Cryptanalysis of Purple, Japanese WWII Cipher Machine

Barjol Lami, Gledis Kallco, Nicholas Guo, Sean Shi

May 2019

Abstract

Cipher machines have been the most frequently used method for safe communication during World War II. Some war messages of very high importance were sent encrypted through these machines, therefore they were designed to be very secure in order to prevent the decryption by the rivals. This paper will focus on one cipher machine, Purple, which was used by the Japanese. In this paper, we give a detailed overview of Purple as well as go though a couple attacks we implemented to break it.

1 Introduction

Purple is the codename used by American cryptanalysts for the *97-shiki-obun In-ji-ki* (Alphabetical Typewriter '97), the machine used by the Japanese during World War II [1]. The Japanese used this machine in order to encrypt important diplomatic and military messages. For example, Purple was used to encrypt the *14-part message*, a 5000 characters long message sent from the Japanese Government to its Washington embassy, which broke off the negotiations between the Japan and United States [1].

Purple is also called *Type B Machine* since it was the successor of a Japanese cipher machine called Red (also called Type A Machine), from which it inherited many of its properties [6]. In their design of Purple, the Japanese fixed some of the security flaws present in the Red machine.

During the war, US cryptanalysts were able to figure out how Purple worked, and also built Purple simulators to help with the cryptanalysis. For our project, we assume we know exactly how Purple works, and have access to a simulation of Purple in Python.

2 System Overview

In this section we will explain the structure of the machine. We will use the version that the United States cryptanalysts used as a replicate of the original one, as the Japanese made sure that no original version of Purple used by them was ever found.

2.1 Elements of Purple Machine

Purple has 3 main components: input plugboard, permutation switches and output plugboard.

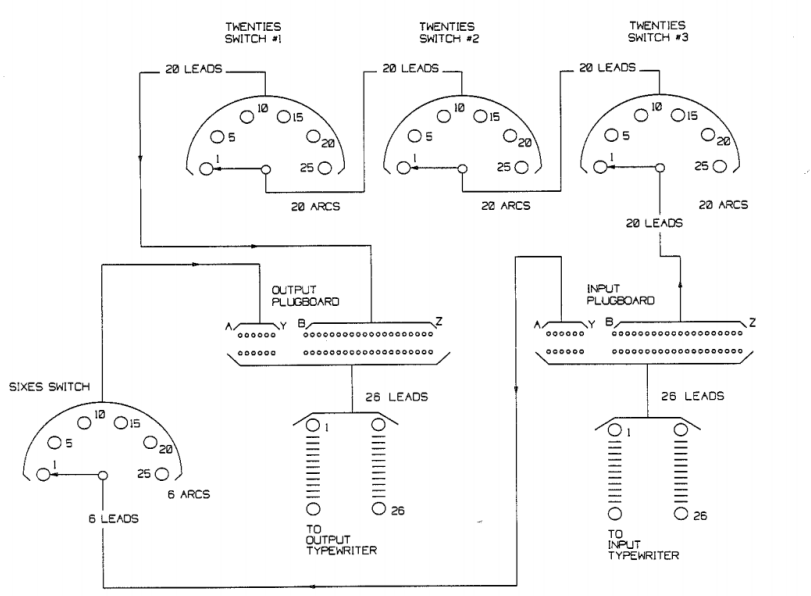


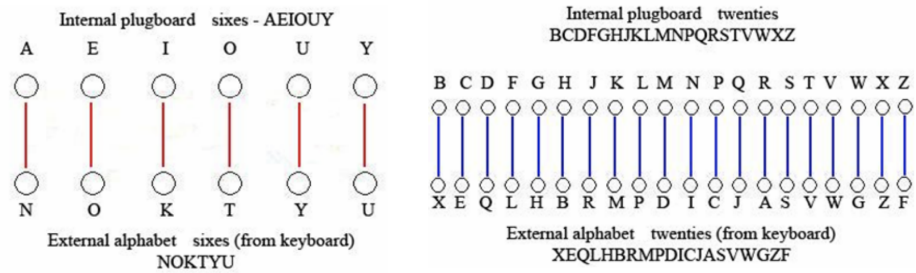
Figure 1: Inner structure of Purple [1]

2.1.1 Input Plugboard

The input plugboard is composed of two parts, the internal and the external alphabets. The external alphabet is where the input from the typist comes from (the keyboard). Each of the letters from the external alphabet is then mapped to one of the letters in the internal alphabet. This is done manually so every possible permutation of the alphabet can be a valid mapping. The internal alphabet is what the machine will use to encrypt. A very important aspect of this alphabet is that it's divided in two parts, the sixes which are the vowels and the twenties which are the consonants. As shown in Figure 2, any of the letters can be mapped to either the sixes or the twenties in the plugboard. This is a significant improvement from Purple's predecessor, the Red machine, which could only encrypt a vowel to a vowel and a consonant to a consonant [6]. However the 6-20 split makes the cipher machine more vulnerable as will be explained in section 3.

2.1.2 Switches

After a letter goes through the input plugboard, it can be encrypted in two different ways depending on whether it maps to one of the sixes or one the twenties. Sixes get permuted through a switch called the *sixes switch*. It has 25 possible positions, which means 25 possible permutations of the sixes out of $6! = 720$ that is the total space of permutations. The twenties have a bigger space of possible permutations. They get permuted through 3 consecutive switches called *twenties switches*, each of which has 25 possible positions.
Combined, 3 twenties rotors can produce 25^3 possible permutations for the twenties. Every time a letter gets encrypted by the machine, the sixes switch and one of the twenties switch will change the position as will be described in section 2.2. By doing this the machine generates a new alphabet permutation for the next letter to be encrypted.



(a) Mapping of sixes

(b) Mapping of twenties

Figure 2: Input plugboard mapping [5]

2.1.3 Output plugboard

The switches permute the input from the internal alphabet of input plugboard to the internal alphabet of the output plugboard. From the internal alphabet of the output plugboard, letters then map to the output typewriter through the same identical mapping as the input plugboard. This is the way Japanese chose to do this mapping however the input and the output plugboards are independent so they could also have different mappings [1].

2.2 Stepping switches

The input and the output plugboards will be fixed for a specific encryption. The machine only uses the rotors to change the permutation alphabet for every letter. Each of the positions of the switches does a unique permutation of its corresponding input space. The permutation by each of the positions is constructed in a manner such that no two of the 25^3 permutations from the twenties are the same.

Switch Positions			
Sixes	Twenties #1	Twenties #2	Twenties #3
21	1	25	5
22	2	25	5
23	3	25	5
24	4	25	5
25	4	25	6
1	4	1	6
2	5	1	6

Sample Switch Position - #1: fast, #2: middle, #3: slow

Figure 3: Stepping switches [5]

The sixes switch advances after every letter gets enciphered, meaning it will start repeating permutations for every 25 characters enciphered. The twenties switches advance

based on the labels they have. Each of them can get labeled as "fast", "middle" and "slow" so that gives 6 different possible labeling. Labeling determines which of them moves at a specific moment. Based on their names, the "fast" switch will advance more frequently than the "middle" one, which will advance more frequently than the "slow" one. More specifically, the "slow" switch advances every time the sixes reaches position 24 and the "middle" switch is at position 25. The "middle" switch advances every time the sixes switch is at position 25 and the "fast" switch is at position 25 and the "fast" switch moves every other time. An example of this movement is given at Figure 3.

2.3 Example of encryption by Purple

In Figure 4 we can see the encryption of a letter that is mapped to one of the sixes. On the left, there's the input plugboard. In our case it takes K as external input from Typewriter and outputs U as internal input. On the right of the figure we see the output plugboard which gets E from internal output, and outputs S. In the middle, the sixes switch permutes letter U depending of the permutation in position 3 of the 25-position switch and outputs E. For the Typewriter, this is equivalent to encrypting K to S, due to the plugboards.

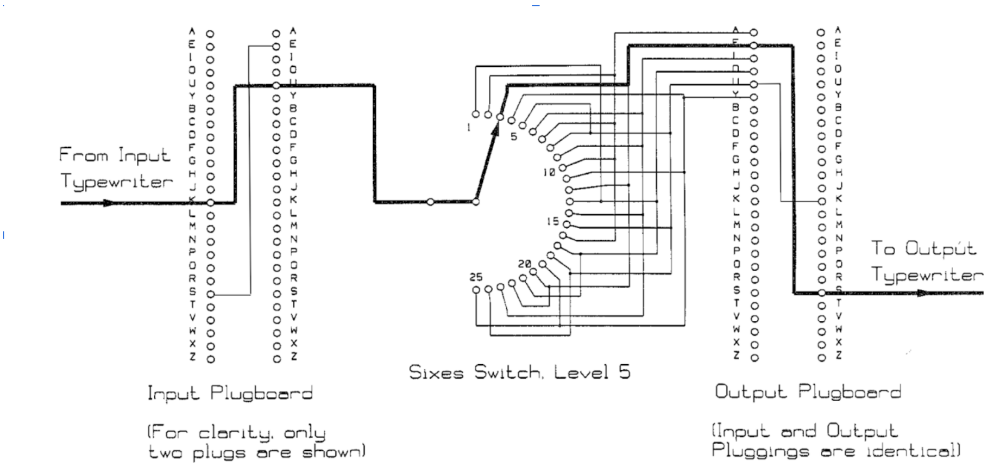


Figure 4: Encryption of a letter from the sixes [1]

2.4 Key Space

If we analyze the structure of the machine, we see that there are 25^3 possible starting positions for the twenties switches. In addition, are 6 different ways we can label them as "fast", "middle" and "slow". There also 25 different starting position for the sixes. Considering also the 26! possible permutations of the alphabet that depends on the wiring the user decides to use in the plugboard, in total we get: $6 \cdot 25^3 \cdot 25 \cdot 26! \approx 9.45 * 10^{32}$ possible keys. This is a considerably big number to be brute forced, especially for the time this machine was used. But taking advantage of some weaknesses the machine has and doing frequency analysis on it's outputs, the complexity of the problem reduces significantly as we can see in the following sections.

3 Weaknesses

Even though Purple shows a high complexity for one to decipher, it has some weaknesses in its structure that an adversary can take advantage of. The Japanese built the machine based on its previous version Red, where the 6 vowels were permuted within themselves, and used the same separation for the internal plugboard. They added the permutation from the external alphabet to the internal plugboard, but that turned out to not add complete security to the fact that these 6 letters are treated separately, especially since American had previous messages from Red and knew about the 6-20 split.

The partition of the keys into 6-20 gives a limited number of configurations possible. For a given letter there are not 26 different possibilities to be permuted to, but only either 6 or 20 due to the connection built in the machine configuration. This connection drops the number of possible alphabet permutations to $6! \cdot 20!$, instead of $26!$ if every letter could be permuted to every other alphabet letter. An adversary can first identify the letters that belong the sixes in a relatively short time, and later has to deal only with 20! possibilities, which is considerably smaller than 26!. Moreover, the number of configuration drops from $6 \cdot 25^4 = 2,343,750$ in total to $6 \cdot 25^3 = 93,750$ after identifying what goes to the sixes.

Another issue with Purple is the limited number of rotors this machine uses is something else that adversaries can exploit. Since there is only one 25-position switch in the sixes and three 25-position switches in the twenties, the alphabet permutations used by the machine repeat after at most $25 \cdot 25 \cdot 25 = 15,625$ inputs, which is a value not that unreasonable for an adversary to brute force.

Based on the weaknesses mentioned in this section, we developed two attacks taking advantage of these weaknesses that run in a reasonable time.

4 Attacks

We implemented two attacks: a known-plaintext attack and a ciphertext-only attack that finds the key through hill climbing. We use a Purple simulator by Brian Neal as our simulator [3].

4.1 Known-plaintext Attack

For the known-plaintext attack, we are given a plaintext and ciphertext pair, and are tasked with finding the entire key. The motivation behind this attack is that the Japanese historically used very few rotor and plugboard settings compared to the entire key space, and that in practice, obtaining a plaintext-ciphertext pair is not that unlikely. For instance, operational errors may occur that give such a pair. For example, suppose some Japanese operator receives a message encrypted with an already broken cipher, decrypts it, and re-encrypts it with Purple. This gives us a plaintext, from the already broken cipher, and a ciphertext, from the Purple-encrypted message.
The known-plaintext attack works in three stages:

1. Finding which letters are wired to the sixes and twenties rotors
2. Finding the sixes rotor position and the specific plugboard setting for the sixes
3. Finding the twenties' rotor positions, which rotors are fast and medium, and the specific plugboard setting for the twenties

Let our plaintext be $\mathbf{x} = x_1x_2\cdots x_n$ and our ciphertext be $\mathbf{y} = y_1y_2\cdots y_n$, where the x_i and y_i are individual characters. Let the unknown input plugboard be a function f where if some letter a is wired to b we have $f(a) = b$. Then the output plugboard is f^{-1}. Let $a_i = f(x_i)$, i.e. $\mathbf{a} = a_1a_2\cdots a_n$ is the message after passing the input plugboard. Let $b_i = f(y_i)$, i.e. $\mathbf{b} = b_1b_2\cdots b_n$ is the message after passing the rotors. Then passing the message $\mathbf{a}$ through the rotors gives $\mathbf{b}$. Also, for any i, either both a_i and b_i are wired to the sixes rotor or they are both wired to the twenties rotors.

A note about the output plugboard being f^{-1} as opposed to f: we found conflicting documentation as to which one it is, specifically [1] suggests f but [3] suggests f^{-1}. Since we are using [3] as our simulator, we will assume the output plugboard is f^{-1}.

4.1.1 Finding which letters are wired to the sixes and twenties rotors

To find these groups, we make the following observation. Let i and j be two indices such that $x_i \neq x_j$ and $y_i = y_j$. Then $a_i \neq a_j$ but $b_i = b_j$. Then, b_i and b_j are both wired to sixes or both wired to twenties, so a_i and a_j are both wired to sixes or both wired to twenties. Hence, either x_i and x_j are both in the sixes or both in the twenties. Repeat this step for all pairs of i and j that you find. Note that this step works no matter what the output plugboard is in relation to the input plugboard, since the only thing used is $b_i = b_j$.

This partitions the letters into multiple groups where all the letters in each group are either all in the sixes or all in the twenties. If there are sufficient letters in the plaintext, there will (usually) be two groups, one corresponding to the sixes, one corresponding to the twenties. Otherwise, if there are more than two groups, it is possible to figure out how to combine the groups to generate plausible sixes/twenties splits.

4.1.2 Finding the sixes rotor position and the specific plugboard setting for the sixes

For this step, we only consider the letters in each text that correspond to the sixes. Note that the twenties rotors and the plugboard wiring for the twenties do not affect the encryption of the sixes at all. Hence, only the sixes rotor position and plugboard setting matters. Then, there are only $6! \cdot 25 = 18000$ different distinct keys to check. In our algorithm, we just search through all distinct keys, which is fast enough.

4.1.3 Finding the twenties' rotor positions, which rotors are fast and medium, and the specific plugboard setting for the twenties

For this step, we assume we know the sixes rotor position and plugboard settings. It turns out that the sixes plugboard settings are not needed. This step works by running through all possible initial arrangements of the twenties rotors, and making guesses for specific wires in the plugboard and using the rotor settings to figure out more wires in the plugboard.

The subroutine that guesses a plugboard given the rotor settings is discussed in more detail below. The subroutine will actually return all possible plugboard settings consistent with the rotor settings.

Start with an empty plugboard. Make a guess for one of the wires in the plugboard, say this corresponds to guessing $f(m) = n$. Then, if we have $x_i = m$, $a_i = n$, and we can figure out b_i through knowing the rotor position. Then, $f(y_i) = b_i$. This corresponds to another wire in the plugboard. We also can detect if a guess makes a plugboard inconsistent, i.e.

two inputs are wired to the same output. Keep on making guesses as long as the plugboard remains consistent and is incomplete.

An optimization can be made with calculating the permutations given by the rotor positions by simply precomputing these permutations and storing them.

The entire attack, including the precomputation for the twenties rotor permutations, takes roughly 5 minutes on a modern machine. In addition, the first step requires messages of only length 100, and the second and third steps only require messages of roughly length 50 to uniquely determine the key.

4.2 Hill Climbing Attack

In this section we go over our hill climbing attack. We require only about a thousand characters of ciphertext to recover the key that generated them using this approach.

The hill climb algorithm is an iterative optimization technique that greedily finds local maxima of functions by taking some arbitrary solution, then updates that solution to some neighbor with a higher value at each step [1]. Specifically, we repeat the following process. Given x and $f(x)$, we evaluate different values of $x+\delta$, where δ is a small change in different dimensions. For any δ such that $f(x+\delta) > f(x)$, we update x to $x+\delta$.

In our attack we use the hill climb algorithm twice: first to find the position of the sixes switches along with the sixes permutation, and second to find the position of the three twenties switches, the speeds of them, and the plugboard permutation of the twenties. Because of the twenties and sixes switch divide, we are able to break the hill climbing into these two pieces such that we can gain some signal by variating one set of parameters, then with the other.

The scoring function we will use for the hill climb attack is bigram frequency. A bigram is simply a pair of consecutive alphabetic characters that occur frequently in the suspected plaintext. Thus if we obtained a list of bigrams that occur frequently in the corpus of plaintext we wish to decrypt, we know that a decryption that contains more bigrams is more close to the actual decryption than one done by a key that contains less bigrams. Note that the list of bigrams that should be used is dependent on what we think the messages are. In WWII, the Japanese sent messages in Japanese, which was represented by two or three letters per Japanese character. We can use the most common characters to generate bigrams. For us, we were trying to decrypt English text, so we found a list of commonly used English bigrams [4]. And lastly, the way we would evaluate a certain set of permutations and rotor positions was by using that to decrypt the ciphertext and counting the total number of matching bigrams.

To carry out the attack, we first iterated over each possible sixes switch. Then, for each sixes switch, we initialized our hill climb with some random sixes permutation, for example $FVCOIL$. Then, we evaluated the bigram frequency using our switch position and the given permutation (picking random positions / permutations for the twenties). Next, we tried different neighbors of the sixes permutation. The neighbors we chose were permutations that could be generated from the original by swapping a pair of characters. This includes things like $AVCOIL$, where we swapped off the F, and $VFCOIL$, where we swapped the V and F. Then, we repeated this process until no neighbor gave a higher bigram frequency score, which we assume to be the local maximum for the particular sixes switch position. Since local maxima may not be global maxima, we add some additional random restarts to make it much more likely we hit a global maximum for each hill climb. Then, we look for the highest bigram frequency for each of the different switch positions,

then assume that switch position as well as the permutation are correct.

Then, with this ground truth info of what the sixes switch and permutation should be, we repeat the exact same process with the twenties switches and permutation. We try each switch position / speed combinations and hill climb over the possible permutations.

4.2.1 Implementation and Performance

To simulate the purple machine and carry out our attacks, we used the Purple module from Python [3]. The module takes in a set of starting switch positions and the plugboard permutation and allows us to encrypt or decrypt messages using those settings. To set up our attack environment, we generated some random set of switches and permutation, then used that to encrypt some random block of English text from an article found online. The attack itself was written in Python3.

The attack in total took about 8 hours to complete on a macbook pro. The main bottleneck came from the fact that for the twenties hill climb, we had to perform a hill climb for every switch position and speed combination, a total of $25^3 * 6 = 93750$ times. The reason that the switches themselves could not be part of the hill climb was that each switch position corresponded to a totally different hard-coded permutation. Thus modifying a switch position by one was actually a huge change in comparison to swapping two characters in the plugboard permutation.

One optimization we used was to use multiprocessing to speed up computation. Since each separate hill climb is independent of another, we were able to take advantage of using the 8 cores on our Macbook Pro to concurrently hill climb. Another thing we realized was that using the full text we had was more than necessary to derive the switch positions and plugboard permutation. By reducing the amount of ciphertext to decrypt (we can do this since there is a one to one correspondence from ciphertext to plaintext), we were able to reduce the amount of time things took significantly. A text length of 1000 was about the shortest we could use to still crack the machine.

5 Conclusion and Future Work

In this project, we successfully managed to break Purple only given a ciphertext, however, we think that the security of the machine is very good considering the years when it was used. We find it pretty impressive that the American cryptanalysts were able to decrypt messages with it [2].

We were unsuccessful in trying to figure out a way to hill climb over the switches which would theoretically reduce the running time to breaking the machine by a big factor, so that might be a objective to achieve in a future work on Purple. Also, we think that implementing our code in C++ instead of Python would also reduce the running time by at least a factor of two.

Finally, we know that this machine could have been much harder to break if the letters were not divided in 6-20. This could have been avoided if they would wire the machine so that all 26 letters were permuted to every other 26 letters, connecting those the same chain of rotors. Also adding extra rotors would increment the complexity of the machine as it would increase the total number of permutations.

6 References

1. Freeman, Wes, Geoff Sullivan, and Frode Weierud. Purple Revealed: Simulation and Computer-Aided Cryptanalysis of Angooki Taipu B. Cryptologia 27, no. 1 (January 1, 2003): 143. $https://doi.org/10.1080/0161-110391891739$

2. Friedman William F. "Preliminary Historical Report On the Solution of the "B" Machine" $http://cryptocellar.org/files/PURPLE\_History.pdf$

3. Neal, Brian. Purple: Simulation of the WW2 Japanese PURPLE Cipher Machine. (version 0.1.0). OS Independent, Python.
$https://bitbucket.org/bgneal/purple/$

4. Practical Cryptography. $http://practicalcryptography.com/cryptanalysis/letter-frequencies-various-languages/english-letter-frequencies/$

5. Shikhare, Aparna. Cryptanalysis of the Purple Cipher Using Random Restarts. Master of Science, San Jose State University, 2015. $https://doi.org/10.31979/etd.tcqp-x6sz$

6. Type B Cipher Machine. In Wikipedia, May 10, 2019. $https://en.wikipedia.org/w/index.php?title=Type\_B\_Cipher\_Machineoldid=896491081$