain **how** Bob could use these side knowledge to get the information he wants?
2. [4 Marks] Could you come up with a (defence) solution to control information leakage through this type of side-channel knowledge? Please justify your answer in terms of both aspects of privacy and utility.

Differential Privacy [9 Marks]

Tinker realizes that an adversary could use background knowledge to gain more information from Table 1, in which case k-anonymization may not be enough. Thus, they switch to using differential privacy to publish the data, since it is resilient to background knowledge. To do this, Tinker releases a differentially private histogram showing the number of users having a specific number of matches. To generate this histogram, Laplace noise is added to the true value. For instance, if 4 users in the dataset have 5 matches each, Laplace noise would be added to the total number of such users (4) to hide the true number of users with 5 matches.

The *histogram* representation of the dataset $x = (x_0, \dots, x_{n-1})$, where $n = 36$, is a 36-dimensional vector (ranging from a minimum of 0 matches to the maximum number of matches observed in the

Name	Age	Gender	Num. Matches
Henry	42	Male	8
Sarah	36	Female	16
Austin	22	Non-Binary	5
Adrian	44	Male	12
Natalie	30	Female	5
Chloe	23	Non-Binary	20
Tony	45	Male	13
Christine	28	Non-Binary	20
Olivia	39	Female	35

Table 1: Number of Matches Information

dataset, 35) where the j-th entry is the number of x's rows whose number of matches is equal to j. For instance, according to the data in Table 1, $h_{20}(x) = 2$ since two users in the database (Chloe and Christine) have 20 Tinker matches, and thus:

$$h(x) := (h_0(x), \ldots, h_{n-1}(x)), n = 36$$

Consider the following *noisy histogram algorithm* output:

$$\hat{h}(x) := (h_0(x) + L_0, \ldots, h_{n-1}(x) + L_{n-1})$$

where every $L_j \sim Laplace(\lambda)$ is independent Laplace Noise.

Note: Wikipedia gives a good overview of differential privacy and differentially private mechanisms: `https://en.wikipedia.org/wiki/Differential_privacy`. You may also seek out additional resources to help answer this question.

1. [1 Mark] What is the sensitivity of this query of releasing histograms?
2. [1 Mark] Tinker sets the parameters to $\epsilon = 0.01$, then what is λ in Laplace Mechanism?
3. [1 Mark] Please analyze the expected error of this mechanism. ($\mathcal{E} = \sum_{i=1}^{d} \mathbb{E}[(o_i - c_i)^2]$, where o_i is the ith entry of the noisy output, and c_i is the ith entry of the true answer.)
4. [3 Marks] Does this mechanism satisfy the definition of ϵ-differential privacy? Will the histogram output of this mechanism be useful? Justify.
5. [3 Marks] Tinker would like to collect new user data with local differential privacy guarantee. Consider a domain $\Sigma = \{l_1, \ldots, l_k\}$ of k locations, please design a randomized response R that takes in a true location $l \in \Sigma$ and randomly outputs a location $o \in \Sigma$. (Describe your algorithm and show that the algorithm achieves ϵ-local differential privacy.)

Programming Questions [48 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`[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 *Muffin* in order to confirm that your code is correct.

Muffin

[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 *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://doc.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.

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
- "Generic hashing": using BLAKE2b

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

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

- 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 generic third-party libraries, but of course you may not copy code from other people's solutions to this (or similar) assignment. If you use 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://readthedocs.org/projects/lmctvpynacl/downloads/pdf/use-autodoc-for-apis/`) 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. 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.
- **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.se/libcurl/`) for submitting API requests, `Jansson` (`https://github.com/akheron/jansson`) for process-

ing 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 on the ugsters.

Question 1: Using the API [10 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 [6 marks]

Your first task is to send an unencrypted message to Muffin 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 Muffin, your JSON object should also contain `recipient` and `msg` keys. The `recipient` key specifies the username for the recipient of your message; this should be set to `Muffin`. The `msg` key specifies the message to send, encoded using `base64`.

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

```
{"api_token": "3158a1a33bbc...9bc9f76f",
"recipient": "Muffin", "msg": "SGVsb...kIQ=="}
```

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 [4 marks]

Next, you will use the web API to receive a message that Muffin 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 `msg_id`, `sender`, and `msg` keys. The `msg_id` is a unique number that identifies the message. The `sender` value is the username that sent the message to you. The `msg` value contains the `base64`-encoded message.

Decode the message that Muffin sent you. The message should contain recognizable English words. **The messages from Muffin 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 Muffin sent to you in the "Question 1" section to receive your mark.

Question 2: Pre-shared Key Encryption [10 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 Muffin 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 [6 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 `msg` 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", "recipient": "Muffin", "msg":
  base64encode(concat(nonce, secretbox(plaintext, nonce, key)))}
```

To receive marks for this part, send an encrypted message to `Muffin` 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 [4 marks]

Muffin 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: Digital Signatures [10 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 Muffin.

Question 3 part 1: upload a public verification key [4 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 Muffin, 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",
  "pubkey": "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 3 part 2: send a message [6 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 Muffin by sending a request to the following web API page in the usual way:

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

The `msg` 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. Muffin will be able to verify the authenticity of your message using your previously uploaded verification key.

Question 4: Public-Key Authenticated Encryption [10 marks]

While authentication is an important security feature, the approach in question 3 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). Internally, `libsodium` performs a *key exchange* between your secret key and Muffin'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 4 part 1: verify a public key [4 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 Muffin 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 Muffin'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 Muffin'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 Muffin'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, `Muffin`). The server's response will be a JSON object containing `user` (the requested username) and `pubkey`, 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 Muffin 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 4 part 2: send a message [4 marks]

Before sending a message to Muffin, you will first need to generate and upload a public key. While the key pairs generated in question 3 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 3 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 Muffin. Encrypt your message using the equivalent of the C function `crypto_box_easy` in your language. This function takes as input Muffin'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", "recipient": "Muffin",
                  "msg": base64encode(
  concat(nonce, box(plaintext, nonce, Muffin_public, your_secret))
                          )}
```

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

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

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

To receive the mark for this part, you will need to decrypt a message that Muffin 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 Muffin, you will need your secret key and Muffin'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 4" section of the "Question Data" page.

Question 5: Government Surveillance [8 marks]

The government has decided that they must be able to decrypt all secure messages sent between you and Muffin through the web server. They have devised a new protocol that will protect the contents of your message from everyone except you, Muffin, 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 Muffin'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 Muffin and the government will be able to use their secret key to decrypt the message.

Question 5 part 1: send a message [6 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 Muffin'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 5" section.

Now it is time to create your encrypted message for Muffin. First of all, you need to generate a *message key* and encrypt the message using secret box encryption as you did in question 2. Both Muffin and the government need this *message key* to decrypt the message. To securely send both of them your *message key*, you should encrypt the key with your *public encryption private key* and the government/Muffin's public key, so they are the only two parties except you that can decrypt to get the *message key* and decrypt your message. For this part, you can refer to the techniques in question 4.

Then, you can construct the request body like this:

```
{"api_token": "3158a1a33bbc...9bc9f76f", "recipient": "Muffin", "msg":
      base64encode(concat(nonce, secretbox(plaintext, nonce,
           secret_key))),  "recipient_key": base64encode(
        concat(nonce, box(message_key, nonce, Muffin_public,
                         your_private))
                     ),  "gov_key":  base64encode(
      concat(nonce, box(message_key, nonce, government_public,
                         your_private))
                               )}
```

Send this message using `base64` encoding to Muffin in the usual way with a request to the following web API page to earn 6 marks:

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

Question 5 part 2: receive a message [2 mark]

To receive the mark for this part, you will need to decrypt a message that Muffin 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 Muffin, use Muffin'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 key. Use the message key to decrypt the message ciphertext and recover the plaintext. Provide the decrypted plaintext in the “Question 5” section of the “Question Data” page.