Cryptography Project

Due: Thursday, February 14, 2019 @ 11:59 pm

Contents

1 Introduction

You are a student at Minion University. In this project, you will be using cryptography to break the security of various elements of Minion University's IT infrastructure.

Each of the problems in this project are self-contained, so you can complete them (and turn them in) in any order.

1.1 Requirements

CS166 students must complete the three problems in Part I:

- Ivy (Section 2)
- Keys (Section 3)
- Transcript (Section 4)

CS162 students must complete the three problems in Part I, along with an additional component for Ivy (Section 2), and the two problems in Part II:

- Grades (Section 5)
- Padding (Section 6)

For all students, each problem is worth an equal portion of the overall credit for this project. For CS166, this means that each problem is worth $\frac{1}{3}$ of the available credit, and for CS162, $\frac{1}{5}$.

1.2 Late Handins

Late days will be applied automatically to this project. Please consult the syllabus for more information about late handins.

If there are extenuating circumstances preventing you from completing an assignment on time (e.g., illness), please contact the instructor before the assignment is due via the "Extension Requests" form on the `http://cs.brown.edu/courses/cs166/resources/` page.

Part I

CS166 Problems

2 Ivy

In this problem, you will be trying to crack a wireless encryption protocol in order to steal the encryption key.

2.1 Setup

Your dorm at Minion University is Ivy Hall. Each dorm room in the building has a router that provides internet access to occupants. The design of the internet access in Ivy Hall has some oddities, though:

- Occupants can only connect to the internet by plugging a physical ethernet cable into the router.
- Trying to save on cabling costs, the designers of the internet infrastructure in Ivy Hall decided to use wireless connections to connect each room router with a central hub (instead of the more traditional physical cables). The central hub then has a link to the internet.

In order to make sure that students cannot simply sniff this wireless traffic in order to snoop on their neighbors' traffic, the wireless traffic is encrypted with a shared key, k. This shared key is known to the central hub and to all room routers. The network setup is illustrated in Figure 1.

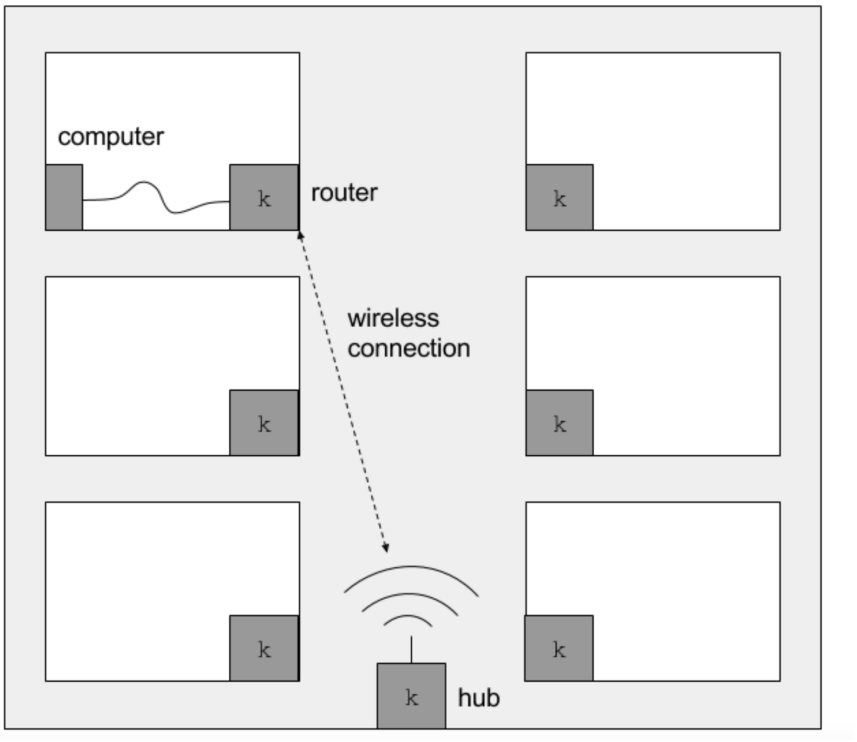


Figure 1: Network configuration for the Ivy problem.

Of course, you're training to be a spy, and snooping on your neighbors' traffic sounds like your cup of tea. But you're also lazy, and stealing the router off the wall and trying to extract the key by tapping the internal

circuitry sounds like too much work. Much easier to break the cryptography and extract the key that way (plus, the school is much less likely to find out if you don't physically steal the router).

Luckily for you, a quick web search for the router's product name reveals a product spec sheet that describes the encryption protocol used; what a great find!

2.2 Protocol

Network traffic is split up into *packets* — chunks of data that are sent over the network independently. At a high level, the router's use of encryption is pretty simple — for each packet, instead of sending that packet over the wireless link as is, it first encrypts the packet with the key, k, and then instead sends the ciphertext over the wireless link. This happens in both directions — for packets going from the routers to the hub, and also from the hub to the routers.

As we saw in class, though, encryption is not enough. If the same plaintext always produces the same ciphertext, it's trivial for an eavesdropper to decipher any ciphertext that they've seen before (for example, if Eve sends the packet "hello" over the network, and sniffs the wireless link so that she sees the corresponding ciphertext, she'll now know that whenever she sees the same ciphertext, it must be the encryption of the word "hello"). To fix this problem, each encryption also includes a random *initialization vector* (IV) so that, so long as two different IVs are used, the same plaintexts will produce different ciphertexts. The encryption function relies on the following components:

- G is a pseudorandom stream generator. Given a seed, s, it generates a pseudorandom stream of output such that, given the same seed, it always produces the same output. Additionally, if given a length parameter, n, it will only generate the first n bytes of the stream.
- R is a source of randomness. Each time R is queried, it generates a uniformly-distributed random number which is 16 bits long (that is, its outputs are uniformly distributed in $\{0, ..., 2^{16} - 1\}$).

The encryption function takes a key, k, and a message, m, and returns both the randomly-generated IV and the ciphertext:

```
ENCRYPT(k, m)
1  iv = R()              // Generate initialization vector
2  s = iv ++ k           // Concatenate iv and k
3  r = G(s, |m|)         // Generate |m| random bytes
4  c = m ⊕ r             // XOR m and r to get ciphertext c
5  return (iv, c)
```

In order to decrypt a packet, the receiver must know the IV so that it can reconstruct the random stream. Thus, the encryption function returns the IV used along with the ciphertext. When the packets are sent over the wireless link, the IV is included (that is, the IV is sent in addition to the ciphertext). When the packet arrives at the receiver, the following decryption function is used:

```
DECRYPT(k, iv, c)
1  s = iv ++ k           // Concatenate iv and k
2  r = G(s, |m|)         // Generate |m| random bytes
3  m = c ⊕ r             // XOR c and r to get plaintext m
4  return m
```

The reason that this works is that both sides know the same key, k, and since the IV is sent along with the ciphertext, they both know the same IV. As a result, they are able to reconstruct the same seed *and* the same random stream, r. Additionally, observe that a number XORed with itself is 0, and that a number XORed with 0 is itself. Thus, when we first generate the ciphertext, we do:

$$c = m \oplus r$$

And when we decrypt the ciphertext, we do:

$$\begin{aligned} c \oplus r &= (m \oplus r) \oplus r \\ &= m \oplus (r \oplus r) \\ &= m \oplus 0 \\ &= m \end{aligned}$$

And thus, we recover the original message!

There's one last part of the protocol, though. In order to prove to the hub that a router is authentic (that is, that it knows the key, k), the first thing a router does after establishing a wireless link with the hub is to send the key itself to the hub. That is, it sends $E_k(k)$.

2.3 Problem

Given this setup, you'd like to be able to extract the key. In particular, you have a few tricks up your sleeve:

- You can sniff wireless traffic, so you can see (IV, ciphertext) pairs that are sent between routers and the hub.
- You can send network traffic of your own. Combined with the ability to sniff wireless traffic, this means that you can use your router as an *encryption oracle* — given a plaintext, you can see what ciphertext it encrypts to simply by sending it over the network and sniffing the wireless traffic that's generated as a result. Using the ability to ask for the encryptions of plaintexts as a way to break a cryptographic system is known as a *chosen plaintext attack*.

2.4 Assignment

There's a binary at `/course/cs1660/student/<your-login>/cryptography/ivy/ivy` which simulates your router. It reads plaintexts on stdin (encoded as hex) and prints the corresponding ciphertexts to stdout using the format:

```
<iv> <ciphertext>
```

The first line of output is the ciphertext corresponding to the authentication packet that the router first sends to the hub. Your assignment is to, through interacting with this binary, recover the shared key, k.

We've also provided an alternate version of the ivy binary at `/course/cs1660/pub/cryptography/ivy/ivy` which accepts a key as a command-line argument. Once you've recovered the key, you can use this program to verify that the key you recovered is correct.

2.5 Handing In

Your handin should consist of two files: `KEY` and `README`. `KEY` should contain the recovered key, encoded in hex. It should contain the key on the first line and nothing else. Having a correct `KEY` file is worth 70% of the credit for this problem. Your `README` is worth 30% of the credit for this problem, and should cover the following:

- (15%) Explain in detail how your attack works. Your explanation should be detailed enough to convince somebody who has only read the specification of the protocol that your attack will work.
- (6%) Describe the vulnerability that made your attack possible, and discuss whether your attack (or a similar one) would have been possible without the vulnerability.
- (9%) Discuss how the vulnerability could be fixed.

- Propose a change to the protocol which would make your attack and any like it difficult or impossible.
- What would an attacker need to do to defeat the new design? How much more secure is this new design than the old one?

CS166 students can hand in by running `cs166_handin cryptography_cs166_ivy` from a directory containing these files. CS162 students should instead run `cs166_handin cryptography_cs162_ivy`.

2.6 CS162

CS162 students should additionally include a Go, Ruby, or Python script named `sol.go`, `sol.rb`, or `sol.py`. Your program should have the following usage (replace `script-name` with the name of your script or, if you are using Go, the name of your compiled binary):

```
./script-name <path-to-ivy-binary>
```

When run, your program should call the `ivy` binary at `<path-to-ivy-binary>` (for example, the one in your `/course/cs1660/student/<your-login>/cryptography/ivy` directory) and perform your attack against that binary. When the attack has completed, your program should print the recovered key to stdout.

Please also include instructions on how to compile and/or run your code in your `README`.

For CS162 students, the credit for this problem is broken down as follows:

- `sol.{go, rb, py}` — 40%
- `KEY` — 30%
- `README` — 30%

3 Keys

In this problem, you're faced with a block cipher encryption scheme that uses two separate encryption keys.

3.1 Setup

For a while, Minion University had been using a block cipher with a 24-bit key for all of its encryption needs — secret mission email, grades databases, and so on. However, as computers became more powerful, they started to realize that 24 bits was no longer enough; 2^{24} is around 17 million, and even a slow computer can perform that many encryptions or decryptions in a reasonable amount of time.

Not wanting to redesign the cipher entirely, some professors in the Computer Science department had a clever idea. They'd simply generate two separate 24-bit keys, and in order to encrypt a block, they'd encrypt it first with the first key, and then encrypt the resulting ciphertext with the second key; to decrypt, they'd decrypt with the second key, and then with the first.

To see why this works, recall that a block cipher takes a fixed-length message and a key, and produces a ciphertext:

$$E_k(m) = c$$

Decryption under a key is the inverse of encryption under that key — it takes the ciphertext, and produces the original message:

$$D_k(c) = m$$

The new system invented by the professors first creates a *midway ciphertext*, c', by encrypting the message with the first key, k_1, and then produces the final ciphertext, c, by encrypting the midway ciphertext with the second key, k_2:

$$E_{k_1}(m) = c'$$

$$E_{k_2}(c') = c$$

In order to decrypt, the process is simply reversed:

$$D_{k_2}(c) = c'$$

$$D_{k_1}(c') = m$$

The professors figured that two 24-bit keys would be just as good as a single 48-bit key. In order to encrypt or decrypt a message or ciphertext properly, an attacker would need to know the correct values for both keys. The number of possible pairs of 24-bit keys is 2^{48}, so the professors reasoned that in order to break the system, an attacker would have to just try all of the 2^{48} possible pairs — the exact same as if the system had one 48-bit key.

3.2 The Attack

Unfortunately for the professors, their simple approach, while elegant, doesn't work quite as well as they'd hoped. Its insecurity hinges on one critical insight...

Suppose that you were trying to break this scheme, and you were given a message, m, and a corresponding ciphertext, c. The simplest attack would be to try all of the possible key pairs, and for each, try encrypting m using that key pair, and seeing if the resulting ciphertext matched c. But suppose that, in addition to m and c, you also had c' — the midway ciphertext that's the result of encrypting m only with the first of the two keys (or, alternatively, the result of decrypting c with the second of the two keys). In this case, instead of trying all 2^{48} possible key pairs, you could instead try only 2^{25} keys — almost as few tries as if the system had never bothered to add the second key in the first place! Here's how:

- First, you'd crack the first key. Since you know m and you know c', you'd just be searching for an appropriate k_1 such that $E_{k_1}(m) = c'$. Since there are only 2^{24} possible values for k_1, you'd only have to perform at most 2^{24} encryptions.
- Second, you'd crack the second key. Since you know c' and you know c, you'd just be searching for an appropriate k_2 such that $E_{k_2}(c') = c$. Since there are only 2^{24} possible values for k_2, you'd only have to perform at most 2^{24} encryptions.
- This, in total, is at most 2^{25} encryptions (2^{23} times less work than if you'd just tried to brute force all possible key pairs!)

Of course, you probably would never get access to a plaintext/ciphertext pair and the midway cipher used to compute it. However, that doesn't mean that the idea behind this attack is entirely worthless. Maybe you can use the insight behind this attack to come up with an attack that works even if you don't have the midway ciphertext?

3.3 Assignment

You've been given a set of several plaintext/ciphertext pairs at `/course/cs1660/student/<your-login>/cryptography/keys/pairs`. This file contains lines of the form:

```
<plaintext> <ciphertext>
```

where both the plaintext and ciphertext are encoded in hex. The ciphertexts were all created using the same key pair—your challenge is to recover the key pair.

We've provided Go implementations of the cipher and stencil code for your attack in the `/course/cs1660/pub/cryptography/keys` directory. The stencil code takes care of parsing plaintext/ciphertext pairs from stdin; all you need to do is fill in the rest of the `main` function. We've also provided a `Makefile` for compiling your code—running the `make keys` command will compile your program into a binary called `keys`.

Note: While you may be able to recover the key without using all of the plaintext/ciphertext pairs we've provided you, it is possible to get false positives! Thus, when you think you have a solution, you should check it against all of your key pairs to make sure it's not a fluke.

3.4 Handing In

Your handin should consist of two primary files—`KEY` and `README`—along with any code you used to perform the attack. `KEY` should contain the recovered key pair. It should only contain a single line of the form:

```
(<key-1>, <key-2>)
```

where both keys are encoded in hex. The stencil code provides a function, `printKeyPair`, which will print the recovered key pair in the proper format. Having a correct `KEY` file is worth 70% of the credit for this problem.

Your `README` is worth 30% of the credit for this problem. In addition to explaining how to compile and/or run your program, it should also cover the following:

- (15%) Explain in detail how your attack works. Your explanation should be detailed enough to convince somebody who has only read the specification of the protocol that your attack will work.
- (6%) Describe the vulnerability that made your attack possible, and discuss whether your attack (or a similar one) would have been possible without the vulnerability.
- (9%) Discuss how the vulnerability could be fixed.
  - Propose a change to the protocol which would make your attack and any like it difficult or impossible.

- What would an attacker need to do to defeat the new design? How much more secure is this new design than the old one?

You can hand in by running `cs166_handin cryptography_keys` from a directory containing these files.

3.5 CS162

Nothing extra is required of CS162 students for this problem.

3.6 Technical Details

There's one important detail of the implementations that you should be aware of. The encryption and decryption functions in all three languages take their key arguments as integers or unsigned integers. While these can represent values greater than $2^{24} - 1$, the higher bits will be ignored. Thus, you will never need to consider key values larger than $2^{24} - 1$.

4 Transcript

In this problem, you'll exploit an insecure use of the RSA public key cryptosystem in order to forge your college transcript.

4.1 Setup

Minion University's website uses RSA to secure many of its operations. Two of these are particularly relevant to you:

- When transcripts are produced digitally, they are signed with the website's RSA private key so that their authenticity can be verified.
- When users connect to the website, RSA is used to ensure that it is in fact the website that the user is talking to (i.e., to prove to the user that they are not talking to an attacker trying to pretend to be the website).

You're applying to grad schools, and all of your applications require that you send a transcript. However, your grades aren't quite what you'd like them to be. Luckily for you, the website designer made a critical mistake when configuring RSA: the website uses the same RSA key pair for decrypting messages as it does for signing messages. This is a big no-no, and it just might let you forge your transcript...

4.2 Challenge/Response Details

When a user connects to the website, the website provides a way for them to verify that the website they're talking to is authentically the website of Minion University. This is done through the use of a *challenge/response* protocol.

In this challenge/response protocol, the user generates a random "nonce" (in cryptography, a nonce is a "number used once" — it is just a random value). Then, the user encrypts that nonce using the website's RSA public key, and sends the resulting ciphertext to the website. The website then decrypts the ciphertext with its private key and sends back the resulting plaintext. The user verifies that the plaintext matches the nonce they generated. If they match, it means that the website must be in possession of the private key corresponding to the public key, since that's the only key which can decrypt messages encrypted for the public key.

4.3 Signing Details

When the website signs important documents (including transcripts), it does so by encrypting them with its RSA private key. Recipients of the document can then verify the signature by decrypting it with the server's public key and verifying that the decrypted plaintext matches the document.

However, RSA keys are only a few thousand bits long, while documents can be any length. RSA cannot encrypt or decrypt values which are larger than the key. In order to get around this limitation, the document to be signed is first hashed, which produces a value small enough to be encrypted or decrypted in RSA. In the case of Minion University's website, the SHA-256 hash is used. The document is hashed, and the hash is then encrypted with the website's private key.

When a recipient wants to verify the authenticity of the document, it first takes the SHA-256 hash of the document. Then, it decrypts the signature with the website's public key, and verifies that the decrypted plaintext matches the hash that it computed.

4.4 Assignment

There's a binary at `/course/cs1660/student/<your-login>/cryptography/transcript/challenge` which simulates the challenge/response protocol. It accepts hex-encoded challenges on `stdin` and produces the corresponding decryptions (also encoded as hex) on `stdout`. Additionally, we've provided you a copy of the server's public key (json-encoded) in the file `/course/cs1660/student/<your-login>/cryptography/transcript/server.pub`.

We've also provided the following utilities in the `/course/cs1660/pub/cryptography/transcript` directory which you may find useful:

- `encrypt` — takes a public key file and a message and encrypts the message using the public key
- `verify` — takes a public key file, a message, and a signature, and verifies that the signature is an authentic signature for the message issued by the owner of the public key

Finally, you may find the `sha256sum` utility useful.

Your assignment is to produce two files: `TRANSCRIPT` (your forged transcript), and `TRANSCRIPT.sign` (the website's signature that the `TRANSCRIPT` file is authentic). What you put in `TRANSCRIPT` is not important, so feel free to get creative there. However, it does need to be a plain text file (no PDFs, RTFs, etc).

Once you have a signature for your transcript, you should be able to verify the signature by using the `verify` binary in the `/course/cs1660/pub/cryptography/transcript` directory:

```
verify server.pub TRANSCRIPT $(cat TRANSCRIPT.sign)
```

4.5 Handing In

Your handin should consist of three files: `TRANSCRIPT, TRANSCRIPT.sign`, and a `README`. Having correct `TRANSCRIPT` and `TRANSCRIPT.sign` files is worth 70% of the credit for this problem. Your `README` is worth 30% of the credit for this problem, and should cover the following:

- (15%) Explain in detail how your attack works. Your explanation should be detailed enough to convince somebody who has only read the specification of the protocol that your attack will work.
- (6%) Describe the vulnerability that made your attack possible, and discuss whether your attack (or a similar one) would have been possible without the vulnerability.
- (9%) Discuss how the vulnerability could be fixed.
  - Propose a change to the protocol which would make your attack and any like it difficult or impossible.
  - What would an attacker need to do to defeat the new design? How much more secure is this new design than the old one?

You can hand in by running `cs166_handin cryptography_transcript` from a directory containing these files.

4.6 CS162

Nothing extra is required of CS162 students for this problem.

Part II

CS162 Problems

These problems should be completed only by students in CS162.

Important: *These problems are significantly more difficult than the first three.* You should aim to get the first three problems done in the first week so that you can devote the second week to these two.

5 Grades

In this problem, you'll explore how statistical correlation can be a powerful cryptanalytic technique — powerful enough, in this case, to let you extract information from an encrypted grades database.

5.1 Setup

Eventually, the professors who designed the 2-key scheme that you broke in the Keys problem figured out the flaw, and increased the number of 24-bit keys to 3, which increases the effective key length of the scheme to 48 bits (can you figure out why this is?). Needless to say, without some serious hardware, you won't be cracking it any time soon.

When this new scheme was being deployed, one of the first systems to get it was the database that stores students' grades. In doing this deployment, the professors had to pick a *block cipher mode*. They chose Electronic Codebook (ECB) mode.

5.1.1 Block Cipher Modes

A block cipher like the one you attacked in the Keys problem takes fixed-size messages and produces fixed-size ciphertexts. They're very powerful cryptographic primitives, but alone they aren't enough to encrypt anything interesting, since they can't encrypt very much. A block cipher mode is a method of combining multiple uses of a block cipher in order to encrypt longer messages. There are quite a few different block cipher modes out there, but in this problem, we'll be exploring one in particular: Electronic Codebook (ECB) mode.

ECB is about as simple as a block cipher mode can get. In ECB mode, the plaintext is split up into chunks, each a single block long. Then, each plaintext block is separately encrypted using the block cipher, and the resulting ciphertext blocks are concatenated to create the ciphertext. Figure 2 illustrates ECB mode encryption and decryption.

While ECB mode is simple, it has some serious weaknesses. Most importantly, the same plaintext block will always produce the same ciphertext block. This allows for a number of attacks such as replay attacks. The attack that is relevant to this problem, though, is statistical analysis.

The problem is this: ECB mode completely leaks the statistical distribution of plaintext blocks (though it doesn't leak what those plaintexts are). In some cases, this can be a very serious weakness. This is often illustrated in terms of images, where overall patterns are generally more important than local detail. For example, consider the image in Figure 3a. If we break this image's pixels up into blocks, encrypt them using a block cipher in ECB mode, and then treat those ciphertexts as the pixels of a new image, we get the image in Figure 3b.

The fact that the professors chose ECB mode to encrypt the grades database is a big problem. In Minion University's student population of 10,000, there are bound to be some statistical patterns you can attack...

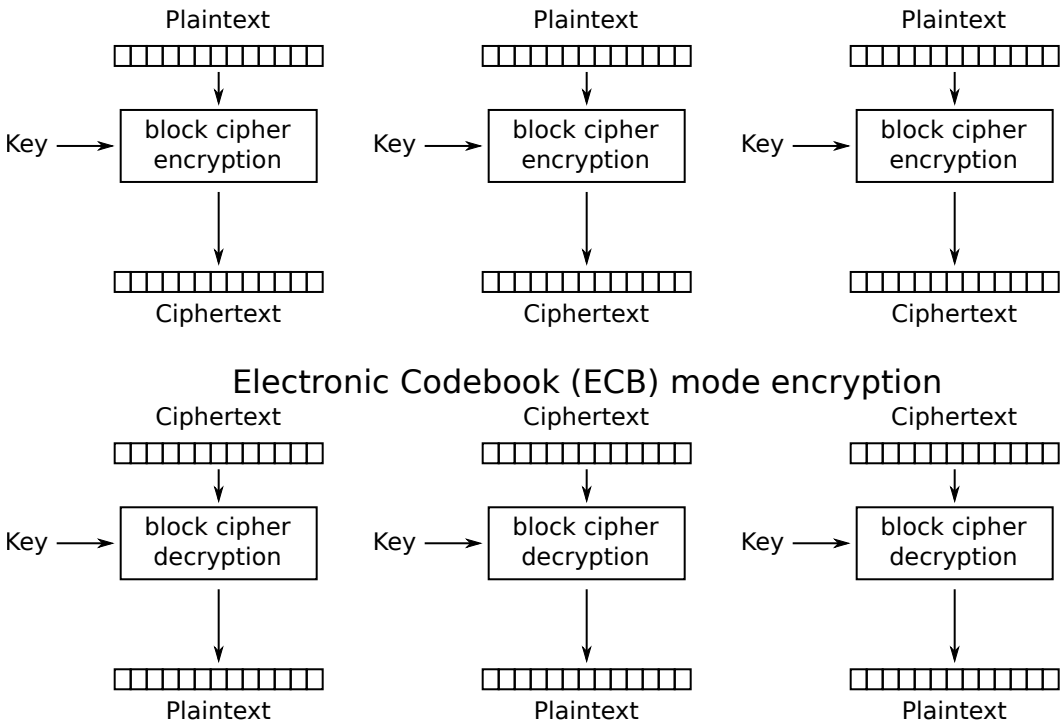
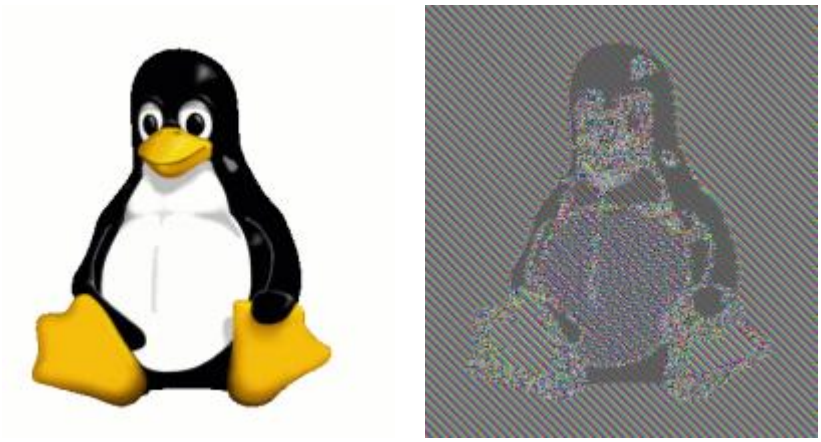


Figure 2: Symmetric encryption and decryption in ECB mode. Image by WhiteTimberwolf from Wikipedia, `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation`.



(a) (b)

Figure 3: Image encrypted with a symmetric cipher in ECB mode: (a) plaintext; (b) ciphertex. The image in part (a) shows Tux the Penguin, the Linux mascot, and was created in 1996 by Larry Ewing (`lewing@isc.tamu.edu`) with The GIMP. The image in part (b) was derived from that in part (a) by Lunkwill. Both images were downloaded from Wikipedia, `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation`.

5.2 Assignment

A copy of the encrypted grades database is available at

`/course/cs1660/student/<your-login>/cryptography/grades/grades.enc`

In plaintext, it consists of a series of newline-separated entries, each of the form:

`id:01234, grd:A`

All student IDs are randomly-generated 5-digit numbers (and are represented with leading 0s if need be). All grades are represented as a single letter; the available grades are A, B, C, and N (for "No Credit"). It happens that the ID part of the line ("`id:01234`") is 8 bytes long, as is the grade part of the line (starting at the comma and including the newline: "`, grd:A\n`"). That's the block length of the cipher that's being used. What a coincidence!

This means that the ciphertext blocks of the encrypted grades database that you have access to are laid out like this:

`<student X><student X's grade><student Y><student Y's grade><student Z><student Z's grade>...`

Thanks to statistics that Minion University is required to report under federal law, you also know the following:

- Minion University has 10,000 students
- Each student has 30 courses — there are 30 entries for each of the 10,000 students
- The distribution of grades at Minion University is approximately:
  - A — 50%
  - B — 30%
  - C — 15%
  - N — 5%

Using this knowledge of the database layout and statistics about the university, you should be able to extract a lot of information from this encrypted database even without knowing the encryption key.

5.3 Handin

Your handin should consist of a single `README` containing answers and your justifications to the following questions:

1. Given the plaintext format of the database, how many possible unique ciphertext blocks exist (not in this database, but in general)?
2. What ciphertext block corresponds to an A grade? B? C? N?
3. There's a student who's famous at the school for being the only student to ever get both As and Cs but no Bs. Exactly how many As, Cs, and Ns has this student received?

You should also turn in any code you used to analyze the data—please include a brief explanation of what your code does and how to run your code in your `README`.

You can hand in by running `cs166_handin cryptography_grades` from a directory containing your `README`.

6 Padding

In this problem, you will explore just how subtle cryptography can be — and exploit a seemingly minor leak of information to completely compromise a "secure" protocol.

6.1 Setup

It turns out you weren't the only one who figured out that the encryption on the grades database was insecure. Some members of Anonymous managed to get their hands on a copy of the database and published their findings...ouch.

Around the same time, Minion University was developing a system that would let administrators email out student grades. Vowing never to make the ECB mistake again, the professors who had originally chosen the encryption scheme for the grades database set out to find an alternative block cipher mode for use in this new email system. They came across Cipher Block Chaining (CBC) mode. While they were at it, they also switched to using a block cipher with 16-byte blocks.

6.1.1 CBC Mode

In CBC mode, each plaintext block is XORed with the previous ciphertext block before being encrypted. This way, the encryption of a given plaintext block depends on all preceding plaintext blocks — any change in the plaintext will affect all ciphertext blocks following it. This makes statistical attacks much more difficult. The diagram in Figure 4 illustrates CBC encryption and decryption.

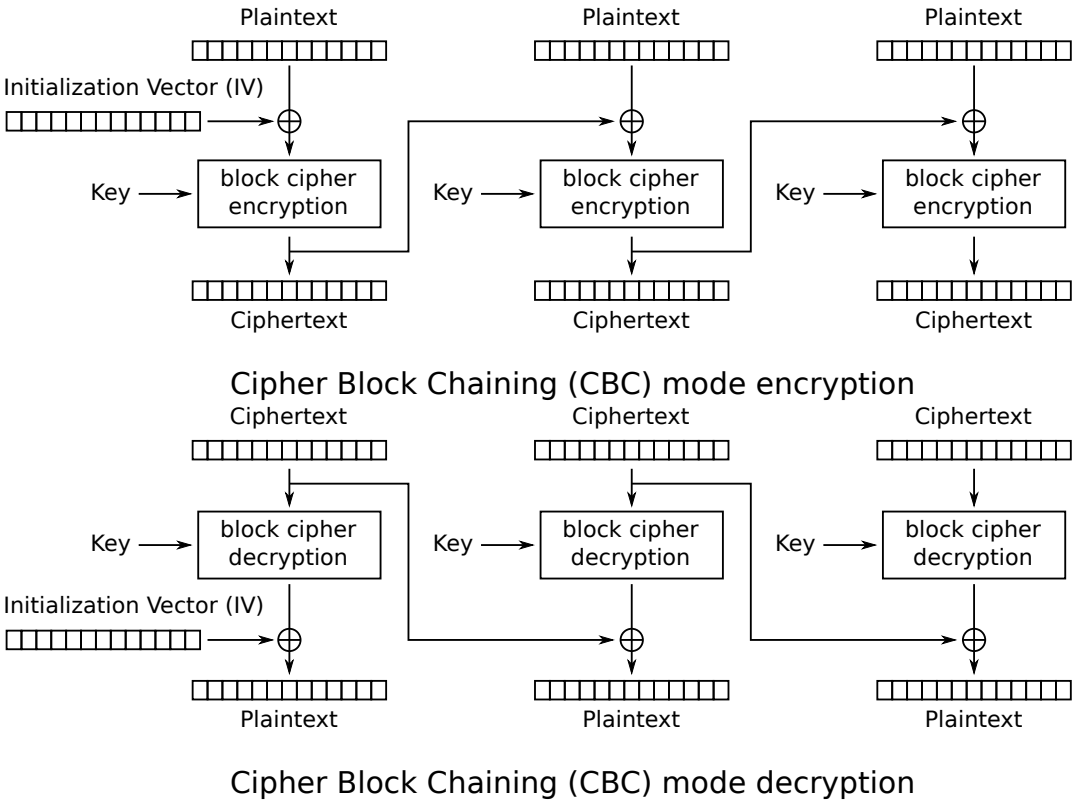


Figure 4: Symmetric encryption and decryption in CBC mode. Image by WhiteTimberwolf from Wikipedia, `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation`.

One issue with any encryption scheme with a fixed block size is that the plaintext's length may not be a multiple of the block size. A common solution is to use padding — extra data added to a plaintext (usually at the end) to make the plaintext's length a multiple of the block size. In designing the email system, the professors chose PKCS#7 padding.

6.1.2 PKCS#7 Padding

In the PKCS#7 padding scheme, plaintexts are padded with copies of a byte encoding how much padding there is. For example, if one byte of padding is used, that byte's value is 1. If two bytes are used, those bytes' values are each 2, and so on:

```
01
02 02
03 03 03
04 04 04 04
05 05 05 05 05
etc.
```

Finally, if a plaintext's length is already a multiple of the block size, an entire block of padding is added. Since the block cipher being used in this system has 16-byte blocks, a plaintext whose length is a multiple of 16 bytes would be padded with 16 bytes, each with the value 16.

6.1.3 Leak

In designing the email system, the professors were careful to use best practices and implement proper error handling. When decrypting a ciphertext, the grades server performs the following steps:

1. Make sure that there is a valid IV and ciphertext (i.e., the IV is 16 bytes long, and the ciphertext is a multiple of 16 bytes, and there's at least one ciphertext block). If anything is wrong, report an error to the user who sent the malformed input and stop processing.
2. Decrypt the ciphertext using CBC mode.
3. Check the padding on the plaintext. If the padding is valid, strip off the padding. If the padding is invalid, report an error to the user who sent the malformed input and stop processing.
4. Interpret the resulting plaintext as a command, and send any output or error messages to the user.

While this sequence of steps may seem reasonable, **it leaks a small piece of information: whether the padding at the end was correct or not.** While that may seem like a relatively uninformative piece of information, it turns out to be enough to completely break the security of the system.

6.2 The Attack

Your goal will be to use this small piece of information — whether or not the padding for a given ciphertext is correct — to forge an (IV, ciphertext) pair which decrypts to a plaintext of your choosing.

We will present an outline of the attack, and your job will be to fill in the pieces. While there's no requirement that you follow these guidelines, we highly suggest that you do, as they will point you in the right direction and guide your thinking, and make the assignment easier overall to complete.

6.2.1 Recovering Intermediate State

The first building block you'll need is the ability to recover intermediate state. "Intermediate state" is basically just CBC-speak for "the decryption of a given ciphertext block." The block cipher that is used

comprises two functions: encryption and decryption (which we'll call E and D). What we're trying to do at this stage is figure out what the decryption of a single ciphertext block is (i.e., $D_k(c)$) even without knowing the key. Consider the diagram of Figure 5, which shows part of the process of decrypting a ciphertext in CBC mode.

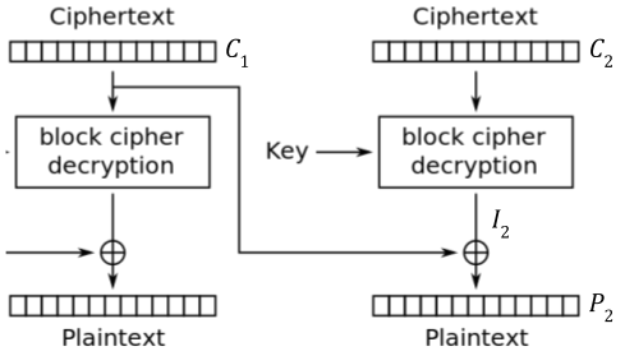


Figure 5: Intermediate state in CBC mode decryption. Derivative of image by WhiteTimberwolf from Wikipedia, `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation`

We have two ciphertext blocks: C_1 and C_2 (although C_1 here could also be the IV). What we'd like to be able to do is determine the decryption of C_2 (i.e., $D_k(C_2)$). This is the "intermediate state" I_2. Remember that you can send whatever ciphertexts you want to the server. Here's a hint: if you can somehow make it so that the final byte of the plaintext is the byte 0x01, then the padding will be correct (and you will not get an "incorrect padding" error from the server). Once you've done that, since you know the final byte of the plaintext (0x01) and the final byte of C_1 (since you chose C_1), you will know the final byte of I_2.

Once you know the final byte of I_2, you should be able to use this information to recover more bytes of I_2. Eventually, you should be able to recover I_2 in its entirety.

6.2.2 Forging Multiple Blocks

Once you've figured out how to recover the intermediate state, forging a single plaintext block is very straightforward: you simply pick an IV such that when the IV is XORed with the intermediate state, it produces the plaintext block you want. What's slightly more difficult is to forge plaintexts of arbitrary length. However, you should be able to use the ability to recover intermediate state as a building block.

6.3 Assignment

You're given a binary which will simulate the grades server at `/course/cs1660/student/<your-login>/cryptography/padding/server`. It listens for network connections on a host and port that you specify. If you run it with the `--debug` flag, it will print useful debugging information.

The client-server protocol works as follows:

- Client-to-server messages are hex-encoded IV-ciphertext pairs formatted as the concatenation of the IV and the ciphertext. In other words, for each message sent to the server, the IV is encoded in the first 16 bytes of the message; all other bytes after the first 16 bytes are the ciphertext.
- Each message sent to the server should be sent on its own line (that is, followed by a newline character).
- Data should be encoded in UTF-8 format over the socket.
- Server-to-client responses are plaintext. You do not have to decrypt them.

To avoid any server-side inconsistencies, you may need to pause/sleep briefly (0.01 seconds is sufficient) after sending each message to the server sent before attempting to read the server's response.

Your assignment is to figure out how to get the server to reveal the grades of the student whose ID is 12345. The server accepts certain commands, but you aren't sure what those commands are. All you know is that if you send a ciphertext whose plaintext is the command "help", the server will respond with a help menu. Using that help menu, you should be able to figure out where to go next. Note that sending the "help" command should only require a single ciphertext block (not including the IV), so this should allow you to test your solution to recovering a single ciphertext block's intermediate state (even if you haven't yet figured out how to forge multi-block messages).

6.4 Stencil Code

We've provided stencil code for your attack in the `/course/cs1660/pub/cryptography/padding` directory. The stencil code is in multiple languages—Go, Ruby, and Python—you can choose the one that you're most comfortable with.

The stencil code implements command-line argument validation and sets up the necessary networking code to send data to and read data from the `padding` server.

6.5 Handing In

Your handin should consist of three files: a `GRADES` file containing the grades you've extracted, a `README`, and a Go, Ruby, or Python script named `sol.go`, `sol.rb`, or `sol.py`.

The `GRADES` file should contain the exact contents of the grades report (what this means will be obvious once you've successfully extracted the grades). Having a correct `GRADES` file is worth 20% of the credit for this problem.

Your `README` is worth 30% of the credit for this problem. In addition to instructions on how to compile and/or run your solution code, it should cover the following:

- (20%) Explain in detail how your attack works. Your explanation should be detailed enough to convince somebody who has only read the specification of the protocol that your attack will work.
- (10%) Imagine that you've intercepted an (IV, ciphertext) pair sent from a legitimate client to the server, and you'd like to decrypt it. Describe in detail how you could modify your attack to allow you to decrypt the ciphertext.

The `sol.{go, rb, py}` script is worth 50% of the credit for this problem. Your program should have the following usage (replace `script-name` with the name of your script or, if you are using Go, the name of your compiled binary):

```
./script-name <host> <port> <command>
```

When run, your program should connect to the email server at the given `host` and `port`, and perform your attack in order to send the given `command`. It should then print the server's response as a result of executing that command.

You can hand in by running `cs166_handin cryptography_padding` from a directory containing these files.