Breaking Enigma

Lynda Tang, Nayoung Lee, Sophie Russo

Project Overview

Our project was attempting to crack the *Enigma*, a rotor machine used by Nazi-Germany to encrypt messages during World War 2 (WWII). In doing so, we took into account various mechanical exploits as well as human error, the latter of which was based on actual mistakes made by the Germans in WWII.

We found that given messages with certain properties, we were able to reduce the keyspace by a factor of up to 10^{15} using these techniques, which gave us a runtime of approximately 3 minutes rather than the worst-case 300+ years it would take a single machine to brute force a keyspace of the Enigma's magnitude.

Background

Enigma Overview

The Enigma is a mechanical encryption machine used in WWII. It consists of 4 parts: keyboard, lampboard, rotors, and plugboard (shown below in Figure 1).

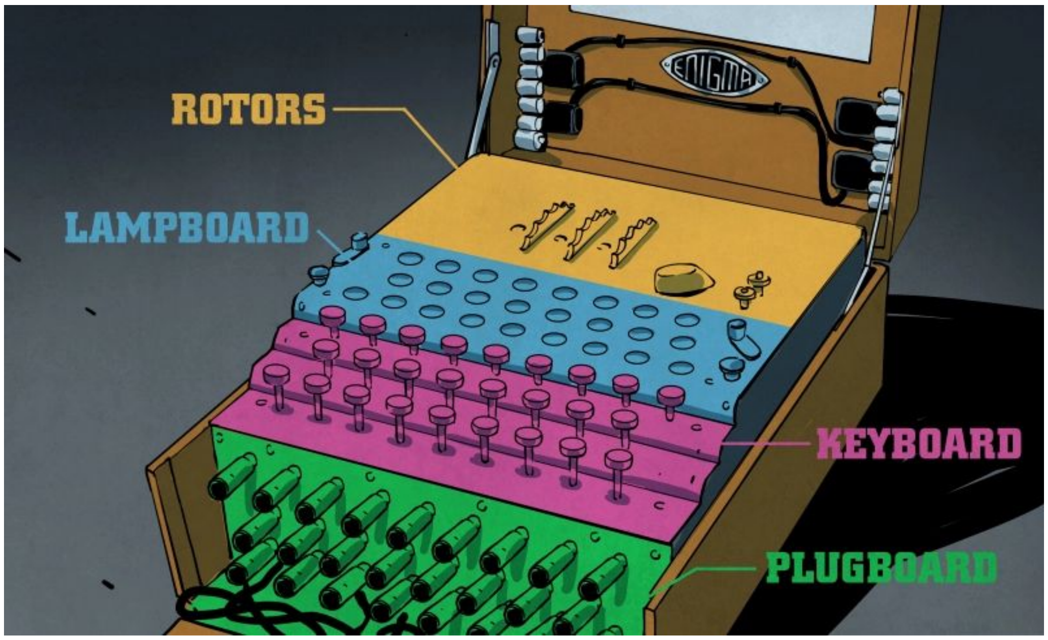


Figure 1: The basic Enigma setup[1].

Keyboard and Lampboard

The keyboard represented the input and the lampboard represented the output. An operator would type in the plaintext they needed to encrypt or the ciphertext they needed to decrypt into the keyboard, and the corresponding enciphered or deciphered letter would light up on the lampboard.

Rotors

The rotors were the key components to the Enigma machine and they each had their own internal wirings between letters, which depended even on the rotor's rotational position. Each rotor had 26 positions, for each letter of the alphabet, and the letter on the top of the rotor determined the "position" of the rotor. Each time a letter was inputted, the rightmost rotor would rotate by one position, and every time the rightmost rotor completed a full rotation, the middle rotor would rotate by one position, and this process extended to the leftmost rotor as well. Because each time a rotor rotates, it creates a new internal mapping of letters, each letter would essentially be encrypted differently; for example, if a message had two of the same letters "AA", the Enigma would not encrypt those two A's to the same ciphertext because the rotors would rotate by one position.

Plugboard

The plugboard was added to the Enigma machine later on in the war, to drastically increase the difficulty of cracking the Enigma. Plugboards used a hole for each letter of the alphabet and up to 13 cables. Each cable could have one end plugged into one letter and the other end plugged into another letter. This would switch the electrical impulses for the two letters on both input and output. For example, without the plugboards, let's say the letter "A" encrypted to the letter "E", the letter "F" encrypted to "B". Then, we added in a cable into the plugboard connecting "A" to "F" and "B" to "C". Then, our inputted letter "A" would now map to "F" first, which get encrypted to "B", and before "B" is outputted, it would get switched to "C", meaning "A" would now encrypt to "C", instead of "E".

Encryption and Decryption

The "key" of the Enigma consisted of the following:

- The order of the rotors
- The starting (or initial) positions of the rotors
- (Military only) The 10 plugboard settings

To encrypt: An operator would first set the key of the Enigma, and type the message. The operator typed each plaintext letter of the message into the keyboard and would see the encrypted letter outputted by the lampboard. The internal wirings of the Enigma follow the diagram shown in Figure 2 below. After the operator pressed a key on the keyboard, the electrical impulse for the letter would arrive at the plugboard, which would switch the impulse to whatever other letter the inputted letter was connected to. After the plugboard, the impulse travel through the rotors, getting mapped to whatever letters the rotors dictated. (In Figure 2, there is an additional 4th “static” rotor wheel which was added in by the German Navy later in the war, in order to heighten the security of the German Navy. However, to remain compatible with 3-rotor Enigma-encrypted messages, the 4th rotor wheel had an option to be “static”, where each letter maps to itself, and that is what is happening in this case.) Then, it reaches a reflector which also switches the letter before the impulse travels back through the rotors and back to the plugboard. The resulting encryption is then finally outputted through the lampboard.

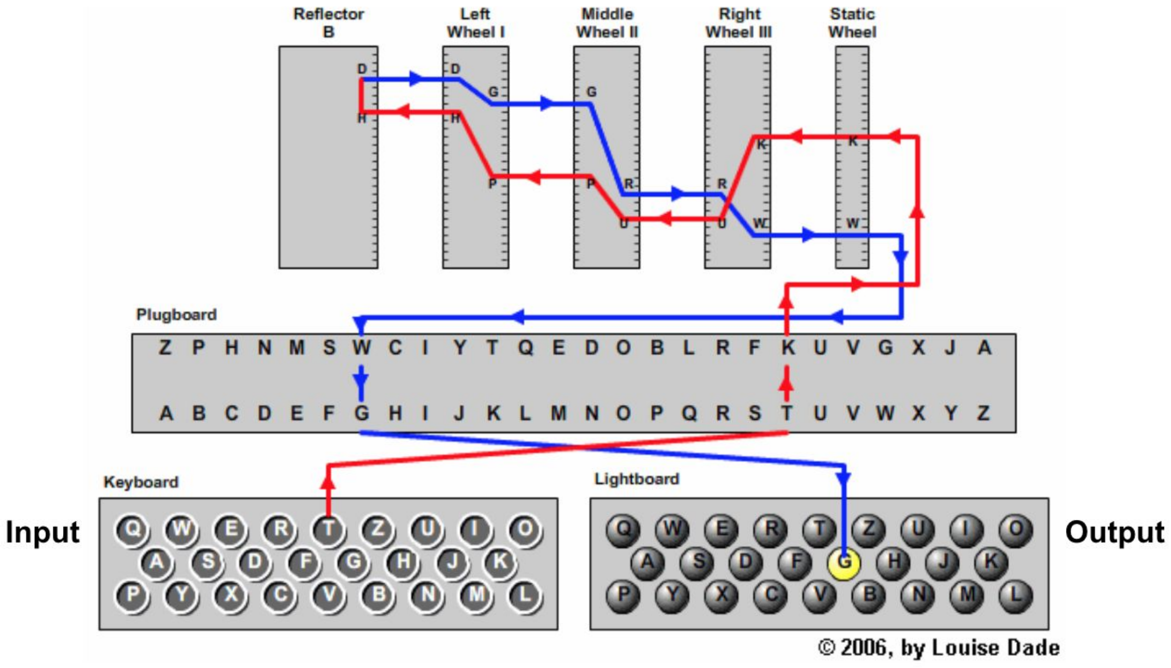


Figure 2: The internal wirings of the Enigma[2].

To decrypt: An operator sets the key of the Enigma to the same as the one that was used to encrypt the message, and then proceed to type in the ciphertext into the keyboard. Because of the nature of the Enigma, if “A” encrypts to “G” at a certain rotor position with a certain key, “G” would also encrypt to “A” at the same rotor position with the same key. An operator would simply have to set the proper key settings and type in the ciphertext to obtain the plaintext.

Types of Enigma Machines

There were two types of Enigma Machines that we wanted to attempt to break:

(1) The Commercial Enigma Machine (Enigma C)

The Commercial Enigma is an Enigma that only uses 3 rotors that can be ordered arbitrarily. This is the simplest version of the Enigma and as such was our starting point.

A harder variation of the Commercial Enigma has 5 rotors, and operators choose 3 out of the 5 to encrypt with.

(2) The Military Enigma Machine

The Military Enigma Machine is an Enigma that uses 3 out of 5 rotors and their respective starting positions. It also includes the *plugboard*, which uses 10 cables or *plugboard settings*.

Key Space of the Enigma

The characteristics of the Enigma Machine could have made it impossible to crack within the space of a day.

Theoretical Key Space

Given how the Enigma was built, there were 5 variables that contributed to the Enigma's theoretical keyspace: 1) ordered rotors, 2) initial rotor positions, 3) rotor notches, 4) reflector, and 5) plugboard combinations. In our following calculations, we assume a 3-rotor Enigma machine.

1. The total possible number of rotors that could have been constructed is $26!$. We get this number because each rotor is a permutation of the letters in the alphabet. If each rotor has positions 1-26, there are 26 letters that could have gone in position 1, and afterwards 25 letters that could have gone in position 2, and so on; this give us $26 \cdot 25 \cdot 24 \cdot ... \cdot 1 = 26!$ possible rotors that could have been constructed. If we're only using 3 rotors, that means there are $26!$ possible rotors that could have been placed in the first rotor position, $26! - 1$ possible rotors that could have been placed in the second rotor position, and finally $26! - 2$ possible rotors that could have been placed at the last rotor position. Overall this gives us $26! \cdot (26! - 1) \cdot (26! - 2)$ possibilities in the rotor construction and ordering alone.

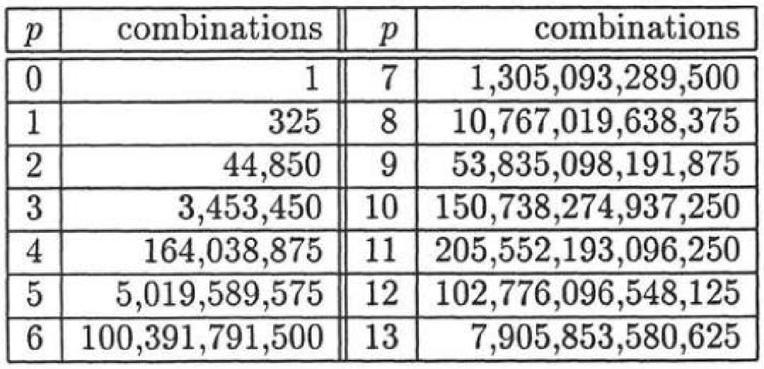
2. Given 3 rotors in a particular order, the initial positions of each of the rotors also affected how a message was encrypted. Since each rotor has 26 possible positions it could start at, this gives us 26^3 possible initial positions.

3. Each of the rotors rotated at different times. The rightmost rotor, for example, would rotate at each keypress. Depending on where a notch in an outer ring of the rightmost

rotor was placed, the middle rotor would rotate. For example, if the notch was placed at position 10 of the rightmost rotor, the middle rotor would rotate whenever the rightmost rotor reached position 10. In this way, a notch in the outer ring of the rightmost rotor would control the rotation of the middle rotor, and a notch in the outer ring of the middle rotor would control the rotation of the leftmost rotor. Because each rotor had 26 possible positions for the placement of the notch, this gives us 26^2 possible positions for the notches.

4. At the end of the rotors, there was reflector plate with a contact point for each letter. Internally, the reflector plate was made up of 13 wires, with one end of each wire connected to one contact point, and the other end of each wire connected to another contact point. For the first wire, we would pick one end to attach to a contact point, and we would be left with 25 other contact points to choose from to attach the other end into. After the first wire, we could attach one end of the second wire into a contact point, and we would be left with 23 other contact points to attach the other end into. Continuing with the other wires, we have a total of $25 \cdot 23 \cdot 21 \cdot ... \cdot 1 = 25!!$ ways to attach the wires to all the contact points.

5. For the plugboards, we have a similar situation as with the reflector plate. There are 26 holes in the board, each hole representing a letter in the alphabet. Then, we can have up to 13 wires, with one end of each wire plugging into a hole and with the other end of each wire plugging into another hole. Depending on how many wires we use, call this number p, we would have different amounts of possible combinations. First, we have $\binom{26}{2p}$ letters that we need to choose to be plugging into. Then, for the first cable, we plug one end into a random hole, and we're left with $(2p-1)$ holes to plug the other end into. For the second cable, we plug one end into a hole, and we're left with $(2p-3)$ holes to plug the other end into. So on with all the other cables and this gives us $\binom{26}{2p} \cdot (2p-1)!!$ possible combinations for any given p from 0 to 13. The total number of possible combinations for each p is shown below in Figure 3. If p were unknown, the total possible combinations would be the sum of the possible combinations for each p.



p	combinations	p	combinations
0	1	7	1,305,093,289,500
1	325	8	10,767,019,638,375
2	44,850	9	53,835,098,191,875
3	3,453,450	10	150,738,274,937,250
4	164,038,875	11	205,552,193,096,250
5	5,019,589,575	12	102,776,096,548,125
6	100,391,791,500	13	7,905,853,580,625

Figure 3: The number of combinations given *p* plugboard settings[2].

Combining all of the variables, we get:

$$26! \cdot (26! - 1) \cdot (26! - 2) \cdot 26^3 \cdot 26^2 \cdot 25!!\binom{26}{2p} \cdot (2p-1)!! = 3 \cdot 10^{114}$$

for the theoretical size of the key space[2].

Key Space in Practice

Although the Enigma had a theoretical key space of $3 \cdot 10^{114}$, it didn't have nearly as large a key space in practice, because of the following reasons:

1. The Germans generally only used the same three rotors for all of their Enigma machines. (At the peak of the war, they created up to eight possible rotors, but we'll just be assuming three to get the lower bound of the calculations.) Once the internal wirings of these rotors were discovered, it was only a matter of the ordering of the rotors that contributed to the key space, $3! = 6$.
2. There were still the same amount of initial positions for each rotors, so this value did not change.
3. Usually the Germans used the same notches for both of the outer rings of the leftmost and middle rotors, so this possibility was simply 1 after the Allies were able to figure the notches out.
4. There was usually only one unique reflector plate used in practice, leading to the possible combinations here again being 1 after the Allies figure out the wirings for the plate.
5. For the plugboards, 10 cables were most often used, meaning there were $1.5 \cdot 10^{14}$ possible plugboard combinations.

Putting it all together, in practice the keyspace was usually:

$$6 \cdot 26^3 \cdot 1.5 \cdot 10^{14} = 1.58 \cdot 10^{19}$$

Previous Work

The Bombe

The Bombe is a machine designed by the British to help crack the Enigma. Although it wasn't designed to break the machines, it crunched the permutations with respect to possible *cribs*, or possible plaintext messages. The Bombe operator selected the various Bombe settings based on the crib text and the ciphertext. The machine then ran through every possible Enigma setting, the goal being to convert all the ciphertext into plaintext with no errors. Given this history, our goal was to construct our own version of the Bombe.

Weaknesses in German Encoding

Despite the impressive theoretical keyspace the Enigma boasts, there were a number of exploitable weaknesses in practice[3].

Machine Errors

One of the weaknesses of the German Enigma Machine was that no letters ever encrypted to themselves. This meant that an “A” would never encrypt to another “A”. This fact becomes useful in reducing decryption time via Crib Matching.

Human Weaknesses

Although the Enigma Machine did have inherent weaknesses in its design, human weaknesses were the main reason the Allies could find patterns that allowed them to decrypt the messages. We detail 4 of the major ones below:

1. “Random” message keys were not truly “random”. When encrypting messages, operators were supposed to choose their own “random message key” (just the initial rotor settings), and then use the daily key configuration to encrypt their message key at the beginning of the transmitted message. This prevented identical messages from being encrypted the same way on the same day.

   The receiving operator would first decrypt the message key, and then use that key to decrypt the rest of the message. Because operators were human, they often didn’t bother to change the message keys between messages, or would only use a small subset of message keys. The cryptographers at Bletchley Park used this to create what was essentially a “rainbow table” of message keys for each operator and tested decrypting the message with those keys before anything else.

2. Messages used common phrases. The Allies discovered that the Germans frequently encrypted a certain phrase that demonstrated their appreciation for Hitler at the end of every message. This guaranteed at least one crib-ciphertext match.

3. The Germans sent out a daily weather report. Every day at 6 A.M. the Germans would send a daily weather report in a specific templated format. The cryptographers would ask the meteorologists what the weather was for that day and could then construct a complete crib for the ciphertext.

4. The Germans had complete faith in the Enigma and its invincibility, and therefore didn’t try to tighten up security or change their ways (with the exception of their Navy).

Given its theoretical keyspace, these errors became vital to the eventual cracking of the Enigma.

Implementation

We decided to go forth with the following assumptions, ie. a simplified military grade Enigma:

- a 3-rotor Enigma
- random initial rotor positions/settings
- 10 plugboard cables
- knowledge of “good” cribs (ie. we know some parts of the messages for certain)

Note that extending to a 5-rotor Enigma would simply multiply the time taken by $\binom{5}{3} = 10$, in order to check all possible different choices of 3 rotors.

Between the internal wiring of the rotors, notches, and reflector-type alone, there are 10^{95} different key combinations possible. In the interest of time, we decided to assume the basic set-up of the Enigma was known. The Allies were able to make a similar assumption, as they were able to get their hands on some of the German Enigma Machines[3].

This assumption reduces the theoretical keyspace of ~10^{114} significantly. Only considering all the different possible rotor orders, initial positions, and plugboard settings, the keyspace is on the order of 10^{19}, which is similar to a 64-bit key. Keys of this size are not considered secure by today’s standards, given the availability of computational power. But, supposing we only have access to our home machines, we require some way of narrowing down the keyspace rather than attempting simple brute force. Otherwise, under the (very generous) assumption that our computers can check 10^9 keys / second, pure brute force on this keyspace will have a worst-case runtime of 10^{10} seconds = 300+ years.

We programmed our code in C, with the exception of the loop-finding algorithm, which was programmed in Python. We worked from a template Enigma implementation in C as our base and modified the code from there[4]. We modified the code to implement a couple different exploits, taking advantage of both weaknesses that resulted from human error and those from mechanical weaknesses, and used the following steps to recover the key.

Crib Matching

As letters will never encode to themselves, it was not necessary to check the crib against every single ciphertext “chunk”. In Figure 4, we see that SAM can never decrypt to HAM, as the A’s and M’s align. Therefore, it’s impossible to find a key that does so and

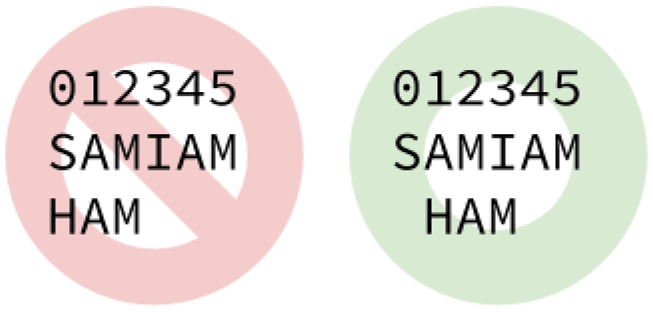


Figure 4: An example of using alignment to figure out crib-to-cipher matches.

there's no point in trying. On the other hand, it's possible for AMI to decrypt to HAM, as there are no such alignments, and we must then look for the keys that do so.

One can see that, in this small example, we've cut our runtime in half. Instead of checking HAM against all 4 chunks (SAM, AMI, MIA, IAM), we only need to find keys that properly decrypt 2 chunks (AMI, MIA).

Finding Key from Crib Match

While this is not a means of reducing the keyspace, it's worth mentioning how we recover the original Enigma key from the partial crib match. Because the rotor positions change for each typed letter, we can consider each letter to be encrypted with a different key. Additionally, because these rotors change in a predictable manner (the first rotor always rotates one position, the second rotates when the first reaches a notch, etc.) and the plugs will not change mid-message, one can simply look at the starting index of the crib match (call this *i*), and roll the Enigma back *i* steps to get the original key (consider *i* to be 0-indexed).

We could then use this key to decipher the entire ciphertext, and we could find the correct key by, for example, looking at the keys which decrypted the message to sensible English.

Loops

A useful tool in finding key and plugboard settings is "loops". Given a ciphertext and crib, we can line up the ciphertext and crib by letter position, creating pairs of ciphertext and crib letters. Then, we can create a graph, where the nodes are the letters and where undirected edges exist between each ciphertext and crib letter pair, with edge weights representing the letter position in the text. Any cycles that appear in the created graph are loops.

Once we find a ciphertext to crib match, we can construct loops and use those loops to find possible key and plugboard settings.

Finding Loops Given Match

We first align the ciphertext and crib, numbering each character by its index in the given text. For example, if we had a given ciphertext "IUGHLUVFAOBNEWNAGZW" and a given crib "MARKWORTHXATTACKEDX", we would line up the characters as follows:

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
CT	I	U	G	H	L	U	V	F	A	O	B	N	E	W	N	A	G	Z	W
Crib	M	A	R	K	W	O	R	T	H	X	A	T	T	A	C	K	E	D	X

Given this alignment, we can create a graph by first taking each of the letters and treating them as nodes in the graph. Then, we can add edges between each pair of ciphertext and crib characters. For example, “U” and “A” would each be a node in the graph, and there would be an edge of weight 1 between them. “H” would also be a node in the graph and there would be an edge of weight 8 between node “H” and node “A”. The graph version of this example can be seen below in Figure 5. This graph is also referred to as a *menu*.

To find the loops in this graph, we run DFS on each of the nodes, keeping track of paths as we traverse through the graph, and we return any cycles that we find in the graph.

Exploiting Loops

Once we have a loop, we can proceed to use it to check against each of the different initial rotor settings to find possible keys and plugboard settings.

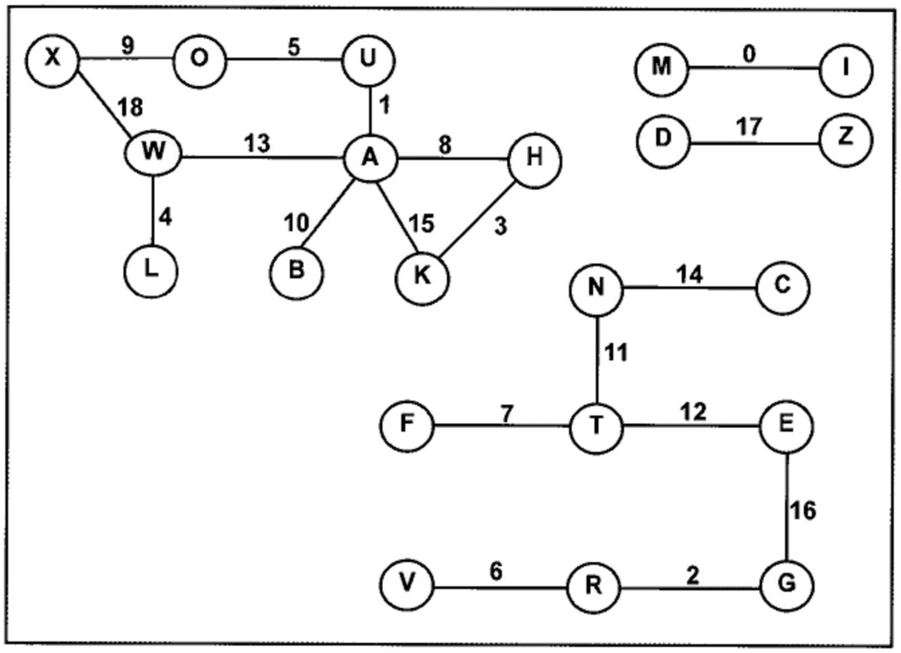


Figure 5: The menu for the crib-cipher matching ““MARKWORTHXATTACKEDX” to “IUGHLUVFAOBNEWNAGZW”[3].

In the diagram above, there are two loops: XOUAW, and AHK. We will use the loop AHK as an example. Since we don’t know whether or not A, H, or K was used in the plugboard, let us denote a mapping:

A ⇒ α
H ⇒ β
K ⇒ γ

Where α, β, and γ are our respective guesses for the plugboard substitutions for A, H, and K.

To test a certain initial rotor setting, we set our rotors to that setting and then make a guess for α. Using the diagram, we know that on the 8th step, the Enigma outputs an H, so if we encrypt a sequence of 8 α's, the character we get at the end of that sequence is our guess for β. We repeat this in a similar fashion to get our guess for γ. Lastly, to close the loop, we repeat this again with γ to obtain another guess for α, which we will denote as α'. We check if α' is the same as our initial guess for α, and if so, we know this rotor setting satisfies the loop.

In order for a rotor setting to be valid, it must satisfy all the loops, and for each unique letter in the loops, we find a possible plugboard setting for that letter. Therefore, the more unique letters there are in the loops for a crib-ciphertext match, the more plugboard settings we can deduce.

The Bombe used a similar method to solve the plugboards. Operators of the Bombe would plug in the loops, and run the Bombe until it "stopped", or when the key settings satisfied the loops.

Exploiting Loop Implementation

We implemented a version of the Bombe that takes in 3 loops and finds the key settings that satisfies those loops, as well as the possible plugboard substitutions for each loop. Since there are cases in which multiple plugboard substitutions could account for a loop, we had to store them all and check which combinations of them provided valid substitutions. We only accepted keys with valid substitutions for all 3 loops, which narrowed down our keyspace significantly (Results). The Allies would also only attempt decryption of messages that had at least 3 loops[5].

Finding Remaining Plugs after Loops

In order to find the rest of the key, we use a brute force search over the keyspace. At this point, our keyspace has been sufficiently reduced such that this method is more than efficient.

Results

Given that the commercial Enigma Machine only had $26^3 \cdot 3 \approx 100,000$ possible different keys, this space was fairly easy to crack just using our Crib Matching technique (taking no longer than a minute or two, usually).

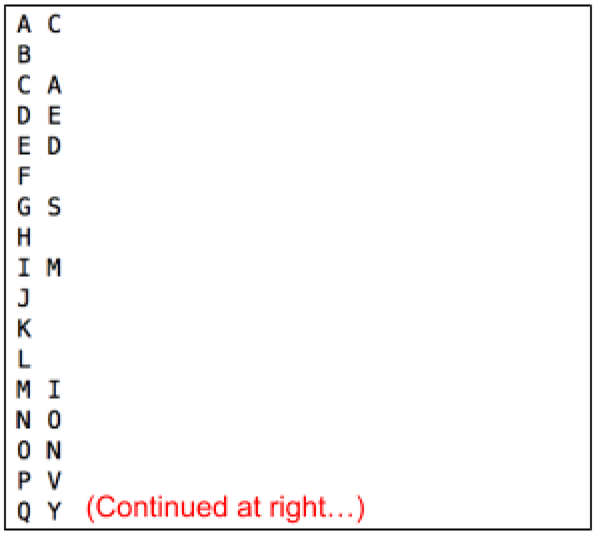
Naturally the addition of the plugboard makes things a little more difficult, and our success depended entirely upon how many loops / plugboard pairs we were able to retrieve from the message (Figure 3). Our loop generation algorithm generates all possible loops in the menu, so when we were deciding which loops to use for our Bombe implementation, we chose the set of loops that would have the most amount of unique letters.

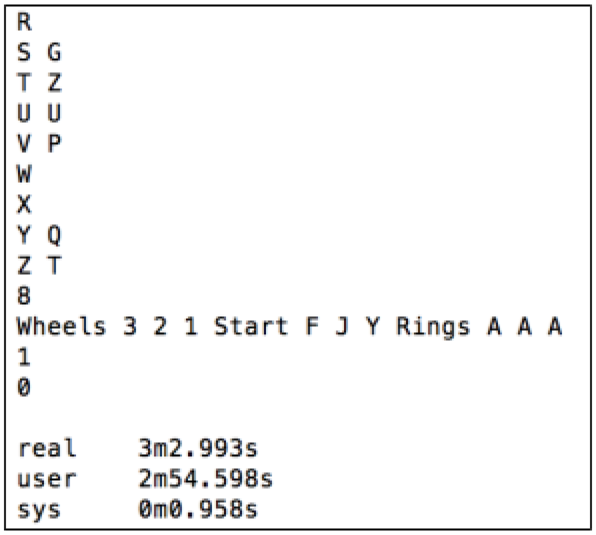
We ran a test with 10 plugboards and our implementation found the key setting and 8 out of the 10 plugboard settings in around 3 minutes. We then used this to figure out the last 2 settings using the brute force method, which took a trivial amount of time, for a total time of approximately 3 minutes. Given 8 out of the 10 plugboard settings, we only had to search for the rest of the plugboard pairings in a space of 10^3, since our 8 found settings eliminated a subset of the 26 letters.

The test was as follows:
Ciphertext: ERWEEMGZDQJSTPCPYUAUEFDZOANLN
Crib: SASUNARUTOANIMEDATTEBAYOANIME
Key (eg. what we wanted to find):

- Rotor order: 3 2 1
- Initial rotor positions: F J Y
- Rings (we didn't take these into account, and thus they're set to default): A A A
- Plugboard settings: AC DE GS IM NO PV QY TZ BH KW

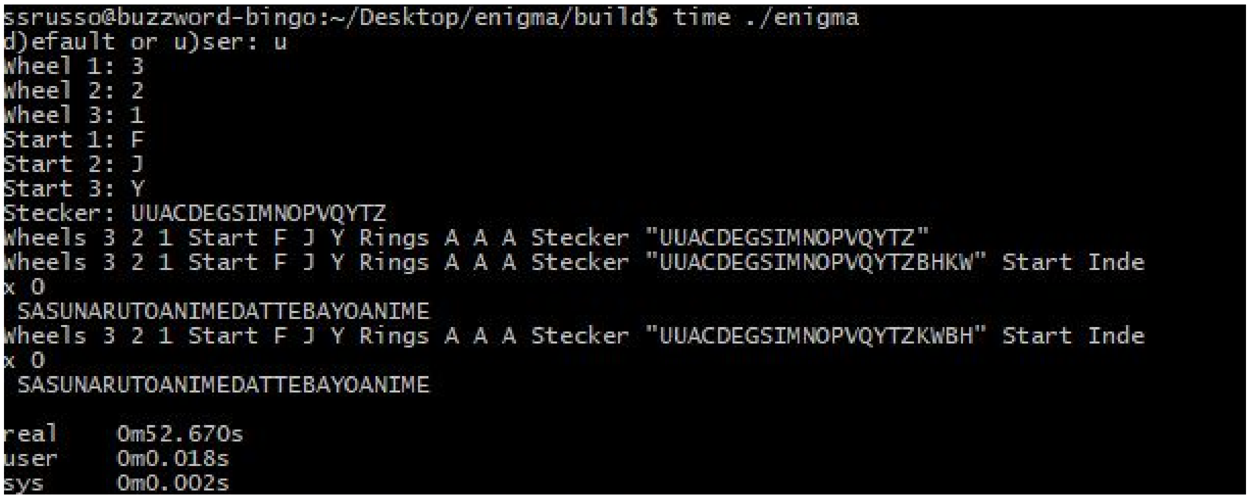
Figures 6 and 7 demonstrate our results and the amount of time they took to run. Though this example in particular didn't make use of Crib Matching, one can see that it would have simply multiplied the runtime by a factor of however many valid crib-ciphertext chunk pairings we had.


```
A C
B
C A
D E
E D
F
G S
H
I M
J
K
L
M I
N O
O N
P V
Q Y
```
(Continued at right…)


```
R
S G
T Z
U U
V P
W
X
Y Q
Z T
8
Wheels 3 2 1 Start F J Y Rings A A A
1
0

real    3m2.993s
user    2m54.598s
sys     0m0.958s
```

Figure 6: A screenshot of our code finding and using loops to narrow down the rotor and plugboard settings. We find 8 out of the 10 plugboard settings, determine that 'U' is unpaired, and see that there's only 1 valid rotor setting.



```
ssrusso@buzzword-bingo:~/Desktop/enigma/build$ time ./enigma
d)efault or u)ser: u
Wheel 1: 3
Wheel 2: 2
Wheel 3: 1
Start 1: F
Start 2: J
Start 3: Y
Stecker: UUACDEGSIMNOPVQYTZ
Wheels 3 2 1 Start F J Y Rings A A A Stecker "UUACDEGSIMNOPVQYTZ"
Wheels 3 2 1 Start F J Y Rings A A A Stecker "UUACDEGSIMNOPVQYTZBHKW" Start Inde
x 0
 SASUNARUTOANIMEDATTEBAYOANIME
Wheels 3 2 1 Start F J Y Rings A A A Stecker "UUACDEGSIMNOPVQYTZKWBH" Start Inde
x 0
 SASUNARUTOANIMEDATTEBAYOANIME

real    0m52.670s
user    0m0.018s
sys     0m0.002s
```

Figure 7: A screenshot of our code using brute force to find the rest of the plugboard settings. It takes significantly more time to enter the known key settings ("real" time) than it does to solve the rest of the key ("user").

Generally, we were able to find at least 7 out of the 10 plugboard settings, in addition to knowledge of letters that had no pair (in the above example, this is 'U'), allowing us to achieve excellent runtimes. Note that the runtime of the loop-finding/plugboard-setting-finding algorithm wasn't dependent on the number of loops — only the brute force algorithm was. This means that even if 7 out of 10 has a keyspace 100x larger than if we'd found 8 out of 10, our runtime would increase by only 1 second ($0.018\ seconds \cdot 100 \approx 1\ second$).

Conclusion & Future Work

Our Bombe managed to reduce the original keyspace of 10^{19} to ~10^4 (for 3 unknown plugboard settings) and ~10^3 (for 2 unknown plugboard settings), which made it manageable to brute force within a relatively short amount of time.

Rainbow Table

We were also fairly generous with our assumption that the initial rotor positions were truly random. Part of what made the Allies able to crack the Enigma was due to the fact humans are lazy, and Enigma operators wouldn't change the initial settings by much between messages. Had we taken this into account and created a rainbow table of common rotor positions, we would have shrunk the keyspace of rotor positions even prior to loop-exploits (perhaps to 50 well-known settings).

Expanding on the Menu

So far, our machine only uses loops to find possible plugboard combinations for the specific characters in that loop, but we can expand on this to find plugboard combinations for all characters connected to that loop as well.

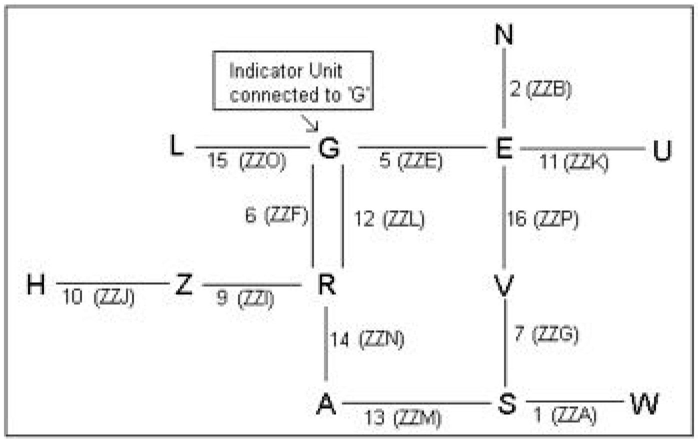


Figure 8: Another menu

Let's say that we're given the menu shown above and we've found a key and plugboard setting for the loop GEVSAR, which means we also now have the corresponding plugboard characters for GEVSAR. Let's say that we find that G's corresponding plugboard character is "O". We can now encrypt "O" 15 times to get the corresponding plugboard character for "L", and likewise do the same to get corresponding plugboard pairs for all the characters that are connected to the loop. In doing so, we've now expanded the number of known plugboard matchings from the original 6 characters in the loop to 12 characters.

Parallelization

We also didn't parallelize our machine, though it could have sped up our computations. In future iterations, it certainly would be advisable to do so, to perhaps handle cases in which we don't find as many plugboard pairs. However, as is, we feel our implementation is plenty fast, beating the 20 minute runtime[3] of the original Bombe by almost a factor of 10. Of course, we have significantly more computational power than the original Bombe in our laptops.

This highlights something interesting about WWII, in that theoretically, the Enigma should have been impossible to crack given the technology of the day. But, there was so much human error in predictable message contents (allowing for cribs and loop exploitations), initial rotor settings, and so on, that the keyspace was reduced to the point that the Enigma could be broken. This, naturally, brings us to the takeaway message of our project: no matter how good your crypto may be, never assume that people are going to implement it correctly.

References

[1] Dufresne, Steven. “The Enigma Enigma: How The Enigma Machine Worked.” Hackaday, Hackaday, 22 Aug. 2017, hackaday.com/2017/08/22/the-enigma-enigma-how-the-enigma-machine-worked/.

[2] Miller, A. Ray. “The Cryptographic Mathematics Of Enigma.” *Cryptologia*, vol. 19, no. 1, 1995, pp. 65–80., doi:10.1080/0161-119591883773, https://www.tandfonline.com/doi/pdf/10.1080/0161-119591883773.

[3] NSA. “Solving the Enigma: History of the Cryptanalytic Bombe.” Polybius at The Clickto Network, Fox News, 2002, web.archive.org/web/20100317234354/http://ed-thelen.org/comp-hist/NSA-Enigma.html.

[4] Schmidl, Harald. “On Enigma and a Method for Its Decryption.” Enigma and a Way to Its Decryption, 1998, www.cs.miami.edu/home/harald/enigma/index.html.

[5] Carter, Frank. “The Turing Bombe.” The Turing Bombe in Bletchley Park, Rutherford Journal, www.rutherfordjournal.org/article030108.html.