**CS 346: Cryptography** **Fall 2023**

# Homework 2: Symmetric Cryptography

**Due:** September 27, 2023 at 11:59pm (Submit on Gradescope) **Instructor:** David Wu

**Instructions.** You **must** typeset your solution in LaTeX using the provided template:

`https://www.cs.utexas.edu/~dwu4/courses/fa23/static/homework.tex`

You must submit your problem set via Gradescope (accessible through Canvas).

**Collaboration Policy.** You may discuss your general *high-level* strategy with other students, but you may not share any written documents or code. You should not search online for solutions to these problems. If you do consult external sources, you must cite them in your submission. You must include the names of all of your collaborators with your submission. Refer to the official course policies for the full details.

**Problem 1: CBC Padding Oracle Attack [20 points].** Recall that when using a block cipher in CBC mode, the message must be a multiple of the block size. When encrypting messages whose length is not a multiple of the block size, the message must first be padded. In the TLS protocol (used for securing traffic on the web), if $v$ bytes of padding are needed, then $v$ bytes with value $(v-1)$ are appended to the message. As a concrete example, if 1 byte of padding is needed, a single byte with value 0 is appended to the message before applying CBC encryption. In TLS, the record layer is secured using an approach called "MAC-then-Encrypt[1]" (which as we will soon see, is not the ideal combination). At decryption time, the ciphertext is first decrypted (and the padding verified) *before* checking the MAC. In older versions of OpenSSL, the library reports whether a decryption failure was due to a "bad pad" or due to a "MAC verification failure." One might think that it was beneficial to provide an informative error message on decryption failure. As you will show in this problem, this turns out to be a disaster for security.

Suppose an adversary has intercepted a target ciphertext ct encrypted using AES-CBC. Let $\mathsf{ct}_i$ be any non-IV block in ct. Let $m_i$ be the associated message block. Show that if the adversary is able to submit ciphertexts to a CBC decryption oracle and learn whether the padding was valid or not, then it can learn the last byte of $m_i$ *with probability 1* by making at most 512 queries. Here, the CBC decryption oracle only says whether the ciphertext was properly padded or not; it does *not* provide the output of the decryption if successful. Then, show how to extend your attack to recover *all* of $m_i$. **Hint:** Start by showing how to test whether the last byte of $m_i$ is some value $t$ by making 2 queries to the decryption oracle.

*Remark:* Are there settings where the server would repeatedly decrypt ciphertexts of the user's choosing? It turns out that when using IMAP (the protocol email clients use to fetch email) over TLS, the IMAP client will repeatedly send the user's password to the IMAP server to authenticate. With the above padding oracle (implemented using a "timing channel"), an adversary can recover the client's password in *less than* an hour! This problem shows that if a decryption failure occurs, the library should provide *minimal* information on the cause of the error. This type of "padding oracle" attack was the basis of the "Lucky 13" attack on TLS 1.0 (2013)—many years after they were first discovered (2002) and thought to be patched!

[1] In MAC-then-encrypt, the encryption algorithm first computes a MAC $t$ on the message $m$, and the ciphertext is the encryption of the message-tag pair $(m, t)$.

**Problem 2: CBC Padding Oracle Attack, Part II [14 points].** In this problem, your task is to implement the CBC padding oracle attack you developed in the previous problem. We have provided starter code that contains an implementation of AES-CBC encryption using the Python `cryptography` library. Your task is to write an algorithm that takes as input a ciphertext encrypted using AES-CBC (with randomized IV) and outputs the associated message given access to a padding oracle. Specifically, the padding oracle takes as input a ciphertext and outputs `True` if the decrypted plaintext has a valid pad (as defined in the previous problem), and `False` if not.

Your task is to implement the `decrypt` method in `cbc.py`. You **cannot** change the interface for `decrypt`; otherwise, you are free to implement the algorithm however you prefer (using *standard* Python libraries, including the Python `cryptography` library). Your code will be evaluated only for correctness. Some helper functions are provided in `util.py`. Your attack must satisfy the following requirements:

- Your algorithm should support decrypting messages of *arbitrary* non-zero length. The message you return should *not* include any padding (you can use the `strip_padding` method in `util.py` to remove the padding).
- The input ciphertexts can be encryptions of arbitrary byte sequences (i.e., they are not necessarily ASCII-encoded strings).
- Your algorithm is allowed to make at most 8192 queries to the padding oracle for each non-IV block of the ciphertext. Note that this is an *upper* bound and many algorithms will require significantly fewer queries.

The following is the output of running `base.py` on our reference implementation (34 lines of code):

```
$ python3 base.py
Plaintext: b'CS 346'
Decrypted output: b'CS 346'
Successful decryption? True
Number of padding oracle queries: 1608
```

**Submission instructions:** Upload your code (consisting of *only* `cbc.py`) to Gradescope under Homework 2A. Note that your implementation must work with our provided `main.py` and `util.py`. Your submission will be autograded, and upon submission, your code will be run on a simple test case. There is **no** written component for this question.

**Problem 3: Cryptographic Combiners [20 points].** Suppose we have two candidate constructions $\Pi_1, \Pi_2$ of a cryptographic primitive, but we are not sure which of them is secure. A cryptographic combiner provides a way to use $\Pi_1$ and $\Pi_2$ to obtain a new construction $\Pi$ such that $\Pi$ is secure if at least one of $\Pi_1, \Pi_2$ is secure (*without* needing to know which of $\Pi_1$ or $\Pi_2$ is secure). Combiners can be used to "hedge our bets" in the sense that a future compromise of one of $\Pi_1$ or $\Pi_2$ would not compromise the security of $\Pi$. In this problem, we will study candidate combiners for different cryptographic primitives.

(a) Let $H_1, H_2 \colon \{0,1\}^* \to \{0,1\}^\lambda$ be arbitrary collision-resistant hash function candidates. Define the function $H(x) := H_1(x) \| H_2(x)$. **Prove or disprove:** if at least one of $H_1$ or $H_2$ is collision-resistant, then $H$ is collision-resistant.

(b) Let $(\mathsf{Sign}_1, \mathsf{Verify}_1)$ and $(\mathsf{Sign}_2, \mathsf{Verify}_2)$ be arbitrary MAC candidates[2]. Define $(\mathsf{Sign}, \mathsf{Verify})$ as follows:

- $\mathsf{Sign}((k_1, k_2), m)$: Output $(t_1, t_2)$ where $t_1 \leftarrow \mathsf{Sign}_1(k_1, m)$ and $t_2 \leftarrow \mathsf{Sign}_2(k_2, m)$.
- $\mathsf{Verify}((k_1, k_2), m, (t_1, t_2))$: Output 1 if $\mathsf{Verify}_1(k_1, m, t_1) = 1 = \mathsf{Verify}_2(k_2, m, t_2)$ and 0 otherwise.

**Prove or disprove:** if at least one of $(\mathsf{Sign}_1, \mathsf{Verify}_1)$ or $(\mathsf{Sign}_2, \mathsf{Verify}_2)$ is a secure MAC, then $(\mathsf{Sign}, \mathsf{Verify})$ is a secure MAC.

**Problem 4: Time Spent [1 point].** How long did you spend on this problem set? This is for calibration purposes, and the response you provide does not affect your score.

**Optional Feedback.** Please answer the following *optional* questions to help us design future problem sets. You do not need to answer these questions. However, we do encourage you to provide us feedback on how to improve the course experience.

(a) What was your favorite problem on this problem set? Why?

(b) What was your least favorite problem on this problem set? Why?

(c) Do you have any other feedback for this problem set?

(d) Do you have any other feedback on the course so far?

[2] Namely, you can assume that they are correct (but could be arbitrarily broken).