CSC321 Historic Crypto Report

Benjamin Hinchliff

May 8, 2024

CSC321 Historic Crypto Report

Principles

Caesar

Offset all letters in the source text by a fixed numerical shift based on a given alphabet, usually the English alphabet. For example, a shift of 3 would shift the letter A -> C. If it passes the end of the sequence wrap around to the start of the sequence. To decrypt, simply shift the letter the opposite amount, or an equivalent number in the period of 26.

Monoalphabetic

Map all letters to another letter in the English alphabet based on a known mapping. To decrypt, simply perform that mapping in reverse. For example, one might map all As to Xs, and then to decrypt map all Xs to As.

Vigenère

Encrypt each letter by addition of the source text with the letter of a repeating key of fixed length, wrapping around the alphabet. To decrypt, one uses the repeating key and subtracts it from the encrypted text to reverse the process.

Implementation

Caesar

Iterate through a string character by character. For each character, convert it to a numerical value using `ord`, and then subtract the letter A to get the offset of the character from the start of the alphabet. Of course, a bit of consideration is required to correctly shift both upper and lowercase English characters in identical manners, since they correspond to different blocks of ASCII codes, so the case of the character must be detected beforehand, and its offset calculated by subtracting either `ord('A')` or `ord('a')`. Once the offset is calculated, one can simply add the shift and modulo by 26 to properly implement wrapping behavior.

For decryption, the same process can be followed but simply with an opposite shift of the initial shift.

This is implemented in caesar.py in `rotn`, which accepts the text and the amount to shift it by.

Usage:

```
./caesar.py -s 18 decrypted/caesar_easy_decrypted.txt
```

```python
def rotn(text: str, shift: int) -> str:
    """
    Rotate the alphabetic chracters in a given string by a known shift, with wrapping

    text - the input text
    shift - an integral value to shift the characters by (positive or negative)

    returns - the text shifted by `shift`
    """

    def shift_rel_to(c: str, to: str) -> str:
        return chr(((ord(c) - ord(to) + shift) % 26) + ord(to))

    def rotnc(c: str) -> str:
        return shift_rel_to(c, "A") if c.isupper() else shift_rel_to(c, "a")

    return "".join(rotnc(c) if c.isalpha() else c for c in text)
```

Monoalphabetic

First construct a mapping between letters of the alphabet. In my implementation, this is done by taking input from the user, where the user inputs a 26 character long strong of uppercase alphabetic characters where each character corresponds to the mapping of the equivalent place in the English alphabet. For example, a strong staring with 'W' would indicate mapping the first letter, 'A' to 'W'. This string is then converted to a Python dictionary. (I could've used an array for better performance in theory but like I'm already using python.)

After constructing a mapping, the source text can be iterated character-by-character and then mapped to the corresponding character via the mapping. If a character is lowercase, it is converted to uppercase, mapped to the corresponding character, and then converted back to lowercase. Any characters not present in the mapping are mapped directly without modification.

For decryption with a known mapping, the same process can simply followed in reverse. With my implementation this can simply be done by swapping the keys and values in the dictionary to reverse the map and performing the same process.

This is implemented in monoalphabetic.py in `map_letters`, with mapping construction done in `alphabet_to_map`.

Usage:

```
./monoalphabetic.py --alphabet ZYORBUNXLPMJTISQVGFKWCAHDE decrypted/mono_easy_decrypted.txt
```

```python
def map_letters(src: str, letters_map: dict[str, str]) -> str:
    """
    Maps the letters in the source string according to the given mapping

    src - the source string
    letters_map - a bijective function to map each letter to another in the english alphabet

    returns - the mapped string
    """
    return "".join(
        (
            letters_map[c]
            if c.isupper()
            else letters_map[c.upper()].lower() if c.islower() else c
        )
        for c in src
    )
```

Vigenère

Iterate through each character in the source string and find the corresponding index in the key based on a modulo of it's length (to cause it to repeat). After applying appropriate math to get the offsets as with the previous ciphers, add the two together to get a new offset and then convert that back to the final code point. A minor note is that a separate index must be maintained for the key than for the current character, as non-alphabetic character need to be mapped directly without incrementing the index.

Decryption is the same process, only the key is subtracted from the offset in the cypher text as opposed to adding it.

This is implemented in vingere.py in `encrypt` and `decrypt`, respectively.

Usage:

```
./vingere.py --encrypt --key MFXJT decrypted/vigerene_easy_decrypted.txt
./vingere.py --key MFXJT encrypted/vigerene_easy_encrypted.txt
```

```python
def encrypt(text: str, key: str) -> str:
    """
    Encrypt the source text using the vingere cypher

    text - source text
    key - key

    returns - encrypted text
    """
    encrypted = ""
    i = 0
    for l in text:
        if l.isalpha():
            off = ord("A") if l.isupper() else ord("a")
            p = ord(l) - off
            k = ord(key[i % len(key)]) - ord("A")
            c = (p + k) % 26
            encrypted += chr(c + off)
            i += 1
        else:
            encrypted += l
    return encrypted

def decrypt(text: str, key: str) -> str:
    """
    Decrypt the source text using the vingere cypher

    text - source text
    key - key

    returns - decrypted text
    """
    decrypted = ""
    i = 0
    for l in text:
        if l.isalpha():
            off = ord("A") if l.isupper() else ord("a")
            p = ord(l) - off
            k = ord(key[i % len(key)]) - ord("A")
            c = (p - k) % 26
            decrypted += chr(c + off)
            i += 1
        else:
            decrypted += l
    return decrypted
```

Cracking

N-Gram Fitness Scoring

The core metric all these algorithms rely on for evaluating the quality of a particular solve is based on English n-grams. I first learned of this metric from a page about Quadgrams Statistics as a Fitness Measure on Practical Cryptography

The basic concept is that given a sequence of characters the probability of all of them occurring in a given language can be calculated using the joint probability that each n-letter sequence can occur in the language, assuming the probabilities of each sequence are independent.

In other words, the probability that a given sequence of characters will occur in the English language is measured as:

$$p(s_1, s_2, ..., s_n) = p(s_1) \cdot p(s_2) \cdot ... \cdot p(s_n)$$

Where each $p(s_i)$ represents the probability of a given ngram per character, calculated as

$$p(s) = \frac{count(s)}{N}$$

where N is the total length of the corpus.

However, multiplication is a comparatively costly operation and probabilities for long sequences are at risk of growing untenably small, risking floating point precision issues. To resolve this, we can simply apply log to both sides of the equation to get the log probability, expressed as a sum of the log individual probabilities using log properties.

$$\log(p(s_1, s_2, ..., s_n)) = \log(p(s_1)) + \log(p(s_2)) + ... + \log(p(s_n))$$

For our purposes, the higher this metric, the better the solve. In our implementation, trigrams are used as they provide better results than simple bigrams, without incurring significantly higher costs. For the probabilities, we calculate them based on the frequencies in a sample of 4.5 billion characters of English text, sourced from Wortschatz by Practical Cryptography.

For completenesses sake, code is included to generate similar files in count_trigrams.py, however it wasn't used in this case due to compute and storage constraints.

Caesar

With this fitness metric in place, cracking the Caesar cipher is trivial. All possible shifts between 0 and 26 can be enumerated and the respective fitness score calculated. The one with the highest score is very likely to be the correct source text.

This is implemented in caesar.py at `decrypt_without_shift`.

Usage:

```
./caesar.py encrypted/caesar_easy_encrypted.txt
```

```python
def decrypt_without_shift(text: str) -> str:
    ngram = NGramScore("english_trigrams.txt")

    chi_squareds = [ngram.score(rotn(text, shift)) for shift in range(26)]
    best_shift = max(range(len(chi_squareds)), key=chi_squareds.__getitem__)
    eprint(f"Best Shift: {best_shift} with chi-squared: {chi_squareds[best_shift]:.4f}")

    return rotn(text, best_shift)
```

Monoalphabetic

Monoalphabetic ciphers are not so simple because they become very difficult to brute force given that there are 26! possible substitutions for a given piece of text.

As a result, a smarter algorithm needs to be devisied. Our implementation uses a variation of Markov-chain Monte-carlo (MCMC) described by dCode in their page on monoalphabetic ciphers.

The basic principle is why brute force when you can just try to randomly stumble in the right direction.

1. Start from an initial mapping, we use a direct mapping of substituting each character to the same one.
2. Calculate the fitness of this given substitution, the current best fitness.
3. Swap two characters in the mapping randomly.
4. Calculate the fitness again, if it's better than the current best fitness, then it becomes the new best fitness.
5. Calculate how close the fitness is to the current best fitness, and the close it is the more likely we are to randomly accept the solution.
    - This allows for exploratory behavior instead of purely exploitative behavior, as otherwise the algorithm will very often converge to false maxima.
6. Repeat from step 3 ~~until the solution has converged~~ (not implemented) or we get bored.

Using this algorithm, and a little luck, we are able to reliably decode arbitrary monoalphabetic ciphers. (If you run this and it doesn't work the first time, try again, as it can end up converging to local minima on occasion.)

Implementation is in monoalphabetic.py at `decrypt_without_alphabet`.

Usage:

```
./monoalphabetic.py encrypted/mono_easy_encrypt.txt
```

Vigenère

Now that we've created an algorithm to explore large search spaces like this, we're already very close to a way to crack vigenère ciphers. However, to make this method more effective, it would be useful to know (or at least have a good idea of) the length of the key of the cipher beforehand.

This can be done by exploiting the periodic behavior of the vigenère cipher, due to the limited length of the key. A simple way to do this is using the index of coincidence, a measure of how often characters coincide in a given text. In other words, the more a single character occurs in a text, the higher it will be.

From Wikipedia, for the English alphabet, this is calculated as:

$$IC = 26 \cdot \frac{\sum_{i=1}^{c} n_i(n_i - 1)}{N(N-1)}$$

Where n_i is the count of a given letter in the text and N is the total letters in the text.

Sine we know the key is periodic, we know that every n characters we would expect that the same character would occur more often than others. Using this fact, we can slice the text based on the period and then inspect the ICs of those slices. The higher they are, the more likely that this is the true length of the key.

Using this fact, we can calculate these ICs for each given key length less than a given maximum key length L_max and then accept the key length with the highest sum of the ICs, as seen below.

```
def index_of_coincidence(text: str) -> float:
    counts = Counter(text)
    numerator = sum(c * (c - 1) for c in counts.values())
    total = sum(counts.values())
    return 26 * numerator / (total * (total - 1))


def slice_by_period(text: str, period: int) -> list[str]:
    slices = [""] * period
    for i in range(len(text)):
        slices[i % period] += text[i]
    return slices


def guess_period(text: str, max_period: int) -> int:
    text = clean_text(text)
    best_period = 1
    best_ioc = None
    for period in range(1, max_period):
        slices = slice_by_period(text, period)
        total = 0
        for i in range(period):
            total += index_of_coincidence(slices[i])
        ioc = total / period
        if best_ioc is None or ioc > best_ioc:
            best_period = period
            best_ioc = ioc
    return best_period
```

Once we have a key length, we can use effectively the same algorithm as before:

1. Assume the key is all 'A's of the key length.
2. Randomly chose a character from the key.
3. Enumerate all possible 26 letters for that character.
4. If any produce decrypted texts with higher fitness than the current best, accept it.
5. Repeat from 2 until we get bored.

Funnily enough, this actually has much better convergence properties than the monoalphabetic cipher cracking algorithm, but that might just be down to the quality and tuning of my implementation.

Implementation can be found in vingere.py at `decrypt_without_key`.

Usage:

```
./vingere.py encrypted/vigerene_hard_encrypted.txt
```

```
def decrypt_without_key(text: str) -> str:
    ngram = NGramScore("english_trigrams.txt")

    key_length = guess_period(text, args.max_key)
    eprint(f"key length: {key_length}")

    best_key = ["A"] * key_length
    best_score = ngram.score(decrypt(text, "".join(best_key)))
    epochs_since_improvement = 0
    for _ in range(args.max_key * 5):
        key = best_key[:]
        i = random.randrange(key_length)
        for c in range(26):
            key[i] = chr(ord("A") + c)
            solve = decrypt(text, "".join(key))
            score = ngram.score(solve)
            if score > best_score:
                best_key = key[:]
                best_score = score
                epochs_since_improvement = 0
                eprint(f"new best scrore: {best_score}")
            eprint(f"key: {''.join(key)}")

        if epochs_since_improvement > MAX_EPOCHS_TO_IMPROVE:
            break
        epochs_since_improvement += 1
```

Keys

Caesar

File	Shift
caesar_easy_2_encrypted.txt	11
caesar_easy_encrypted.txt	18
caesar_hard_2_encrypt.txt	23
caesar_hard_encrypt.txt	6

Monoalphabetic

File	Encrypt Mapping	Decrypt Mapping
mono_easy_encrypt.txt	ZYORBUNXLPMJTISQVGFKWCAHDE	WEVYZSRXNLTIKGCJPDOMFQUHBA
mono_medium_encrypt.txt	VLMEXBPHSTWGFJCIKDYARNOQUZ	TFORDMLHPNQBCVWGXUIJYAKESZ

Vigenère

File	Key
vigerene_easy_encrypted.txt	MFXJT
vigerene_medium_encrypt.txt	DBDUBDFBF
vigerene_hard_encrypt.txt	TOBSLTBSFSFBM

Demos

Caesar

https://asciinema.org/a/p17udVsMgRVDV2La0g2l8cIoo

Monoalphabetic

https://asciinema.org/a/OcJW10GARQ7HyMhCuOTXNqJri

Vigenère

https://asciinema.org/a/YrfzodDkgnKNC6i5A1SJDL11T

Questions

Caesar

1. The simple key is a good advantage, and the fact it can be easily computed by hand. However, it is also very simple to crack, even by hand.
2. No, it's easy to crack and I include a handy program to do that as well.
3. This one was easy, though I initially used chi-squared with frequency analysis instead of n-gram fitness, but later switched it for consistency.

Monoalphabetic

1. It's quite hard to crack, for what it is, since not that many assumptions can be made about the mapping and there are a lot of possibilities. It suffers, however, from the fact that the mapping is hard to remember and encode, often leading to shortcuts that make it substantially less secure. In addition, it isn't really *that* hard to crack, and there's no way to make it more secure with a longer key or anything.
2. No, it's a pain to use and once again I have a program to crack it on the same hard drive I'd have files on.
3. Learning about n-gram fitness and MCMC was fun.

Vigenère

1. It's more difficult to crack, and can be made more secure with longer keys (though it eventually devolves into a OTP). However, it's periodic behavior can be exploited to determine the key length at which point cracking it becomes quite easy.
2. No, for the aformentioned ease of cracking. Also, program on same hard drive again.
3. IC is pretty cool it's surprising how well it works.