UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

**CS 458/658** **Computer Security and Privacy** **Fall 2017**

**Urs Hengartner, Stefanie Roos**

# ASSIGNMENT 3

Assignment due date: **Friday, December 1st, 2017 3:00 pm**

**Total Marks: 51**

**Written Response Questions TA:** Peiyuan Liu peiyuan.liu@uwaterloo.ca
(Office hours: Monday 2:00 pm–3:00 pm in DC 3333)
**Programming Questions TA:** Navid Nasr Esfahani nnasresf@uwaterloo.ca
(Office hours: Wednesday 1:30 pm–2:40 pm in DC 3333)

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 [21 marks]

**Note**: For written questions, please be sure to use complete, grammatically correct sentences where appropriate. You will be marked on the presentation and clarity of your answers as well as the content. You are welcome to use additional resources for solving the questions but be sure to cite them appropriately.

1. [7 marks total] **Diffie-Hellman**

   Diffie-Hellman (DH) key exchange allows Alice and Bob to exchange a secret key using an insecure channel. See http://mathworld.wolfram.com/Diffie-HellmanProtocol.html for detail.

   (a) [2 marks] Assume Alice and Bob agree to use modulus $p = 97$ and base $g = 5$. Then Alice chooses secret parameter $a = 36$ and Bob chooses secret parameter $b = 58$. What are the public values that Alice gives to Bob and Bob gives to Alice? What is the resulting secret key that is generated as a result of DH protocol?

   (b) [2 marks] Assume that Alice and Bob agree to use modulus $p$, and base $g$. During the key exchange, Eve observes these values as well as public parameters sent by Alice and Bob: $A = g^a \pmod p$ and $B = g^b \pmod p$. Can Eve recover original secret values $a$ or $b$ given public values? Explain.

   (c) [3 marks] If Mallory behaves as active Man-In-The-Middle (MITM) attacker, how can she manipulate the DH protocol to obtain all of the plaintext communications between Alice and Bob? How can this be prevented?

2. [8 marks total] **GnuPG**

   A GnuPG public key for peiyuan.liu@uwaterloo.ca is provided along with the assignment on the course website (peiyuan.liu.asc). Perform the following tasks. You can install GnuPG on your own computer, or use the version we have installed on the ugster machines.

   (a) [2 marks] Generate a GnuPG key pair with 2048 bits for yourself. Use RSA as the encryption algorithm, your real name, and your uwaterloo university email address. Export this key using ASCII armor into a file called **key.asc**. [Note: older versions of GnuPG might not have the RSA option, so check that the version you are using has this option. The ugster machines have a new enough version, but the student.cs machine may not.]

   (b) [2 marks] Use this key to sign (not local-sign) the peiyuan.liu@uwaterloo.ca key. Its true fingerprint is: `F70B 904A 616B B1FD BAAB 8BED 6151 5CBD FF6B 05F3`. Export your signed version of the peiyuan.liu key into a file called **peiyuan.liu-signed.asc**; be sure to use ASCII armor. [Note: signing a key is not the same operation as signing a message.]

(c) [2 marks] Create a message containing your userid and name. Sign it using the key you generated, and encrypt it to the peiyuan.liu key. You should do both the encryption and signature in a single operation. Make sure to use ASCII armor, and save the output in a file called **message.asc**.

(d) [2 marks] Briefly explain the importance of fingerprints in GnuPG. In particular, explain how users should check fingerprints and what type of attacks are possible if users do not follow this procedure properly.

3. [6 marks total] **Inference Attacks**

The Human Resources department of FrobozzCo International has a table called `Employee`, containing $N$ records, in its database. This table stores the following information about each employee:

- Name: The employee's first name, which is unique for every employee in the database.
- Birthdate: The employee's year of birth.
- Occupation: The position the employee fills at the company. An employee with an occupation other than "Staff" is considered a specialist.
- Allegiance: The realm the employee swears their allegiance to. Although FrobozzCo is located in the land of Quendor, they believe that a diverse work force is beneficial to the production of magical products and employ workers from as far away as Antharia and Kovalli.
- Salary (in Zorkmids): the amount of hard-earned cash each employee takes home over the course of the year.

Below is an excerpt (not the entire table):

Table 1: Part of `Employee` Table

| Name | Birthdate | Occupation | Allegiance | Salary |
|---|---|---|---|---|
| John | 725 | Founder | Quendor | 15,000 |
| Blair | 788 | Enchanter | Quendor | 10,225 |
| Natalie | 770 | Staff | Quendor | 5,000 |
| Ralph | 737 | Public Relations | Quendor | 13,721 |
| Frank | 741 | Staff | Antharia | 4,908 |
| Leonardo | 731 | Director of Design | Quendor | 12,000 |
| Jill | 738 | Navigation Expert | Antharia | 10,426 |
| Matthew | 722 | Staff | Kovalli | 5,923 |
| Rachel | 760 | Staff | Kovalli | 4,907 |

To deter employees from comparing their earnings and complaining about the amount of Zorkmids they take home to their families, the database is set up to suppress the `Salary` field in the output of queries. However, users can execute queries of the form

```
SELECT SUM(Salary) FROM Employee WHERE ...
```

where queries that match fewer than $k$ or more than $N - k$ but not all $N$ records are rejected. We will use $k = floor(\frac{N}{8})$.

(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 Leonardo's salary. Both the tracker and the three queries need to be of the form

```
SELECT SUM(Salary) FROM Employee ...
```

Assume that employees' names are unique and not known to the attacker (apart from Leonardo's) and that the attacker has no additional information about Leonardo (not even his status as a specialist). The only knowledge the attacker knows about the underlying distribution of the database is that FrobozzCo International hires an equal number of specialists (i.e. with an occupation other than "Staff") and staff employees. In your solution, you should give 1) your tracker, and 2) the set of three queries.

(b) [3 marks] After reports of strife between the Kingdom of Quendor and surrounding lands, Pseudo-Duncanthrax, the King of Quendor declared employee allegiance to be private information. Cornelius pondered the problem of protecting the newly sensitive information in their employee database. After having discovered the flaws in his previous defences, he decided to discuss the problem with his brother John during the Flathead family celebration of Undergroundhog Day. Together, they decided to anonymize the database, producing the following table:

Declaring the first three columns of the table to be identifiers, the brothers proudly proclaim that the table segment above is 3-anonymous. Are they correct? If so, explain your answer. If not, produce a 3-anonymous table and give the value $\ell$ for which the table is $\ell$-diverse.

| Name | Birthdate | Occupation | Allegiance |
|---|---|---|---|
| * | 74* | Specialist | Quendor |
| * | 83* | Specialist | Quendor |
| * | 73* | Staff | Quendor |
| * | 74* | Specialist | Antharia |
| * | 77* | Staff | Antharia |
| * | 72* | Specialist | Quendor |
| * | 82* | Specialist | Antharia |
| * | 84* | Specialist | Antharia |
| * | 77* | Staff | Kovalli |
| * | 73* | Staff | Kovalli |

Table 2: Part of `Employee` Table (Anonymized)

## Programming Questions [30 marks]

### Background

In this section we will study padding oracle attacks. Block ciphers operate on a fixed number of bits, such as 64 (DES) or 128 (AES). To encrypt longer messages, a mode of operation such as CBC can be used (see Figure 1). However, this still requires that the message length is a multiple of the block size. In order to accommodate arbitrary-length messages, modes of operation apply a padding scheme. The decryption routine will need to check that the message padding is valid. Unfortunately, information about the validity of the padding can easily be leaked to an attacker. Based on the feedback regarding padding validity, an attacker might infer plaintext properties. More precisely, a "padding oracle", which reveals no information apart from the validity of the padding of a given ciphertext, can allow the attacker to decrypt (and sometimes encrypt) messages without knowing the encryption key.

The padding oracle attack was originally described in a 2002 paper by Serge Vaudenay. This paper is available at `https://www.iacr.org/archive/eurocrypt2002/23320530/cbc02_e02d.pdf` and will serve as your reference for this attack (Reading the first three sections will be enough to tackle this assignment; however, reading the whole paper is recommended). We provide some hints that might be helpful in understanding the paper in Section "Remarks on Required Reading". Padding oracle attacks similar to Vaudenay's attack have continued to be relevant since their inception, with new attacks being discovered regularly. One recent example is the POODLE attack on SSL 3.0 and TLS discovered by Google security engineers in 2014 (not a required reading).

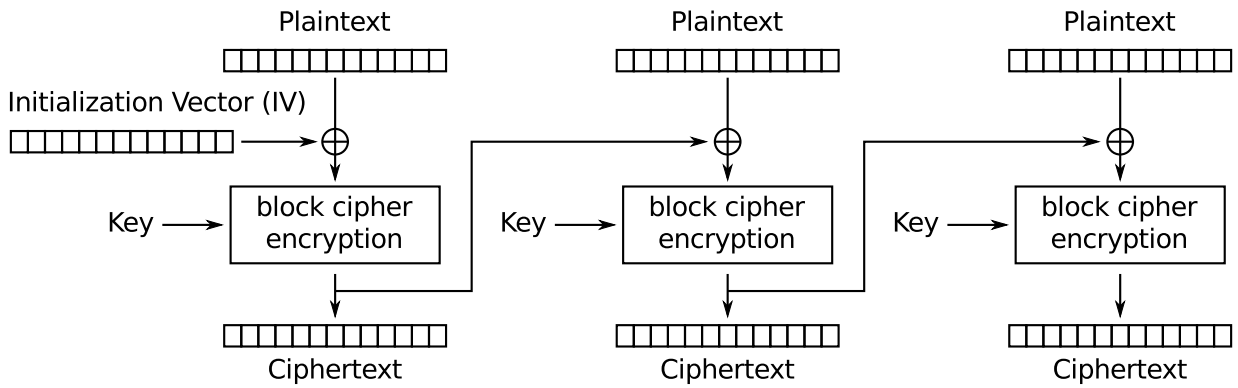


Figure 1: CBC Mode Encryption

### Application Description

A simple web application is running at `http://localhost:4555` on each ugster machine. To view this application in your web browser, you can use `http://ugsterXX.student.cs.uwaterloo.ca:4555` (when off campus you need to use the university's VPN). However, please use `http://localhost:4555` in your programs to ensure that they run quickly no matter which ugster machine we test them on.

When you visit the web application it will set a cookie named `user` in your browser. In other words, the HTTP response will contain a `Set-Cookie` header like the following:

```
Set-Cookie: user="K/wLMO+V8Qbo5B4spv6/PP98W1mofu7vQT83dbascyLcyc9mlFi4qzLyYjsyTLWF"
```

The value of this cookie is encrypted with AES using a key known only to the web server. AES is used in CBC mode, and the randomly chosen IV (=nonce) is prepended to the ciphertext. The result is then Base64 encoded to form the cookie value. Base64 is a standard way of encoding binary data into printable ASCII characters. It is implemented in many programming languages and the `base64` Unix utility, as specified in RFC 3548.

The padding scheme used by the web application is designed specifically for this assignment, and it is **not** a standard padding, appropriate for being used in real-life. A padding of length $n$ starts with the value 0x00, and is followed by a decreasing sequence of numbers form $n-1$ to 1. For example, 0x00 is a padding of length 1, 0x00 0x01 is a padding of length 2 , and 0x00 0x04 0x03 0x02 0x01 is a padding of length 5. All paddings start with a byte of value 0x00. There is always at least one byte of padding, i.e., messages whose length is a multiple of the block length receive an additional block of padding.

When you visit the web application again, your browser will send the cookie back to the server. This is done using a `Cookie` header of the same form as the `Set-Cookie` header above. The web server will attempt to decrypt any cookie sent to it using its encryption key. If the decryption is successful, the HTTP response will contain a 200 (OK) response code. However, if the decryption fails a 500 (Internal Server Error) response code will be returned. See the section below titled "HTTP with `curl`" for more information on working with HTTP. Instead of manually visiting the web application using your browser, you can also emulate browser behaviour in a programming language of your choice.

In this assignment, you analyze the scheme discussed in the above-mentioned paper, suggest a fix for the issue, then you modify the attack so that you can decrypt the cookies, and encrypt messages on behalf of the server, without having the secret key. Eventually, you implement one of the two other modes of operations, which do not overcome the problems of the presented CBC method.

Note that some of the questions require written answers, which should be provided as part of a3.pdf, together with the answers for the written part of the assignment.

### Questions

1. [3 marks] How would you modify the attack described in the paper to account for the different padding scheme used by the web server? List the lines in Sections 3.1 and 3.2 that require changes and show how they should be changed.

2. [2 marks] What is the average case and worst case number of padding oracle calls required to perform the modified attack?

3. [2 marks] What cryptographic tool could be used to fix this vulnerability? Which of the properties Confidentiality, Integrity, Authenticity does the chosen cryptographic tool provide? How does it fix the vulnerability? Assume that the web server still has to return a different response for users with valid versus invalid cookies.

4. [10 marks] Write a program called `decrypt` that implements the padding oracle attack on the web application described above. Your program will be called with a single command line argument containing a Base64-encoded cookie value to be decrypted. These cookies will have at least two 16-byte blocks (the IV and at least one ciphertext block), but does not need to have the same number of blocks as responses returned by the server. Your program should generate the appropriate cookie values, send them to the web server using HTTP, and observe its response codes in order to decrypt the given cookie value. It should print the resulting plaintext to `stdout`, which will consist of only printable ASCII characters. You can print debug output to `stderr` if you like. Your program should run in a maximum of 10 minutes.

   You may use any programming or scripting language available on the ugster machines. If your preferred language is not available, we may be able to accommodate requests. Most common programming languages have built-in libraries for making HTTP requests to web servers. Alternatively, you can use the `curl` program as described in the section below titled "HTTP with `curl`". You will submit a single file named `src.tar`. For evaluation, this file will be extracted and we will attempt to run an executable at the top level with `./decrypt <cookie>`. This executable could be your program itself, or it could be a script that compiles and then runs your program. For the example cookie shown above, the invocation would be:

   ```
   ./decrypt QvePu/BjvFOnonNGpvZmubz7pHrHeCv03rfi8PVE4B69Et7HKR8PfelN5lwCWDp2nfeSkx3Xo
   XK1OGiAAIDVDA==
   ```

5. [5 marks] Write a program called `encrypt` that encrypts arbitrary cookie values such that they will be correctly decrypted by the web application. Your program does not receive the web application's encryption key as input. It will be called with a single

command line argument containing a printable ASCII plaintext to be encrypted. Your program should generate the appropriate cookie values, send them to the web server using HTTP, and observe its response codes in order to encrypt the given value. It should print the resulting Base64-encoded ciphertext to `stdout`. You can print debug output to `stderr` if you like. Your program should run in a maximum of 10 minutes.

Submission is similar to the previous question. Your program source files should be included in the same `src.tar`. For this question, we will attempt to run an executable at the top level with `./encrypt <plaintext>`.

**Hint:** In CBC mode there are three stages a block goes through in decryption. It starts as a ciphertext, the ciphertext is decrypted by the block cipher to an intermediary value, and then the intermediary value is XORed with the previous ciphertext block (or the IV for the first block) to form the final plaintext value. If we know the intermediary value for a given ciphertext block, we can manipulate the IV to gain complete control over what that block will be decrypted to. If we want to produce a plaintext $x$, then the IV should be $x$ XORed with the intermediary value. This idea can easily be extended to multi-block messages. Start from the last block and move backwards. Pick any value to be the ciphertext for the last block and decrypt it (using your method from the previous question) to find the corresponding intermediary value. Then calculate the IV required to produce the desired plaintext. This IV will be the ciphertext for the second-last block, and so on.

6. [8 (half if as bonus) marks] Write a program called `CounterMode` that gets a message, and a key $K$ and encrypts the message using XChacha20-Poly1305 with the given $K$, using counter mode, with the padding described for this assignment. For any message $m$, the program first breaks it into blocks of proper length for XChacha20-Poly1305. Then, it encrypts the blocks with the algorithm using the CounterMode. The program should only print the ciphertext. Explain why the counter mode prevents the attack you performed in question 4.

   Submission is similar to the previous question. Your program source files should be included in the same `src.tar`. For this question, we will attempt to run an executable at the top level with `./CounterMode <plaintext> <Key>`.

7. [8 (half if as bonus) marks] Write a program called `PackageTransform` that has a message and a key $K$ as input. It then encrypts the message using XChacha20-Poly1305 with the given $K$ in combination with the following algorithm (see https://people.csail.mit.edu/rivest/pubs/Riv97d.prepub.pdf for the more detailed version):

   - Choose a random keys $K_{PT}$ and $K_0$, using the XChacha20 key generator.
   - For message blocks $x_1, x_2, \ldots, x_n$, compute $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$ as

$$\bar{x}_i = x_i \oplus E(K_{PT}, i),\ i \in \{1, 2, \ldots, n\}$$

$$h_i = E(K_0, \bar{x}_i \oplus i),\ i \in \{1, 2, \ldots, n\}$$

where $E$ is XChacha20-Poly1305 encryption.

- Compute

$$\bar{x}_{n+1} = K_{PT} \oplus h_1 \oplus h_2 \oplus h_3 \oplus \cdots \oplus h_n$$

(a) For $i \in \{1, 2, \ldots, n+1\}$ compute:

$$y_i = E(K, \bar{x}_i)$$

- output $y_1, y_2, y_3, \ldots, y_n, y_{n+1}$ and $K_0$.

When deriving the message blocks for the second step, apply the padding algorithm used during the first parts of this assignment. Your program should only print the ciphertext. Explain why the counter mode prevents the attack you performed in question 4.

Submission is similar to the previous question. Your program source files should be included in the same `src.tar`. For this question, we will attempt to run an executable at the top level with `./PackageTransform <plaintext> <Key>`, and the output should be formatted as `<ciphertext> <TransformKey>`.

For the programming questions, make sure your scripts do not generate any extra output (e.g. debugging output, compiling output, etc.)

## Remarks on Required Reading

Here are a couple of hints that should help in understanding the paper:

- The author frequently uses the term "word" where we would usually say "byte"
- The author denotes the block cipher encryption (see Fig. 1) by $C$, i.e., $C(x)$ computes the encryption of the block $x$. The corresponding decryption function is consequently called $C^{-1}$.
- In Section 3.1. of the paper, 1 is the most likely correct padding of a random string because only the last byte has to be guessed correctly. In contrast, to have correct paddings of the form 22, 333, 4444, and so on, multiple bytes have to be correct.
- Step 5 of the algorithm in Section 3.1 is one way to check if a certain word is part of the padding or not. If changing the word $r_n$ (by XOR-ing 1) affects the validity of the padding (i.e., turns it from valid into invalid), the $n$-th word is part of the padding, otherwise it is not.

## HTTP with `curl`

HTTP (Hypertext Transfer Protocol) is a simple protocol for transferring text over a network. If you've used the Internet, you've used HTTP. In this protocol, a client makes a request to a server and the server replies with a response. A request generally asks for a resource located at a particular URL, and the response provides that resource. These resources are often HTML web pages, but here we'll be using mostly plain textual content. Let's see an example of a request and a response:

```
\$ curl -v http://localhost:4555
> GET / HTTP/1.1
> Host: localhost:4555
> User-Agent: curl/7.47.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 37
< Set-Cookie: user="QvePu/BjvFOnonNGpvZmubz7pHrHeCv03rfi8PVE4B69Et7HKR8PfelN5lwCWDp2nfeSkx3Xo
XK1OGiAAIDVDA=="
< Server: TornadoServer/4.3
< Etag: "4b44c375286c4a668502645e4da51dd29441e4e9"
< Date: Wed, 08 Nov 2017 14:25:32 GMT
< Content-Type: text/html; charset=UTF-8
<
Hello! I'll remember when I saw you.
```

Note that some extra debug output from `curl` has been omitted. The lines starting with "`>`" are the client talking to the server, and lines starting with "`<`" show communication in the other direction. The HTTP request starts with a method; we'll only be using GET. It then specifies the requested path and the protocol version. The following lines contain headers which specify additional information. These are present in both requests and responses. The response starts by confirming the protocol version. It then presents the status code, which is essential for us as it is the source of our padding oracle. The main response header of interest to us is the `Set-Cookie` header. We can use the `base64` utility to decode it, although it will likely result in unprintable characters, so we'll display the binary data as a hex dump with `xxd`:

```
\$ echo "QvePu/BjvFOnonNGpvZmubz7pHrHeCv03rfi8PVE4B69Et7HKR8PfelN5lwCWDp2nfeSkx3XoXK1OGiAAIDVDA=="
 |base64 -d | xxd
00000000: 42f7 8fbb f063 bc53 a7a2 7346 a6f6 66b9  B....c.S..sF..f.
00000010: bcfb a47a c778 2bf4 deb7 e2f0 f544 e01e  ...z.x+......D..
00000020: bd12 dec7 291f 0f7d e94d e65c 0258 3a76  ....)..}.M.\.X:v
00000030: 9df7 9293 1dd7 a172 b538 6880 0080 d50c  .......r.8h.....
```

You can see that this cookie contains four 16-byte blocks. The first is the IV, and the remaining three are ciphertext blocks. To send the cookie back to the server, we can do the following:

```
\$ curl -v -b user=QvePu/BjvFOnonNGpvZmubz7pHrHeCv03rfi8PVE4B69Et7HKR8PfelN5lwCWDp2n
feSkx3XoXK1OGiAAIDVDA== http://localhost:4555
> GET / HTTP/1.1
> Host: localhost:4555
> User-Agent: curl/7.47.0
> Accept: */*
> Cookie: user=QvePu/BjvFOnonNGpvZmubz7pHrHeCv03rfi8PVE4B69Et7HKR8PfelN5lwCWDp2nfeS
kx3XoXK1OGiAAIDVDA==
>
< HTTP/1.1 200 OK
< Date: Wed, 08 Nov 2017 14:30:44 GMT
< Content-Length: 47
< Etag: "5521c43621121032ca93d4ba4ef260b16ee9a044"
< Content-Type: text/html; charset=UTF-8
< Server: TornadoServer/4.3
<
Hello! I first saw you at: 2017-11-08 09:25:32
```

It would be best to use an HTTP library for your programming language when writing your solutions, but you can call the `curl` program directly as a last resort. `curl` is also very useful for manually debugging web applications.

## Choosing a programming language

Before beginning the questions, you should choose 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.

However, you will need to use `libsodium` to complete the assignment. Hence, you have to choose a language that provides bindings for `libsodium`. Most common languages such as C, Java, Python, Javascript, and Go indeed provide the required binding but you should check the list of language bindings to find an interface for your language.

Not all `libsodium` language bindings support all of the features needed for this assignment.

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.

- **C**: While C has the best `libsodium` documentation, all of the other tasks are more difficult than other languages.

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 and libraries will be made available on the Ugsters in case you do not have access to a personal development computer. We might be able to install additional languages if required.

### `libsodium` documentation

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

https://libsodium.org/doc/

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

## What to hand in

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

**a3.pdf:** A PDF file containing your answers to the written response and relevant programming questions. **a3.pdf** must contain, at the top of the first page, your name, UW userid, and student number. If it does not, **a 3 mark penalty will be assessed**.

**key.asc:** Your GnuPG public key from question 2.

**peiyuan.liu-signed.asc:** Your signed version of Peiyuan's public key from question 2.

**message.asc:** Your encrypted and signed message from question 2.

**src.tar:** A tar archive directly containing your `decrypt`, `encrypt`, `CounterMode`, and/or `PackageTransform` program source files. (Please notice that these programs need to be executables, and without any extensions, e.g., `decrypt.sh` **is wrong** and should be sent as `decrypt`. To create the tarball, `cd` to the directory containing your code and run the command

```
tar cvf src.tar .
```

(including the `.`) in the directory where you have your files, **not** the parent directory.