CRYPTANALYSIS OF TYPEX

Kelly Chang
San Jose State University

CRYPTANALYSIS OF TYPEX

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Kelly Chang

May 2012

The Designated Project Committee Approves the Project Titled

CRYPTANALYSIS OF TYPEX

by

Kelly Chang

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2012

ABSTRACT

Cryptanalysis of Typex

by Kelly Chang

Rotor cipher machines played a large role in World War II: Germany used Enigma; America created Sigaba; Britain developed Typex. The breaking of Enigma by Polish and (later) British cryptanalysts had a huge impact on the war. Despite being based on the commercial version of the Enigma, there is no documented successful attack on Typex during its time in service.

This project covers the Typex machine. We consider the development of Typex, we discuss how Typex works, and we present and analyze two distinct cryptanalytic attacks on the cipher. The first attack assumes the rotor wirings are known and uses Turing's crib attack–originally developed for Enigma–to recover the settings of the stepping rotors. It then performs a hill-climb to recover the static rotor settings. The second attack assumes that the rotor wirings are unknown. This attack uses a nested hill-climb to find the wirings of the stepping rotors.

ACKNOWLEDGMENTS

I would like to thank my parents for their support and persistent haranguing.

TABLE OF CONTENTS

APPENDIX **Page**

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

After World War I, it was clear to the British government that they needed a stronger, more efficient, mechanized cipher system. In 1926, the government established the Inter-Departmental Cipher Committee to explore possible cipher machines to replace their current book cipher systems. Almost ten years later, in 1935, the Committee decided upon “Enigma type cipher machines improved through the use of ‘Type X’ attachments” or Typex [11]. These improvements included the use of patents designed for the commercial Enigma that went unused by the Germans. The Typex machine, developed by Wing Commander O.G.W. Lywood [6], was such a close relative of the Enigma machine used by the Germans that not only did the British use Typex machines in place of Enigma ones when trying to decipher Enigma messages, but when German soldiers recovered a Typex machine sans rotors, they successfully converted it into an Enigma machine [11]. Ironically, the similarity between Typex and Enigma discouraged German cryptanalysts from attempting to cryptanalyze Typex enciphered messages because they believed Enigma to be unbreakable [14].

The goal of this project is to design, implement, and test two possible attacks on the Typex cipher machine. We discuss the first attack in Section 3. It combines Turing’s crib attack on the Enigma (Section 3.1), and a hill-climbing heuristic technique (Section 3.2). This attack assumes two things: 1. the attacker knows the rotor wirings but not the key used to encrypt the ciphertext, and 2. that the attacker has some knowledge of the contents of the ciphertext, that is, the attacker knows a so-called crib [18]. We apply the crib attack to recover settings of the stepping rotors and the hill-climbing technique to get settings of the static rotors. The two methods used in the first attack highlight the similarities and differences between the Typex and its German cousin, the Enigma.

The second attack, which is discussed in Section 4, assumes the rotor wirings are unknown, but the key is known. For this attack, we utilize a nested hill-climb technique to solve the rotor wirings. The implementation of this attack is not specific to the Typex cipher machine and therefore may be extended to other rotor cipher machines.

In this project report, we begin in Section 2 by giving a reasonably complete description of the Typex cipher, and we examine the size of the Typex keyspace. In Section 3, we discuss our first attack on the cipher machine and Section 4 describes our second attack on the Typex. In Section 5, we analyze the results of the second attack. Section 6 discusses Rejewski's method of recovering the rotor wirings. Lastly, Section 7 concludes the report.

CHAPTER 2

The Typex Cipher Machine

The Typex went through several different designs during its years ofservice [6]; the model pictured in Figure 1 is a Mark III, the "portable" version [11]. Its cousin, the Mark VI, another hand powered version of the Typex, weighed in at 30 pounds, slightly more than the Enigma's 26.5 pounds [6]. The most widely produced and distributed versions during World War II were the Mark II and Mark IV, numbering at around 8,000 and 3,000, respectively [6]. A typical Typex machine came with eight rotors, one reflector, a keyboard, and a printer [21]. The Mark III and IV model included a handcrank to power itself [11]; others, like the Mark II, required a 230-volt AC power supply, making them stationary cipher machines [15]. Of those eight rotors, the operator would choose five of them to insert into the machine and assign their initial settings, letters A-Z. The first three rotors from the left of the machine stepped as the operator typed, and two rightmost rotors remained static, also known as stators. It was also possible to reverse the Typex rotors, effectively doubling the amount of rotors an operator could choose from [11].

The stators served a similar purpose to the Enigma's stecker. They permuted the signal before it reached the stepping rotors. However, unlike the stecker, the stators were not reciprocal [11]. The reciprocal property made it easier for British cryptanalysts to recover the stecker settings [18]. By lacking this property, the stators were slightly more secure than the stecker as shown in Section 3.2.

When a Typex operator typed a letter, it traveled in a similar manner to the Enigma: from the keyboard the signal went through the two stators, followed by the

Figure 1: Typex Cipher Machine [11]

three stepping rotors, then across the reflector, and back through the inverse of the stepping rotors and stators where it would then be printed out [21]. The path of the signal can be seen in Figure 2.

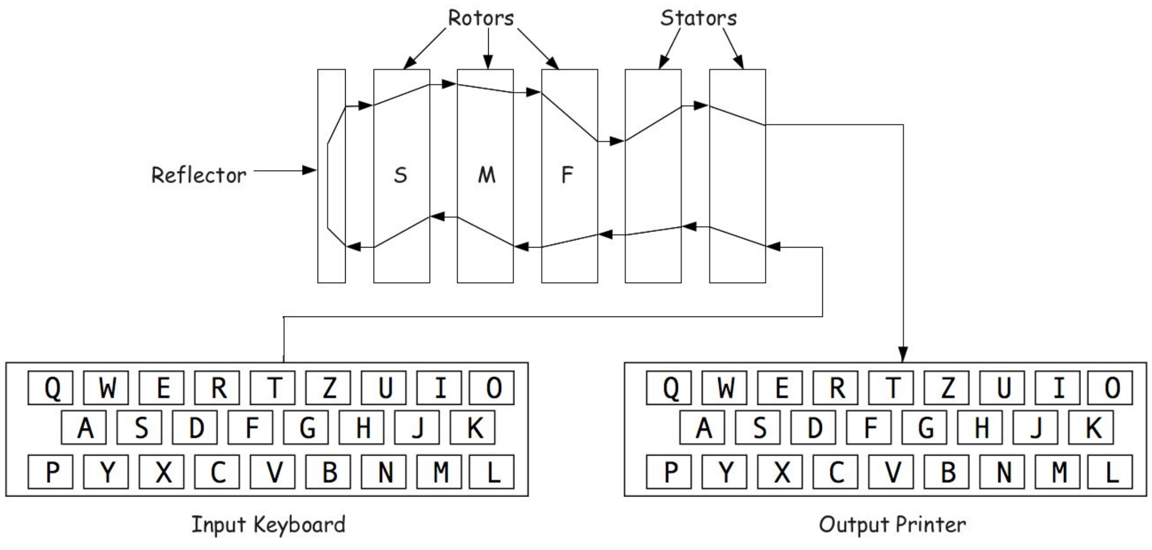


Figure 2: Typex Encryption

We use the following altered notation from [18] to discuss the various permutations in the Typex:

$$S_r = \text{right stator}$$

$$S_l = \text{left stator}$$

$$R_r = \text{right rotor}$$

$$R_m = \text{middle rotor}$$

$$R_l = \text{left rotor}$$

$$T = \text{reflector.}$$

For example, we write the encryption of a plaintext character `x` to ciphertext `y` as:

$$y = S_r^{-1} S_l^{-1} R_r^{-1} R_m^{-1} R_l^{-1} T R_l R_m R_r S_l S_r(x). \qquad (1)$$

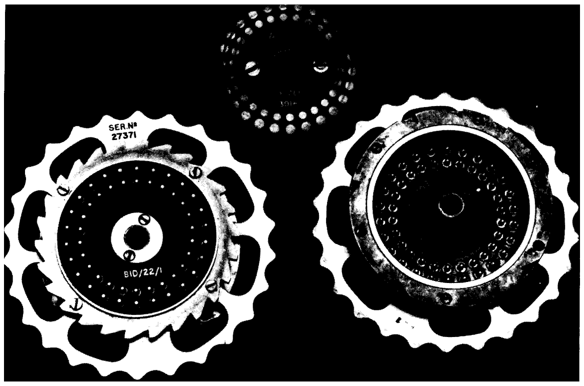


Figure 3: Typex Rotors [11]

The frequency order of the Typex's stepping rotors matches that of the Enigma; however unlike the Enigma, the middle and left rotors stepped more than once per revolution of its neighboring rotor. One of the major improvements made to the Typex over the Enigma was this irregular stepping of the rotors. The irony of the addition of the irregular stepping was that the British merely implemented the existing unused Enigma patents [11]. The Typex stepped anywhere between five and nine times per revolution [2]. The multi-notched rims attached to the rotors determined the exact number of steps per revolution. Unlike the Enigma, where the moveable rims were independent of each other [18], the multi-notched rims of the Typex were the same for both the middle and left rotor, meaning that if the middle rotor stepped when the right rotor was at `ADGILNQUY`, then the left rotor stepped when the middle rotor was also at those positions. By adding these multi-notched rims to the rotors, cryptanalyzing the Typex became a vastly more difficult task. The multiple steps per revolution meant that even if the Germans attempted to decipher Typex encrypted messages they would not only need more ciphertext than the British did to crack Enigma encrypted messages, but it would also take longer [11].

The multi-notched rims made the stepping more random than the Enigma rotors, and therefore increased the difficulty of predicting when the Typex rotors stepped. For example, if the Germans knew the Typex rotors stepped nine times per revolution but not the exact positions, there would still be 26 choose 9, about $2^{21.6}$, different possible combinations. The British had a mere 26^2 combinations, about $2^{9.4}$, given the Enigma's single step per revolution [18].

2.1 Typex Keyspace

The full theoretical size of the Typex keyspace includes three components:

1. The choice of rotors, both stepping and static, and their initial position
2. The choice of multi-notched rim.
3. The choice of reflector.

The rotors create a permutation of the 26 letters of the alphabet and each rotor is initially starts in one of 26 possible positions corresponding to A through Z. Because the two stators remain in a fixed position, they are cryptographically equivalent to one rotor. That is, if the input to the first stator is `A` and the output of the second stator is `K` then for every `A` entered given the current settings the output of the second stator will always be `K`. Therefore, if the wirings of the rotors are unknown there are a total of $(26!)^4$, approximately $2^{353.5}$, possible ways to select the rotors and set their initial positions. The multi-notched rim can have between five and nine notches. This provides at most about $2^{21.6}$ different combinations, as previously mentioned. The Typex reflector worked the same way as the Enigma one, which had approximately $2^{42.8}$ different possible combinations [18]. In total, the theoretical size of the Typex keyspace amounted to approximately $2^{353.5} \cdot 2^{21.6} \cdot 2^{42.8}$, or $2^{417.9}$, roughly a 418 bit key. However, in practice this was much less, largely due to limitations on the number of available rotors and multi-notched rims, as well as having only one reflector [21].

The operator of the Typex machine set the key by selecting the rotors, their order, orientation, and initial position. Assuming that the operator only had the standard eight rotors to choose from the selection of the rotors and their order is $\frac{8!}{3!}$, about $2^{12.7}$, possible permutations. The orientation is a binary choice, forwards or backwards, making that part 2^5. Initial position of the rotors refers to the letter that it started at, giving the operator 26 options per rotor or 26^5 in total. The total keyspace based on these values ends up being $2^{12.7} \cdot 2^5 \cdot 26^5$, or approximately $2^{41.2}$.

2.2 Converting an Enigma Simulator to Typex

Because of the similarities between the Typex and Enigma, we start with an accurate, working copy of Enigma code and convert it to Typex; this involves several different changes. We remove the stecker and replace it with the two stators. However, this requires a little more than a simple find and replace. Because the stators are rotors, we also include them when calculating the inverse permutations of the rotors. Unlike the stecker, which is simply a plugboard, the stators permute all the letters of the alphabet. This permutation changes based on the orientation of the stator and its positioning, which means that the inverse permutation must also adapt accordingly. This leads to expanding the rotor initialization section of the code to not only include the three stepping rotors but also the stators as well. In addition to that, the formula to permute the incoming plaintext to ciphertext, and vice versa, needs to be altered so that the character goes through the stators first and comes out their inverse last.

Other modifications include the extra notches on the rotors and reversing the rotors. Based on our research, we concluded that the notches for each rotor were the same [1, 4]. This means that instead of creating unique notch settings for every rotor, we only need one and each rotor can check the notch settings to determine whether or not to step based on its current position.

By far, the hardest part of the Typex simulator involves reversing the rotors. With physical rotors, the operator simply flipped it over and inserted the rotor into the machine. For the simulator however, the rotor needs to be completely "rewired." With the help of paper rotors, we successfully designed a way to reverse the rotors in the simulator. Keeping in mind that a reversed rotor must contain two poles that retain their previous "wirings," reversing a rotor becomes a matter of swapping these "wirings" with the correct position. We achieve this by using subtraction and the modulo operator.

Finally, the last change we implement adds the space character as a part of the acceptable input. For the Enigma code, the only valid plain or ciphertext are the letters `A` to `Z` all uppercase. With Typex, the operator can use the space bar, but it is connected to the `X` key on the keyboard and enciphers as such [11]. Therefore,

Table 1: Pseudo Code to Reverse a Rotor

```
create a new rotor[26][26]
for $i = 0$ to 26
    for $j = 0$ to 26
        new rotor[$i$][$j$] = old rotor[$i$][$(26 - j)$ %26]
    next $j$
next $i$
```

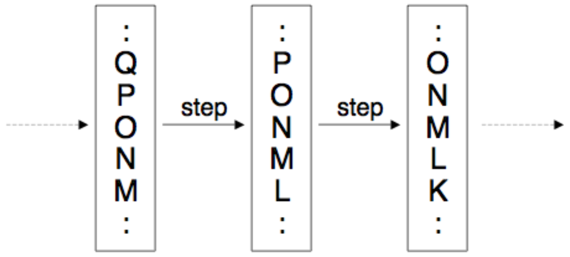


Figure 4: Typex Rotor in Forward Orientation [17]

our simulator checks for the space character and converts it to `X` prior to enciphering, similar to the actual machine. This means that when deciphering any text the end result will have the letter `X` in place of spaces as the simulator has no way to differentiate between spaces enciphered as `X` and actual `X` characters. See Appendix C for the full Typex simulator code.

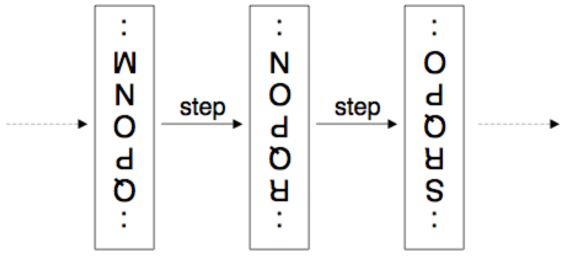


Figure 5: Typex Rotor in Reverse Orientation [17]

Table 2: Pseudo Code of Encryption/Decryption and Stepping Part of Typex Simulator

```
read file one character at a time
while not EOF
    if character is a space
        convert to X
    end if
    if character is not valid
        exit
    end if
    if middle rotor at a notch
        step middle and left rotor
        step backwards if rotor is reversed
    end if
    if right rotor at a notch
        step middle rotor
        step backwards if rotor is reversed
    end if
    always step right rotor
    step backwards if rotor is reversed
    output = RS_inv[init_RS][LS_inv[init_LS][R_inv[cur_R][M_inv[cur_M]
            [L_inv[cur_L][reflector[L[cur_L][M[cur_M][R[cur_R][LS[init_LS][RS
            [init_RS][inChar]]]]]]]]]]
    print output
read in next character
```

CHAPTER 3

Cryptanalyzing the Key

3.1 Breaking the Rotors

With the simulator running correctly, we take the next step: cryptanalyzing the Typex. Because of the similarities between the Typex and the Enigma, we decide to start with a technique that can be used to break the Enigma. We choose Alan Turing's crib attack because it works without knowing the stecker settings of the Enigma; this allows us to ignore the Typex stators as they operate similar to the stecker only without the reciprocal property [11].

Turing's attack requires some known plaintext, also known as a crib. Turing compared the plaintext against the ciphertext produced by the Enigma, looking for cycles between the letters. With enough cycles Turing could recover the rotor settings of the Enigma and, with a little extra work, most of the stecker settings as well [18]. This attack works similarly on the Typex but without the added benefit of recovering the stator settings.

i	0	1	2	3	4	5	6	7	8	9	10	11
Plaintext	T	E	S	T	O	F	T	Y	P	E	X	C
Ciphertext	P	L	M	A	D	L	I	C	V	B	M	Y
i	12	13	14	15	16	17	18	19	20	21	22	23
Plaintext	I	P	H	E	R	M	A	C	H	I	N	E
Ciphertext	S	U	D	T	L	W	P	R	M	V	V	H

Table 3: Typex Crib Attack Example

The crib attack works as follows: we let $S(x)$ be the transformation of the letter x when it passes through the stators, or $S_lS_r(x)$ from our previous notation;

similarly $S^{-1}(x)$ is the transformation when x passes through the stators in the opposite direction, $S_l^{-1}S_r^{-1}(x)$. We also define P_i as the permutation at step i, or the path the signal take as it travels through the two stators, three rotors, across the reflector, and back through the rotors and the stators in the opposite, also called inverse, direction. To sum up:

$$P_i = R_r^{-1}R_m^{-1}R_l^{-1}TR_lR_mR_r. \tag{2}$$

This also means that the inverse permutation, P_i^{-1}, exists [18].

By using $S(x)$, $S^{-1}(x)$, P_i, and P_i^{-1}, we create equations of known plaintext and its ciphertext counterpart, which we then link together to form a cycle for a letter. For example, in Table 3 above, we describe the transition in the column 6 in the following way: the plaintext character `T` passes through the stators, $S(T)$, through the rotors, P_6, and back through the stators, S^{-1}, resulting in the ciphertext character `I`. In equation form, we write this as $S^{-1}P_6S(T) = I$, which we can rearrange as $P_6S(T) = S(I)$. We create similar equations for columns 12, 2, 20, 23, and 15 as shown below,

$$\begin{aligned} P_{12}S(I) &= S(S) \\ P_2S(S) &= S(M) \\ P_{20}S(H) &= S(M) \\ P_{23}S(E) &= S(H) \\ P_{15}S(E) &= S(T) \end{aligned} \tag{3}$$

which, when combined with the first equation we obtain

$$S(T) = P_{15}P_{23}^{-1}P_{20}^{-1}P_2P_{12}P_6S(T). \tag{4}$$

We test this equation against all possible rotor settings. If the equation holds true for some rotor setting, meaning that there exists an input character, A-Z, that has the same output character after running through the cycle, we keep this rotor setting as a potential putative key. If we try all characters and the equation does not hold true for any of them, then we discard the rotor setting as a potential putative key. However, with only one cycle this approach would not help us anymore than guessing at random does because there are 26 possibilities for $S(T)$ and a 1/26 chance that the equation will hold true at random [18].

We can avoid this by finding additional cycles that equal $S(T)$. For example,

$$\begin{aligned} P_0 S(T) &= S(P) \\ P_8 S(P) &= S(V) \\ P_{21} S(I) &= S(V) \\ P_6 S(T) &= S(I) \end{aligned} \tag{5}$$

gives us the equation $S(T) = P_0^{-1} P_8^{-1} P_{21} P_6 S(T)$. When combined with the first equation we reduce the chance that both equations hold true at random to $(1/26)^2$, reducing the number of potential keys by a factor of 26 [18]. Therefore, with the Typex's $2^{41.2}$ possible keys we expect to need about nine or ten pairs of cycles to recover the key.

3.2 Cracking the Stators

When the British discovered the correct rotor settings on the Enigma, they could simply deduce the position of the stecker cables by looking at the putative plaintext and seeing which letters were swapped. For example, if the putative plaintext read `LEIHLITHER` then the British could easily see that a stecker cable swapped the letters `L` and `H` [16]. However, because the stators are not their own inverse like the Enigma's stecker is, we cannot use the cycles found in the crib attack to recover the stator settings. That is, while the cycles continue to hold true if the stator settings are incorrect, the output produced by the settings still looks like ciphertext. Instead, we decide to use a hill-climb attack. In this hill-climb technique, we begin with an arbitrary solution and incrementally improve the solution through a series of minor transformations, only retaining the modification if the new version of the solution is better than the previous [10].

The two main parts of a hill-climb algorithm are the technique used to iteratively modify the solution and the method of calculating how "good" or "close" the putative solution is to the actual one. The easiest way to measure a solution's goodness is to use a numeric score. A solution with a score closer to the actual solution's score can then be considered a better solution than the previous one. As the solutions continue to improve the scores will continue to climb, thereby giving the technique its name. However, a significant flaw exists in the hill-climb technique because the outcome of the algorithm depends heavily on the initial solution. Because hill-climbing only accepts improving scores, if the current solution is the local optimum but not the global optimum then the hill-climb technique will be unable to find the actual solution [5].

In hill-climbing the stators we incrementally guess the stator settings and attempt to solve the ciphertext based on those settings. Figure 6 shows an image of the following description. Given an initial guess of two stators and their positions, we first check all possible settings of the right stator up to `Z`, then increment the setting of the left stator to its next setting. For example, if the guess is rotors 3 and 4 set to `S` and `J` respectively, we first check that setting before moving on to rotors 3 and 4 set to `S` and `K`, then set to `S` and `L` and so on until `S` and `Z`. After checking that setting, we change rotor 3 to `T` and rotor 4 to `A`. When we have exhausted all the settings up to rotor 3 at `Z` and rotor 4 at `Z` we then change the right stator to the next available rotor and start the process all over again.

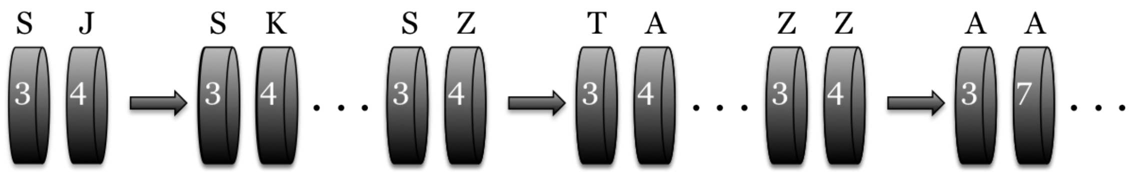


Figure 6: Incrementing the Stators in the Hill-Climb

For each setting we analyze the text and compare it to the previous output using the index of coincidence, or IC. The index of coincidence is technique developed by William F. Friedman, which determines the probability that two random ciphertext letters correlate to the same plaintext character. If the plaintext generated by our putative stators has an IC close to 1.0 this means that the alphabet of the plaintext is evenly distributed. However, this is not the case for the English language, where the IC is close to 1.7; therefore, as the stators approach the correct settings the plaintext should become less random, and the IC of the putative plaintext should begin to climb from 1.0 to 1.7 [8].

If the new settings produce an IC greater than the IC of the previous settings, then we keep the current settings and repeat the process. If the new IC is less than the IC of previous attempt then we stop [19]. The point of the hill-climb technique is that we continue to try new settings so long as the IC produced by the decrypted ciphertext continues to increase. Once the settings begin to generate ICs that decline we stop and assume that the settings with the highest IC is the putative key for the stators. The pseudo code for this initial hill-climb design is in Table 4.

However, with the stators, this approach requires a few slight changes to it. Unlike a typical substitution cipher, where the index of coincidence gradually improves as the guessed key approaches the actual key, the IC of the Typex stators there is very little variation between one setting and the next unless the setting is close to

Table 4: Pseudo Code of Initial Hill Climb Attack

```
// attack assumes stepping rotor settings are known
guess the stator settings
prevIC = currIC = 0
while prevIC $\leq$ currIC
    run Typex simulator
    currIC = IC of text from simulator
    if currIC $\geq$ prevIC
        prevIC = currIC
        set putative settings to current settings
    end if
increment settings
```

the actual key. To ensure that the attack does not stop prematurely, instead of stopping strictly when the IC starts to decrease, we add an additional check to see if the difference between the previous IC and the current IC is greater than 0.25. We choose 0.25 because it provides enough of a buffer between random changes in the IC from a series of incorrect stator settings and a genuine increase in the IC because we are close to the actual stator settings. Table 5 shows the improved attack.

Table 5: Pseudo Code of Revised Hill-Climb Attack

```
guess the stator settings
prevIC = currIC = 0
while prevIC $\leq$ currIC
    run Typex simulator
    currIC = IC of text from simulator
    if currIC $\geq$ prevIC OR (prevIC $-$ currIC) $<$ .25
        prevIC = currIC
        set putative settings to current settings
    end if
increment settings
```

3.3 Some Shortcuts are Better than None

Because of the way the Typex stators work the hill-climbing attack amounts to little more than a brute force attack. For both the hill-climb and brute force attacks we would have to check all $5 \cdot 4 \cdot 2^2 \cdot 26^2$, about $2^{15.7}$, possible stator settings in the worst-case scenario. This is not a very large number given the power of modern computers, but the bulk of the work comes from having to decrypt the ciphertext $2^{15.7}$ times, which we have to do in either attack, making our hill-climb approach on the stators somewhat useless. Therefore, the overall amount of work required to recover the stators is the total number of stator settings multiplied by the size of the ciphertext, in our case 200 characters, which is about $2^{23.3}$ operations.

This differs for the crib attack on the stepping rotors. While we still have to check all possible $8 \cdot 7 \cdot 6 \cdot 2^3 \cdot 26^3$ keys, approximately $2^{25.5}$, we reduce the amount of work done during each iteration by having the cycles. Rather than completely decrypting the ciphertext each time, we simply check the cycles and see if the equation holds true for that particular setting. Therefore the amount of work involved in recovering the rotors is the total number of rotor settings multiplied by the number of cycles required. The Typex requires about nine or ten cycles to reduce the keyspace to one rotor setting; as a result, it takes about $2^{28.8}$ operations to find the correct rotor settings.

Despite having to use a brute force attack to recover the stators, we can still break the attack up into two parts, rotors and stators, decreasing the amount of work that we need to do. Instead of attempting all $2^{41.2}$ keys at once, we only need to check $2^{25.5}$ keys for the rotors plus $2^{15.7}$ for the stators. Even after considering the number of operations performed per iteration of each key this still only amounts to slightly more than $2^{28.8}$ operations in total to recover both the stator and the rotor settings, a vast improvement over a brute force attack on the Typex system as a whole.

These numbers are easily feasible by modern standards, but given WWII technology it would have been more difficult, though not impossible, for the Germans to attempt these two attacks on any intercepted Typex messages. In fact, even though the Typex includes improvements on the design of the commercial Enigma, the amount of work involved in cracking the Typex is still slightly less than that of the military Enigma [1, 7]. The only difficulty the Germans may have faced was in intercepting enough ciphertext to create the necessary number of cycles, but even this would have been conceivable given that the Germans transmitted over two million words a day [16]. Two factors affected Germany's ability to cryptanalyze the Typex: their blind trust in the Enigma's security and a lack of resources [7].

CHAPTER 4

Hill-Climb for Rotor Wirings

Hill-climbing is a heuristic search technique. When hill-climbing the stators, we assume that the rotor wirings are known and therefore only have a choice of $5 \cdot 4 \cdot 2^2 \cdot 26^2$, or about $2^{15.7}$ possibilities. However, if the rotor wirings are unknown to the attacker then there are 26! different wiring combinations per rotor. The amount of work involved in recovering unknown rotor wirings amounts to approximately $2^{88.4}$ per rotor. Because there is such a large key space for the rotor wirings we must use a heuristic algorithm in order to solve for them in polynomial time. Heuristic algorithms use heuristic methods to solve a given problem. A heuristic method is a series of modifications made to a guessed solution to produce a different solution. While heuristic algorithms cannot guarantee success, for certain types of problems they can do well at determining good or close to optimal solutions in a much faster time than other algorithms [5].

4.1 Scoring the Wiring

Much like the stators, when we run our hill-climbing algorithm on the rotor wirings we require some measure of how close we are to the original rotor wirings. Instead of using the index of coincidence, as we did with the stators, we use digram frequencies, the frequency of a given letter to be followed by another letter. We apply this method to score the wiring versus the index of coincidence because we only need the digram frequency of the ciphertext in order to break a simple substitution cipher [9], which is what a rotor wiring is. However, we cannot incorporate the traditional English digram matrix (Figure 7) for two reasons: first, it includes the space character, which Typex automatically converts to an X, and second it is possible to have the correct rotor wiring but the wrong initial position, meaning that the putative rotor is shifted from the original. A typical digram matrix cannot keep track of the position of a rotor nor any kind of shift in the alphabet due to the rotor being offset from the original setting; therefore, we create a new one using the text from [12, 13, 20]. After generating our own digram frequency matrix, we take the sum of the columns of the matrix and create a new matrix that has the shifted versions of the sums, shown in Figure 8.

4.2 Perfecting the Swap

Hill-climbing the stators proved that iteratively changing the solution by incrementally increasing the settings does not work well with Typex. Similarly, due to the large number of possible rotor wirings, we cannot simply pick a starting point and then start incrementing the wirings. Instead, we must design a swapping algorithm that covers enough permutations and shuffles the wirings enough to encourage a good success rate. By doing so, we will be able to cover a larger variety of wirings rather than remaining close to our initial wiring.

We start with the two algorithms described in [5]. The first method we call the fast swap method. In this method, we select the wirings to swap based on a series of incrementing rounds. For example, given a N-letter rotor with wirings W_1 to W_N, in the first round, we would swap W_1 with W_2, W_2 with W_3, and so forth. In the second round, we swap W_1 and W_3, W_2 and W_4, etc.. We continue to swap in this manner until we reach the last round, where we only swap W_1 with W_N. In total, with the fast swap method, we achieve $\binom{N}{2}$ swaps.

We call the second method in [5] the slow hill-climb. The slow hill-climb only

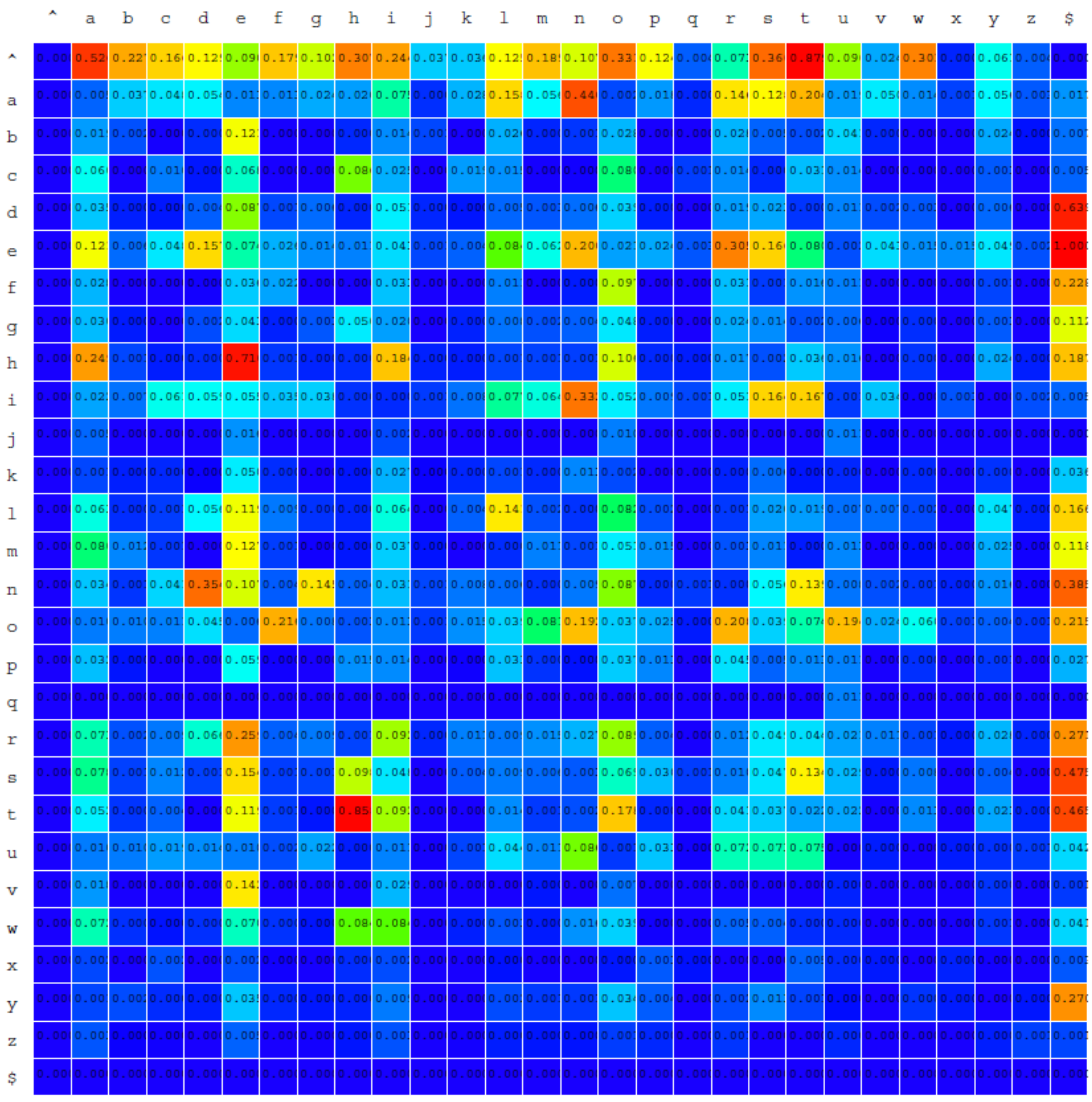
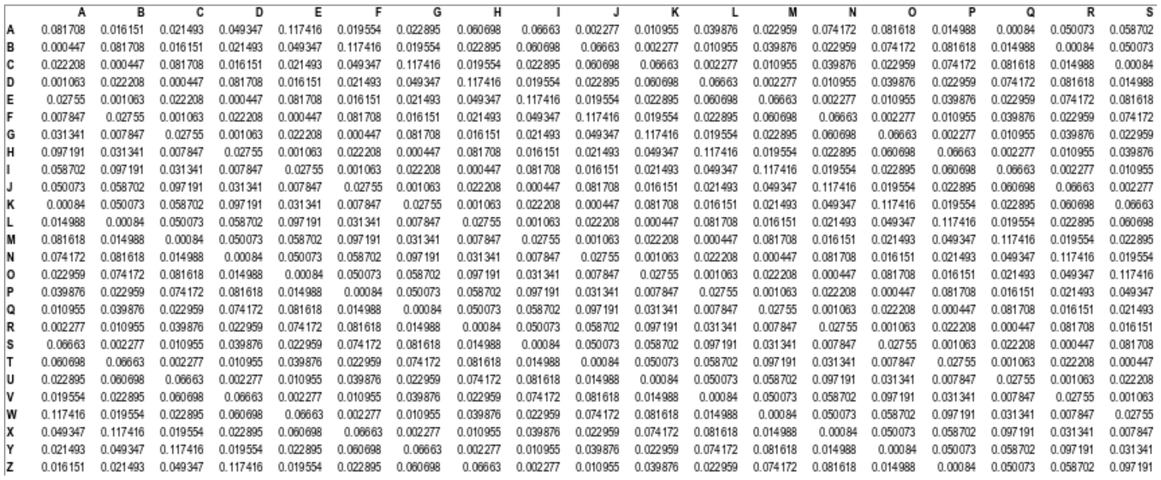


Figure 7: English Digram Frequency Matrix [3]

swaps adjacent wirings; however, it will continue to swap until it runs through the wiring once and fails to keep any changes. The slow hill-climb has the potential to swap a much greater number of wirings as long as the score continues to improve.

Similar to hill-climbing the stators, we find that because of the incremental nature of the slow hill-climb, unless our first guess at the rotor wirings is already close to



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
A	0.081708	0.016151	0.021493	0.049347	0.117416	0.019554	0.022895	0.060698	0.06663	0.002277	0.010955	0.039876	0.022959	0.074172	0.081618	0.014988	0.00084	0.050073	0.058702
B	0.000447	0.081708	0.016151	0.021493	0.049347	0.117416	0.019554	0.022895	0.060698	0.06663	0.002277	0.010955	0.039876	0.022959	0.074172	0.081618	0.014988	0.00084	0.050073
C	0.022208	0.000447	0.081708	0.016151	0.021493	0.049347	0.117416	0.019554	0.022895	0.060698	0.06663	0.002277	0.010955	0.039876	0.022959	0.074172	0.081618	0.014988	0.00084
D	0.001063	0.022208	0.000447	0.081708	0.016151	0.021493	0.049347	0.117416	0.019554	0.022895	0.060698	0.06663	0.002277	0.010955	0.039876	0.022959	0.074172	0.081618	0.014988
E	0.02755	0.001063	0.022208	0.000447	0.081708	0.016151	0.021493	0.049347	0.117416	0.019554	0.022895	0.060698	0.06663	0.002277	0.010955	0.039876	0.022959	0.074172	0.081618
F	0.007847	0.02755	0.001063	0.022208	0.000447	0.081708	0.016151	0.021493	0.049347	0.117416	0.019554	0.022895	0.060698	0.06663	0.002277	0.010955	0.039876	0.022959	0.074172
G	0.031341	0.007847	0.02755	0.001063	0.022208	0.000447	0.081708	0.016151	0.021493	0.049347	0.117416	0.019554	0.022895	0.060698	0.06663	0.002277	0.010955	0.039876	0.022959
H	0.097191	0.031341	0.007847	0.02755	0.001063	0.022208	0.000447	0.081708	0.016151	0.021493	0.049347	0.117416	0.019554	0.022895	0.060698	0.06663	0.002277	0.010955	0.039876
I	0.058702	0.097191	0.031341	0.007847	0.02755	0.001063	0.022208	0.000447	0.081708	0.016151	0.021493	0.049347	0.117416	0.019554	0.022895	0.060698	0.06663	0.002277	0.010955
J	0.050073	0.058702	0.097191	0.031341	0.007847	0.02755	0.001063	0.022208	0.000447	0.081708	0.016151	0.021493	0.049347	0.117416	0.019554	0.022895	0.060698	0.06663	0.002277
K	0.00084	0.050073	0.058702	0.097191	0.031341	0.007847	0.02755	0.001063	0.022208	0.000447	0.081708	0.016151	0.021493	0.049347	0.117416	0.019554	0.022895	0.060698	0.06663
L	0.014988	0.00084	0.050073	0.058702	0.097191	0.031341	0.007847	0.02755	0.001063	0.022208	0.000447	0.081708	0.016151	0.021493	0.049347	0.117416	0.019554	0.022895	0.060698
M	0.081618	0.014988	0.00084	0.050073	0.058702	0.097191	0.031341	0.007847	0.02755	0.001063	0.022208	0.000447	0.081708	0.016151	0.021493	0.049347	0.117416	0.019554	0.022895
N	0.074172	0.081618	0.014988	0.00084	0.050073	0.058702	0.097191	0.031341	0.007847	0.02755	0.001063	0.022208	0.000447	0.081708	0.016151	0.021493	0.049347	0.117416	0.019554
O	0.022959	0.074172	0.081618	0.014988	0.00084	0.050073	0.058702	0.097191	0.031341	0.007847	0.02755	0.001063	0.022208	0.000447	0.081708	0.016151	0.021493	0.049347	0.117416
P	0.039876	0.022959	0.074172	0.081618	0.014988	0.00084	0.050073	0.058702	0.097191	0.031341	0.007847	0.02755	0.001063	0.022208	0.000447	0.081708	0.016151	0.021493	0.049347
Q	0.010955	0.039876	0.022959	0.074172	0.081618	0.014988	0.00084	0.050073	0.058702	0.097191	0.031341	0.007847	0.02755	0.001063	0.022208	0.000447	0.081708	0.016151	0.021493
R	0.002277	0.010955	0.039876	0.022959	0.074172	0.081618	0.014988	0.00084	0.050073	0.058702	0.097191	0.031341	0.007847	0.02755	0.001063	0.022208	0.000447	0.081708	0.016151
S	0.06663	0.002277	0.010955	0.039876	0.022959	0.074172	0.081618	0.014988	0.00084	0.050073	0.058702	0.097191	0.031341	0.007847	0.02755	0.001063	0.022208	0.000447	0.081708
T	0.060698	0.06663	0.002277	0.010955	0.039876	0.022959	0.074172	0.081618	0.014988	0.00084	0.050073	0.058702	0.097191	0.031341	0.007847	0.02755	0.001063	0.022208	0.000447
U	0.022895	0.060698	0.06663	0.002277	0.010955	0.039876	0.022959	0.074172	0.081618	0.014988	0.00084	0.050073	0.058702	0.097191	0.031341	0.007847	0.02755	0.001063	0.022208
V	0.019554	0.022895	0.060698	0.06663	0.002277	0.010955	0.039876	0.022959	0.074172	0.081618	0.014988	0.00084	0.050073	0.058702	0.097191	0.031341	0.007847	0.02755	0.001063
W	0.117416	0.019554	0.022895	0.060698	0.06663	0.002277	0.010955	0.039876	0.022959	0.074172	0.081618	0.014988	0.00084	0.050073	0.058702	0.097191	0.031341	0.007847	0.02755
X	0.049347	0.117416	0.019554	0.022895	0.060698	0.06663	0.002277	0.010955	0.039876	0.022959	0.074172	0.081618	0.014988	0.00084	0.050073	0.058702	0.097191	0.031341	0.007847
Y	0.021493	0.049347	0.117416	0.019554	0.022895	0.060698	0.06663	0.002277	0.010955	0.039876	0.022959	0.074172	0.081618	0.014988	0.00084	0.050073	0.058702	0.097191	0.031341
Z	0.016151	0.021493	0.049347	0.117416	0.019554	0.022895	0.060698	0.06663	0.002277	0.010955	0.039876	0.022959	0.074172	0.081618	0.014988	0.00084	0.050073	0.058702	0.097191

Figure 8: Digram Frequency Matrix with Shift Allowance

Table 6: Pseudo Code of Fast Swap Method

```
—MAIN—
while round < $N$
    SWAP(round,position)
    if newscore better
        keep wiring
    else
        swap back
    end if
    increment position
    if position+round > $N-1$
        reset position
        increment round
    end if
end while

—SWAP(round,position)—
temp = wire[position]
wire[position] = wire[position+round]
wire[position+round] = temp
```

the original wirings, we do not make very many swaps before stopping because the wiring has not changed for an entire loop. Therefore, despite the fact that the fast

Table 7: Pseudo Code of Slow Hill-Climb Method

```
—MAIN—
while wiring continues to change
    SWAP(round,position)
    if newscore better
        keep wiring
    else
        swap back
    end if
    increment position
    if position > $N-2$
        reset position
    end if
end while

—SWAP(position)—
temp = wire[position]
wire[position] = wire[position+1]
wire[position+1] = temp
```

swap method has a fixed number of swaps, it still results in a better putative wiring than the slow hill-climb.

To improve the fast swap method we decide to incorporate a few changes. Namely, instead of having a fixed $\binom{N}{2}$ swaps, we continue to swap until the wiring stops improving. We achieve this by comparing the wiring after the $\binom{N}{2}$ swaps to the wiring before the $\binom{N}{2}$ swaps. The downside to this addition is that we will always perform $\binom{N}{2}$ swaps more than necessary. In the case of one unknown 26-letter rotor, we only perform an extra 325 swaps, but this number increases exponentially with each additional unknown rotor.

To mitigate this situation we try a variation on our improved fast swap method. We swap as the previous method, but instead of having a fixed number of swaps we re-

Table 8: Pseudo Code of Fast Swap Method with Wiring Check

```
—MAIN—
while previous wiring $\neq$ current wiring
    previous wiring = current wiring
    while round $< N$
        SWAP(round,position)
        if newscore better
            keep wiring
        else
            swap back
        end if
        increment position
        if position+round $> N-1$
            reset position
            increment round
        end if
    end while
end while
```

set the position and round counters when we encounter a better wiring. Interestingly, we find that this variation does not perform as well as the method with the wiring comparison, but also that more swaps are necessary than our previous version of the fast swap method. It could be that by resetting the two counters we inadvertently find local optimum solutions instead of the global optimum.

4.3 Hill-Climbing and Nesting Inside Typex

With our swap method selected we move on to adding the swap inside of the simulator. Because the wiring of the rotors changes with each swap, we initialize the rotors and run the simulator within the loop of the swap method. Additionally, in the case of multiple unknown rotors, we must pay attention to perform the initialization and simulation within the innermost swap loop. This ensures that no matter which

Table 9: Pseudo Code of Fast Swap Method with Counter Reset

```
—MAIN—
while round < $N$
    SWAP(round,position)
    if newscore better
        reset position and round counters
        keep wiring
    else
        swap back
    end if
    increment position
    if position+round > $N-1$
        reset position
        increment round
    end if
end while
```

wiring is swapped we score the most current version of the wiring.

When we start to add more unknown rotors to our hill-climbing algorithm we set the rightmost rotor to be the outer loop and work our way left as we nest the swap method within itself. The order in which we recover the rotor wirings should not matter. We choose to nest the slowest rotor in the innermost loop to allow it the most opportunities to try different wirings, with the hope that doing so encourages it to find the correct wiring with fewer swaps than if it were on the outermost loop. Appendix B shows that the choice of which rotor we nest on the inner loop versus the outer loop does not matter.

4.4 Hill-Climbing is Hard Work

As previously mentioned the basic number of swaps is $\binom{N}{2}$. This number is then multiplied by the number of times that we repeat the loop as the putative wiring

Table 10: Pseudo Code of Simulator Inside Swap Method

```
—MAIN—
while previous wiring $\neq$ current wiring
    previous wiring = current wiring
    while round $< N$
        initialize rotors
        run simulator
        calculate score
        if newscore better
            keep wiring
        else
            swap back
        end if
        SWAP(round,position)
        increment position
        if position+round $> N-1$
            reset position
            increment round
        end if
    end while
end while
```

continues to change. For one rotor we found the average number of swaps to be $2^{10.4}$, which means that our loop ran, on average, about four times.

While this might not seem like that many swaps compared to the total $2^{88.4}$ possible rotor wirings, the bulk of the work comes from having to run the Typex simulator for each swap. The amount of work involved in the simulation depends largely on the amount of text given to the simulator. A small message of 100 characters multiplied by $2^{10.4}$ is approximately $2^{17.1}$, certainly a large number but not overwhelming; however, because of Typex's multiple notched stepping mechanism, we require a much larger amount of text than 100 characters. Our tests in Section 5 show that with one unknown rotor, at the very least, we need 1000 characters and on average most initial

Table 11: Pseudo Code of Nested Hill-Climb

```
—MAIN—
while previous right wiring $\neq$ current right wiring
    previous right wiring = current right wiring
    while right round $< N$
        while previous middle wiring $\neq$ current middle wiring
            previous middle wiring = current middle wiring
            while middle round $< N$
                initialize rotors
                run simulator
                calculate score
                if middle newscore better
                    change right newscore
                    keep current middle wiring
                else
                    swap back
                end if
                SWAP(middle round,middle position)
                increment middle position
                if middle position+middle round $> N-1$
                    reset middle position
                    increment middle round
                end if
            end while
            if right newscore better
                keep current right wiring
            else
                swap back
            end if
            SWAP(right round,right position)
            increment right position
            if right position+right round $> N-1$
                reset right position
                increment right round
            end if
        end while
    end while
end while
```

wiring guesses require around $2^{15.6}$ characters. When we combine this number with the number of swaps required the result is approximately 2^{26} operations to recover the wiring of one unknown rotor. While this may seem like a large number, it is a huge improvement over the brute force approach, which requires 2^{87} on average.

With each additional unknown rotor this number increases exponentially. In the case of two unknown rotors, the minimum amount of data our hill-climb algorithm requires is $2^{12.3}$ characters. On average, we need 2^{20} characters. Two unknown rotors average about 2^{21} swaps to recover both wirings. In total, this amounts to $2^{21} \cdot 2^{20}$, or approximately 2^{41}, operations. Strangely, this is much less than the expected $2^{(26 \cdot 2)}$, or 2^{52}, that we expect given the numbers from our one unknown rotor test.

While the number of swaps required grew exponentially, the amount of text required did not. This could possibly be due to the fact that we used the same text in our tests as we did to generate the digram matrix. We created the digram matrix using approximately $2^{19.4}$ characters. Therefore, when running our two rotor tests we reused some of the text for the larger character tests. It would be interesting to see the results of a rotor wiring hill-climb where the digram matrix text and text used in the tests did not overlap.

CHAPTER 5

Analyzing the Hill-Climb

5.1 One Rotor at a Time

We begin our wiring hill-climb with just one unknown rotor and assume the remaining rotors' initial positions, order, and orientations are known. Because the signal must first travel through the stators, the choice of which rotor should be the unknown one does not matter. In our case, we select the rightmost rotor. We create a random wiring generator and use it to create the wirings we set as our initial guesses in the hill-climbing algorithm.

As we can see from Figure 9, our hill-climb does not have a one hundred percent success rate; however, as a hill-climb we do not expect such. Out of 2905 different initial wirings, we recovered the actual wiring used, or were within 5 wirings of the original, 2241 times, providing us with a success rate of 0.771. This is a good success rate and shows that our hill-climb technique works well for one unknown rotor.

With only one unknown rotor, the time it takes to recover the rotor wirings is fairly minimal as shown in Table 12. One interesting thing to note is that as the amount of data increases the number of swaps decreases. However, we can see that the time it takes to run the simulator on such a large amount of text still outweighs the benefit of the decreased swapping.

5.2 Increasing the Number of Unknowns

After verifying that hill-climbing one rotor is possible, we move on to two unknown rotors, the right and middle rotors. As previously mentioned, the amount of

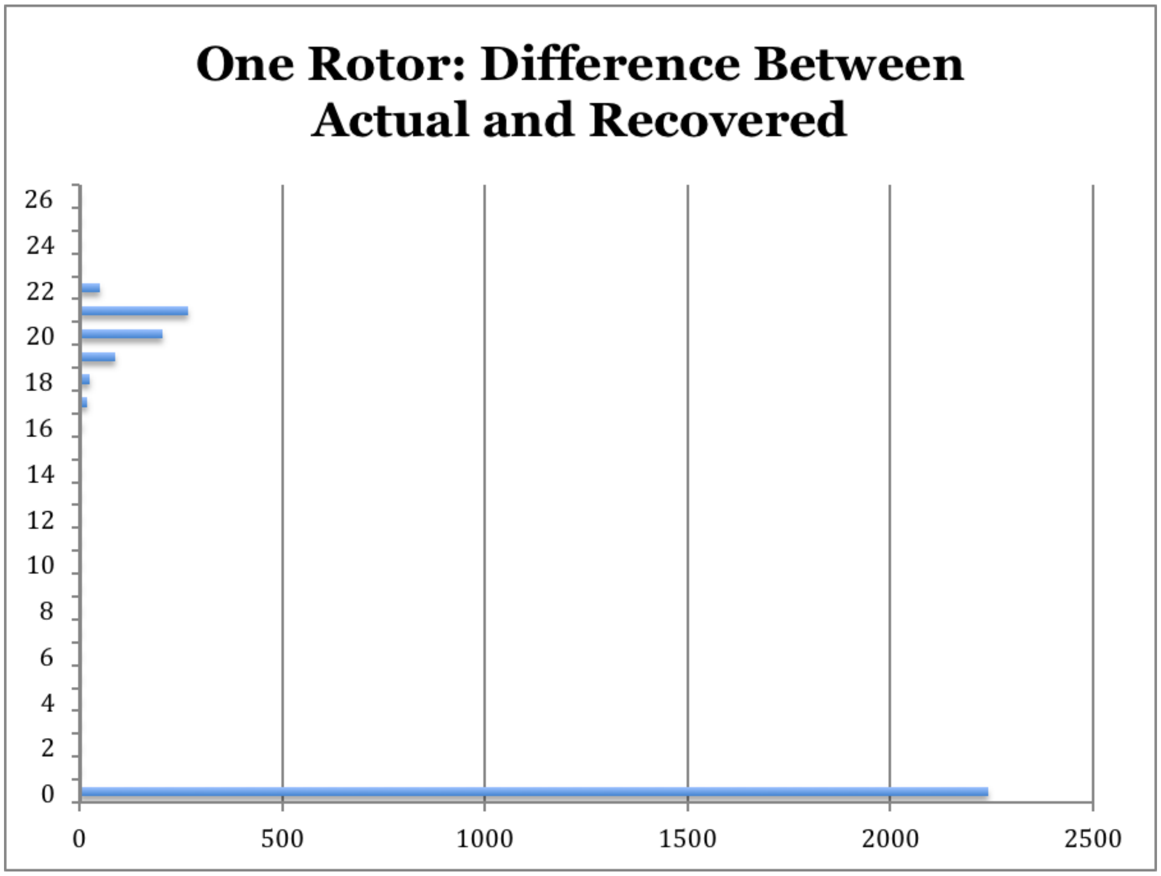


Figure 9: One Rotor Hill-Climbing Results

Data Used (Characters)	1000	2000	5000	10000	25000	50000	75000	100000
Average Swaps	1365	1380	1419	1420	1390	1342	1318	1273
Average Time (secs)	1.439	1.867	2.809	4.107	6.091	11.964	14.394	17.236
Number of Successes	1	15	260	506	918	1472	1559	1770
Number of Test Cases	2000	2000	2000	2000	2000	2000	2000	2000

Table 12: Comparison of Data Used, Average Swaps, Average Time, Number of Successes, and Number of Test Cases (One Rotor)

time it takes to recover two unknown wirings is exponentially longer than that of just one because the number of swaps increases exponentially. Additionally, the amount of data necessary to recover the wirings increases.

For our two unknown rotors test, we use the same initial wirings as the one rotor test. As can be seen in Figure 10, we recover the correct wiring of the middle rotor only 1774 times for a rate of 0.611, about 80% of the amount recovered in the one rotor test. Also notable is the dispersion of the failed attempts. There is a much higher concentration in the high teens and low twenties than the previous test, with some initial guessing points only managing to recover two of the correct wirings. This remains an impressive result despite the drop in number of wirings correctly found.

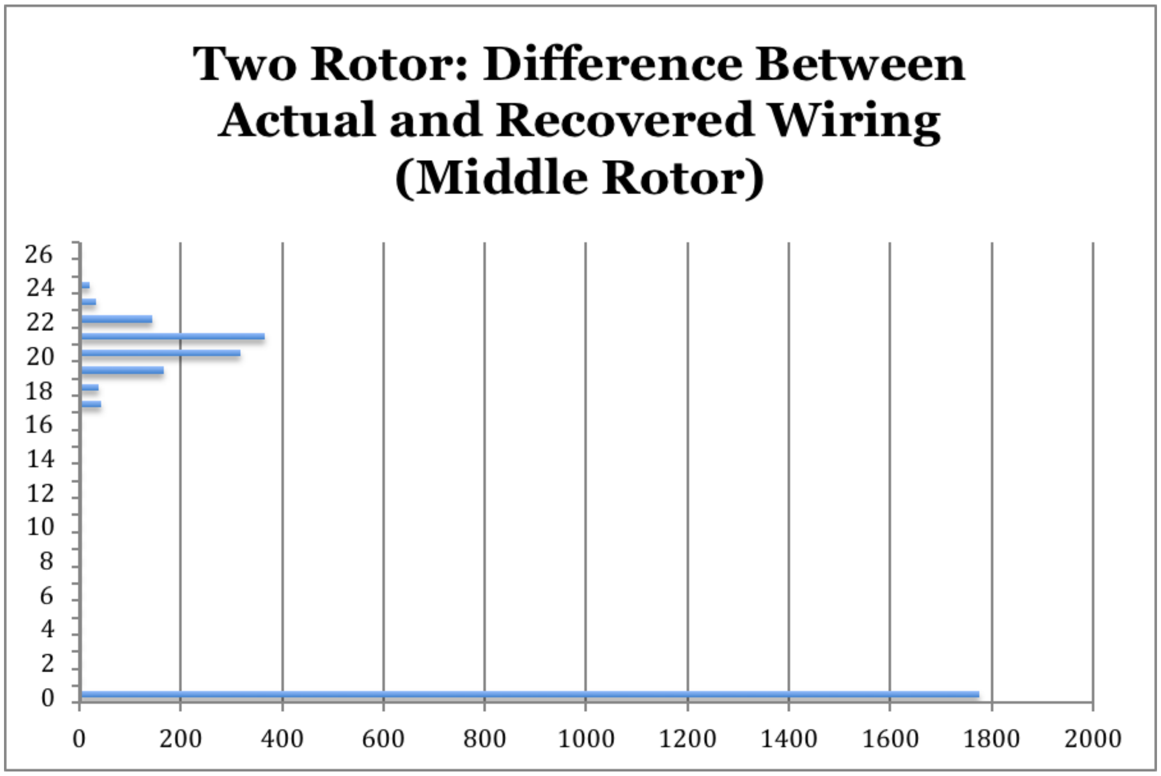


Figure 10: Two Rotor Hill-Climbing Results (Middle Rotor)

Much like the middle rotor, the spread of the failed attempts continues to climb towards the twenties. Because recovering the right rotor is related to the recovery of the middle rotor, we only recover the right rotor 1043 times out of 2905 attempts. This is approximately 60% the amount of middle rotor wirings recovered. In this test, when we successfully recover the right rotor we also successfully recover the middle rotor. There are no situations where the right rotor is recovered but the middle rotor is not. Therefore, given two unknown rotors we have a success rate of 0.359, meaning that the chances of recovering both wirings is about 35.9%.

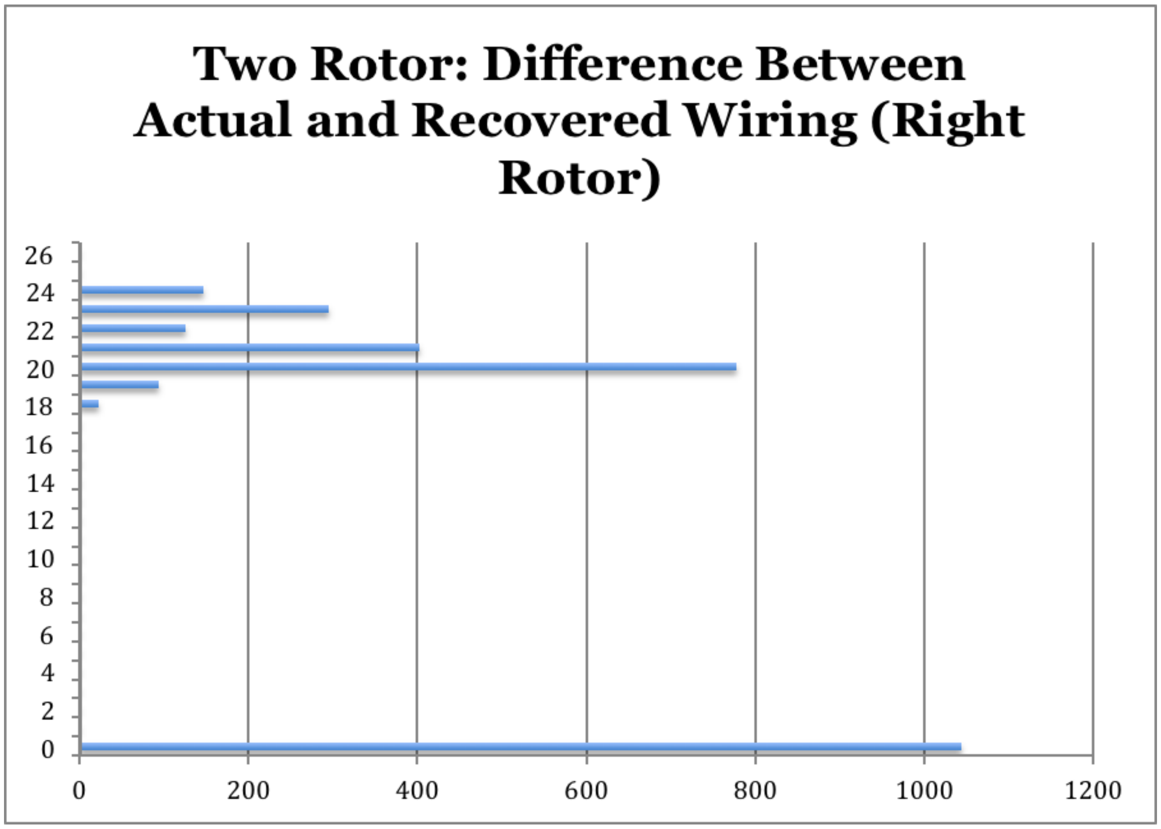


Figure 11: Two Rotor Hill-Climbing Results (Right Rotor)

As expected, the average number of swaps increases roughly by the amount of swaps squared. Likewise, the time also increases based on the number of times the simulator needs to be run.

Data Used (Characters)	50000	100000	250000	500000	750000	1000000
Average Swaps	2047712	2056453	2065281	2054810	2051334	2042556
Average Time (secs)	3168.287	5282.438	8427.554	16228.515	21708.101	24482.764
Number of Successes	7	75	144	203	291	387
Number of Test Cases	1000	1000	1000	1000	1000	1000

Table 13: Comparison of Data Used, Average Swaps, Average Time, Number of Successes, and Number of Test Cases (Two Rotors)

CHAPTER 6

The Polish Attack on the Wirings

Marian Rejewski was a brilliant Polish mathematician and cryptanalyst. His contributions in breaking Enigma undoubtedly made it much easier for those at Bletchley Park to decipher the machine. By far one of the biggest breakthroughs was the recovery of the internal wirings of the Enigma's rotors. His attack exploited the Enigma operators' use of a repeated message key at the beginning of each message. Through the use of disjoint cycles in the repeated message key and theorems on transpositions, Rejewski connected the wirings in Enigma's rotors [4].

Each day Enigma operators were given a new key to use. However to avoid having millions of characters enciphered with same key, only six letters per message used the day key. These six letters were the message key, which was just three letters repeated twice. The Germans implemented the repetition to avoid mistakes from operator error and signal interference. However, this repetition allowed Rejewski to recover the rotor wirings [16].

Over the course of a day, Rejewski collected enough message keys allowing him to find cycles between the first and fourth letters, second and fifth letters, and third and sixth letters. He labeled these cycles AD, BE, and CF, respectively. Equation (6) is an example of these cycles.

$$\begin{aligned} AD &= (kxgzyodvpf)(nqlhteijmu)(bc)(rw)(a)(s) \\ BE &= (veoumblfq)(wizrnhjps)(xta)(gyc)(d)(k) \\ CF &= (jgfcqnyabvikt)(lxwpsmoduzreh) \end{aligned} \tag{6}$$

Rejewski's goal was to factor these cycles into their individual components: A, B, C, D, E, and F. To do so he made use of a couple of theorems on transpositions and the knowledge that the message keys used by the Enigma operators were not entirely chosen at random. The main theorem used to factorize the cycles was Rejewski's converse to the theorem on the product of transpositions. The theorem basically states that if $XY = (a_1a_3)(a_2a_4)$ then $X = (a_1a_4)(a_2a_3)$ and $Y = (a_1a_2)(a_3a_4)$ [4].

From our example we can clearly see that (as) is in both A and D based on the converse to the theorem on the product of transpositions. Similarly, either $(br)(cw)$ is in A or $(bw)(cr)$ is in A, which means that $(cr)(bw)$ or $(cw)(br)$ is in D, depending on which is in A. For the last pair of cycles, $(kxgzyodvpf)(nqlhteijmu)$, there are 10 possibilities, forming a total of 20, $1 \cdot 2 \cdot 10$, factorizations. Rejewski knew that the Enigma operators did not choose the message key at random. They often used initials or patterns in the keyboard to select the key. Because of this knowledge Rejewski could narrow down the number of possible factorizations [4].

With the individual factorizations known, Rejewski then looked at the permutations at each step. He then assumed that the middle and left rotors, R_m and R_l, did not step while the operator typed the message key. This allowed him to reduce $R_mR_lTR_l^{-1}R_m^{-1}$ to a single variable Q, leaving him only three unknowns R_r, Q, and S.

$$\begin{aligned}
A &= SP^1R_rP^{-1}R_mR_lTR_l^{-1}R_m^{-1}P^1R_r^{-1}P^{-1}S^{-1} \\
B &= SP^2R_rP^{-2}R_mR_lTR_l^{-1}R_m^{-1}P^2R_r^{-1}P^{-2}S^{-1} \\
C &= SP^3R_rP^{-3}R_mR_lTR_l^{-1}R_m^{-1}P^3R_r^{-1}P^{-3}S^{-1} \\
D &= SP^4R_rP^{-4}R_mR_lTR_l^{-1}R_m^{-1}P^4R_r^{-1}P^{-4}S^{-1} \\
E &= SP^5R_rP^{-5}R_mR_lTR_l^{-1}R_m^{-1}P^5R_r^{-1}P^{-5}S^{-1} \\
F &= SP^6R_rP^{-6}R_mR_lTR_l^{-1}R_m^{-1}P^6R_r^{-1}P^{-6}S^{-1}
\end{aligned} \tag{7}$$

where P^i is

$$\begin{aligned} P^1 &= (abcdefghijklmnopqrstuvwxyz) \\ P^2 &= (acegikmoqsuwy)(bdfhjlnprtvxz) \\ P^3 &= (adgjmpsvybehknqtwzcfilorux) \\ P^4 &= (aeimquycgkosw)(bfjnrzvdhlptx) \\ P^5 &= (afpuzejotydinsxchmrwbglqv) \\ P^6 &= (agmsyekqwciou)(bhntzflrxdjpv) \end{aligned} \tag{8}$$

Luckily, the French came across documents that included the daily keys, which they sent to the Poles due to a military cooperation agreement signed 10 year earlier [16]. With only two unknowns, Rejewski reordered his permutation equations, placing all known variables on one side of the equation, and re-labeled them [4].

$$\begin{aligned} U &= P^{-1}S^{-1}ASP^1 = R_rP^{-1}QP^1R_r^{-1} \\ V &= P^{-2}S^{-1}BSP^2 = R_rP^{-2}QP^2R_r^{-1} \\ W &= P^{-3}S^{-1}CSP^3 = R_rP^{-3}QP^3R_r^{-1} \\ X &= P^{-4}S^{-1}DSP^4 = R_rP^{-4}QP^4R_r^{-1} \\ Y &= P^{-4}S^{-1}ESP^5 = R_rP^{-5}QP^5R_r^{-1} \\ Z &= P^{-6}S^{-1}FSP^6 = R_rP^{-6}QP^6R_r^{-1} \end{aligned} \tag{9}$$

Because Rejewski only had two unknowns he only needed to factorize four of these equations. With the factorizations of U, V, W, and X, Rejewski formed the products UV, VW, and WX. He noted that each product was a conjugate of $QP^{-1}QP^1$ and that the products had the same configuration of cycles, which allowed him to rewrite the equations and eliminate the common expression [4].

$$\begin{aligned}
UV &= (R_r P^{-1} Q P^1 R_r^{-1})(R_r P^{-2} Q P^2 R_r^{-1}) \\
&= R_r P^{-1}(Q P^{-1} Q P^1) P^1 R_r^{-1} \\
VW &= (R_r P^{-2} Q P^2 R_r^{-1})(R_r P^{-3} Q P^3 R_r^{-1}) \\
&= R_r P^{-2}(Q P^{-1} Q P^1) P^2 R_r^{-1} \\
&= (R_r P^1 R_r^{-1})^{-1}(UV) R_r P^1 R_r^{-1} \\
WX &= (R_r P^{-3} Q P^3 R_r^{-1})(R_r P^{-4} Q P^4 R_r^{-1}) \\
&= R_r P^{-3}(Q P^{-1} Q P^1) P^3 R_r^{-1} \\
&= (R_r P^1 R_r^{-1})^{-1}(VW) R_r P^1 R_r^{-1}
\end{aligned} \tag{10}$$

Rejewski then reordered the cycles UV, VW, and WX such that the pairs of cycles $UV/\ VW$ and $VW/\ WX$ resulted in the same $R_r P^1 R_r^{-1}$. Since P^1 was known, this meant that Rejewski successfully recovered the right rotor [4].

Rejewski's attack, with the same assumptions, also works on the Typex rotors. Unfortunately, the assumption that the middle and left rotors do not step is not a feasible one given the multi-notched rims on the Typex rotors. The attack does not work if either rotor steps as that changes the permutation of the signal, or variable Q in the equations. So while this assumption and attack may have worked for the rotors with five notches; it likely would not have been valid for the rotors with nine notches.

CHAPTER 7

Conclusion

When it became clear to the British that codebook ciphers and other methods used in World War I were no longer sufficient, they began to search for other ways to encrypt sensitive information. The result of their search was the Typex cipher machine, an incredibly close cousin of the German Enigma machine. The Typex swapped out the stecker for two stators (static rotors) and included a printer instead of a light board. Most importantly, the Typex increased the security of the Enigma by stepping more frequently, anywhere from five to nine times per rotation of its neighboring rotor.

While the frequent stepping may have been an improvement over the Enigma, it certainly does not make Typex immune to attacks that work on the Enigma. The Enigma cycle attack, which was developed by Alan Turing during World War II, also works on the Typex. In the case of Typex, we need a few more cycles than the Enigma to narrow the putative keys down to one. The increased number of cycles means that the attack requires a longer crib, which would have been slightly more difficult for the Germans to obtain, but certainly not impossible.

Another challenge the Germans would have faced was the stators, which lack the reciprocal property of the stecker, meaning that their settings could not be recovered with a little extra work on the cycles used to find the stepping rotor settings. To recover the stators, we apply a slightly modified hill-climbing attack that compares the index of coincidences between different settings, but an exhaustive search is also very easy.

While the crib attack requires that the attacker attempt all possible settings for the stepping rotors, the cycles provide a way to quickly eliminate incorrect keys without having to check any ciphertext. They also allow us to ignore the stators, which in turn means that we can break the Typex key up into two parts: rotors and stators. Even if the attacker must check all $2^{15.7}$ possible stator settings against 200 characters of ciphertext, the total amount of work required to recover the correct stator settings is only $2^{23.3}$ operations. This is less than the amount of work required to recover the rotor settings, which is the total number of possible settings multiplied the number of cycles, about $2^{28.8}$ operations. The combination of these two attacks creates enough of a shortcut to reduce the amount of work from $2^{41.2}$ to slightly more than $2^{28.8}$.

The previous two attacks assume that the attacker knows the rotor wirings, as the British did with the German Enigma machine. However, if the rotor wirings are unknown then those must be recovered first. To do this we employ a nested hill-climbing technique. We carefully chose a swapping algorithm that covers a variety of wirings but does not waste too much time in doing so. To measure the accuracy of the wiring we compare it against an English digram matrix modified to allow for all 26 possible positions of the rotor.

Our hill-climbing algorithm proved to be very effective against one unknown rotor, finding the correct wiring, or nearly so, slightly more than 77% of the time. When we add a second unknown rotor the recovery rate of one rotor drops to 61% per rotor, and we recover both rotors 36% of the time.

Similar to how Turing's crib attack successfully finds the settings of the Typex rotors, Rejewski's attack on the wirings is also successful against Typex. However, in order for the attack to be successful, it requires that the same assumptions made with Enigma also hold true for Typex, namely that the operators repeat the message key and that the middle and left rotors do not step during the encipherment of the message key. While this may have held true for the rotors with five notches, it certainly would have been a much rarer occcurence for the rotors with nine notches.

CHAPTER 8

Future Work

While this project shows it is possible to use the nested hill-climbing algorithm to solve for two unknown rotors, it has yet to be tested on three unknown rotors. Clearly, as the number of unknown rotors increases the success rate decreases. The rapid drop from 77% for one unknown rotor to 36% for two unknown rotors leaves some room to question whether solving a third unknown rotor is possible, even with extensive computing power and a large ciphertext.

One way to approach the decreasing recovery rate is to improve upon the swapping algorithm. At its core, the chosen swap method only covers $\binom{N}{2}$ swaps per iteration. There exist other swapping algorithms that shuffle the wirings, much like the fast swap method, and perform more swaps per iteration.

Another possible method of improving the success rate is to use a different heuristic algorithm. Hill-climbing can be a very effective, simple algorithm to solve problems; however, there are several other heuristic algorithms that may be a better option than hill-climbing [1].

A faster scoring method could also be considered. While we thought that the digram matrix would work well because a rotor wiring is analogous to a simple substitution, we were unable to use the method in [9] to avoid running through the putative plaintext every time we swapped a wiring. It would also be interesting to see if using the index of coincidence to score our plaintext could provide better results than the digram matrix.

We successfully recovered the right rotor's wiring using Rejewski's method with the same assumptions. However, given the mutli-notched rims on the Typex rotors it is unlikely that the middle and left rotors do not step during those first six letters. Therefore, it is necessary to find a way to alter Rejewski's attack that takes the rotor stepping into account to fully adapt this attack to the Typex.

LIST OF REFERENCES

[1] Bagnall, A.J., McKeown, G.P, & Rayward-Smith, V.J. (1997). Proceedings from the ICGA, '97: *The Cryptanalysis of a Three Rotor Machine Using a Genetic Algorithm*. San Francisco, CA: Morgan Kaufmann.

[2] Bauer, F.L. (2007). *Decrypted Secrets: Methods & Maxims of Cryptology*. Berlin, Germany: Springer.

[3] Briggs, K. (2004). English and Latin digram and trigram frequencies. Retrieved: 14 January 2012, from Website:
`http://keithbriggs.info/documents/english_latin.pdf`

[4] Christensen, C. (2007). Polish Methematicians Finding Patterns in Enigma Messages. *Mathematics Magazine*, 80(4), 247-273.

[5] Dhavare, A., Efficient attacks on homophonic substitution ciphers, Masters Report, Department of Computer Science, San Jose State University, 2011.

[6] Erskine, R. (1997). The Development of Typex. *The Enigma Bulletin*, 2, 69-85.

[7] Ferris, J.R. (2005). *Intelligence and Strategy: Selected Essays*. New York, NY: Routledge.

[8] Friedman, W.F. (1987). *The Index of Coincidence and Its Applications in Cryptanalysis*. Laguna Hills, CA: Aegean Park Press.

[9] Jakobsen, T. (1995). A Fast Method for the Crytanalysis of Substitution Ciphers. *Cryptologia*, 19(3), 265-274.

[10] Kreher D., Stinson D. (1999). *Combinatorial Algorithms*.

[11] Kruh, L. & Deavours, C.A. (1983). The Typex Cryptograph. *Cryptologia*, 7(2), 145-166.

[12] Lewis, C. (n.d.). *Alices Adventures in Wonderland*. Retrieved from
`http://www.gutenberg.org/cache/epub/11/pg11.txt`

[13] Paine, T. (n.d.). *Common Sense*. Retrieved from
`http://www.gutenberg.org/cache/epub/147/pg147.txt`

[14] Ratcliff, R.A. (2006). *Delusions of Intelligence: Enigma, Ultra, and the End of Secure Ciphers*. Cambridge University Press.

[15] Rusbridger, J. & Nave, E. (1991). *Betrayal at Pearl Harbor: How Churchill Lured Roosevelt into World War II.* New York, NY: Summit Books.

[16] Singh, S. (1999). *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography.* New York, NY: Anchor Books.

[17] Stamp, M. & Chan, W.O. (2007). SIGABA:Cryptanalysis of the Full Keyspace. *Cryptologia*, 31(3), 201-222.

[18] Stamp, M. & Low, R.M. (2007). *Applied Cryptanalysis: Breaking Ciphers in the Real World.* Hoboken, NJ: John Wiley & Sons, Inc..

[19] Sullivan, G. & Weierud, F. (May 2006). Hillclimbing the Enigma Machine. Retrieved: 10 October 2012, from Website: `http://cryptocellar.web.cern.ch/cryptocellar/bgac/HillClimbEnigma.pdf`

[20] Twain, M. (n.d.). *Adventures of Huckleberry Finn.* Retrieved from `http://www.gutenberg.org/cache/epub/76/pg76.txt`

[21] Wescombe, P. (pjrewescombe@gmail.com), "Re: Typex Machine," E-mail to K. Chang (kelly.p.chang@gmail.com) 24 September 2011.

APPENDIX A

Swap Algorithm Test Cases

To determine which swapping algorithm worked best, slow hill-climb or fast swap, we first test it out on a single rotor simulator. A single rotor simulator works as a simple substitution cipher that steps with every character inputted. We chose to use a single rotor to perform the initial test to remove the complications of the stators and other stepping rotors found in the Typex. By doing so, we can easily pick the swap method that performs best in a single rotor situation and then add it to the Typex simulator. If we add the swap algorithms into the simulator without first testing them in a simpler setting we waste more time coding and running the algorithm that does not work in the complicated setting. Once we select our swap method we can then go ahead and implement it within Typex and then compare minor variations of the swap method. We generate 1000 random initial wirings and then run them through both the slow hill-climb and the fast swap algorithms. It is obvious from Figure A.12 that given a simple single rotor construction the fast swap method outperforms the slow hill-climb by a wide margin. The fast swap method recovers the original wiring with nearly a 100% success rate; whereas the slow hill-climb fails to come close to finding the original wiring.

To test the full effectiveness of the fast swap method variations, we add the swap methods to the Typex simulator. We use 3000 random initial wirings and 2000 character data set to solve for the one unknown rotor. Again, we base our choices on the amount of time it takes to recover a wiring.

Figure A.13 shows that the reset method without the wire comparison performs

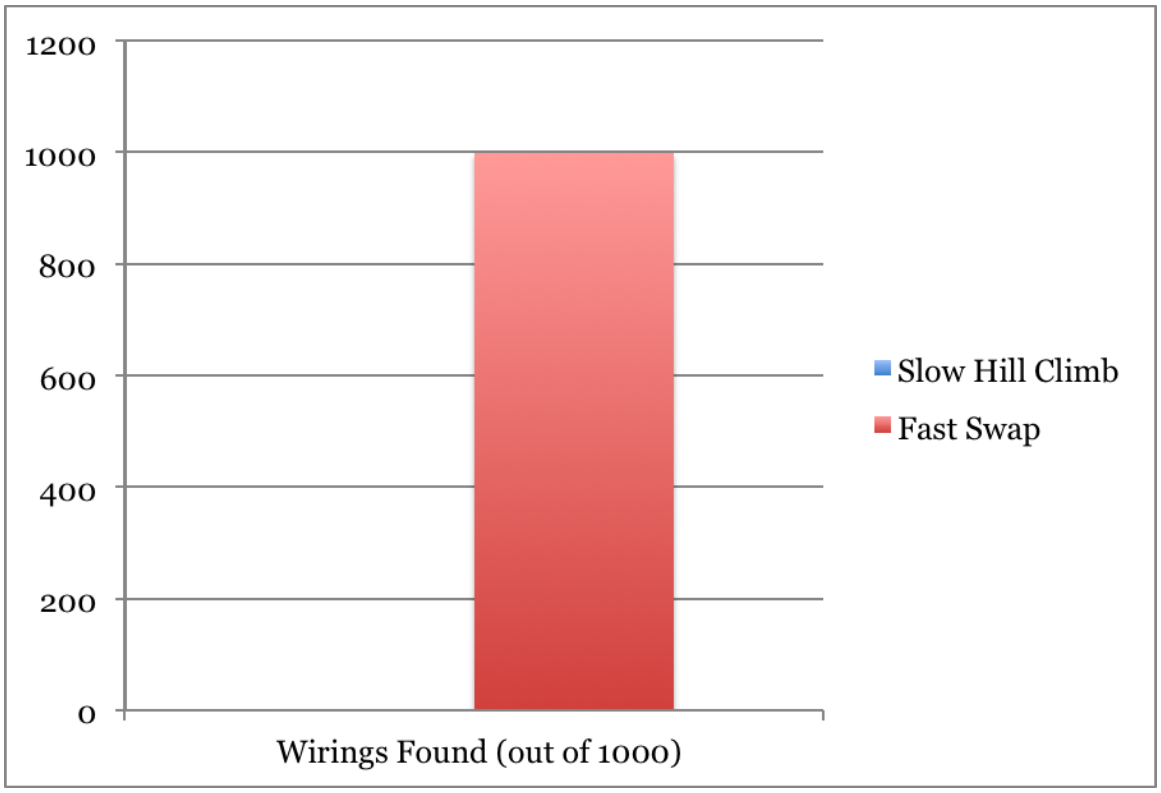


Figure A.12: Slow Hill Climb vs. Fast Swap (Single Rotor)

just as well as the reset method with the comparison. Interestingly enough, when we remove the reset and run the fast swap method with the wiring comparison, the resulting swap algorithm performs slightly better than the fast swap with reset. The wire comparison recovered the correct wiring 34 times, and the reset method only recovered the correct wiring 25 times.

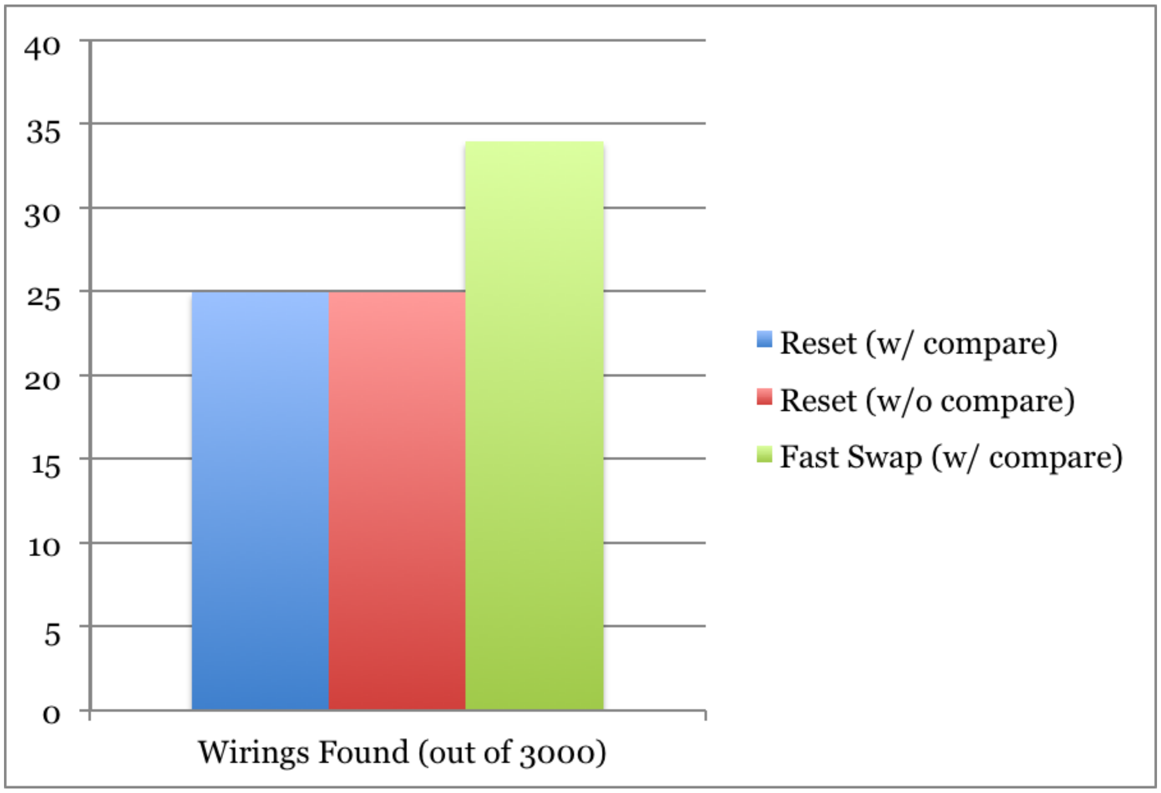


Figure A.13: Slow Hill Climb vs. Fast Swap (Single Rotor)

APPENDIX B

Nested Loop Test Cases

To verify that the choice of rotors and the order in which they are nested does not matter we run a few short test cases. Because of time constraints we only perform the tests on 100 initial wirings for each combination of two rotors. The figures below show their results.

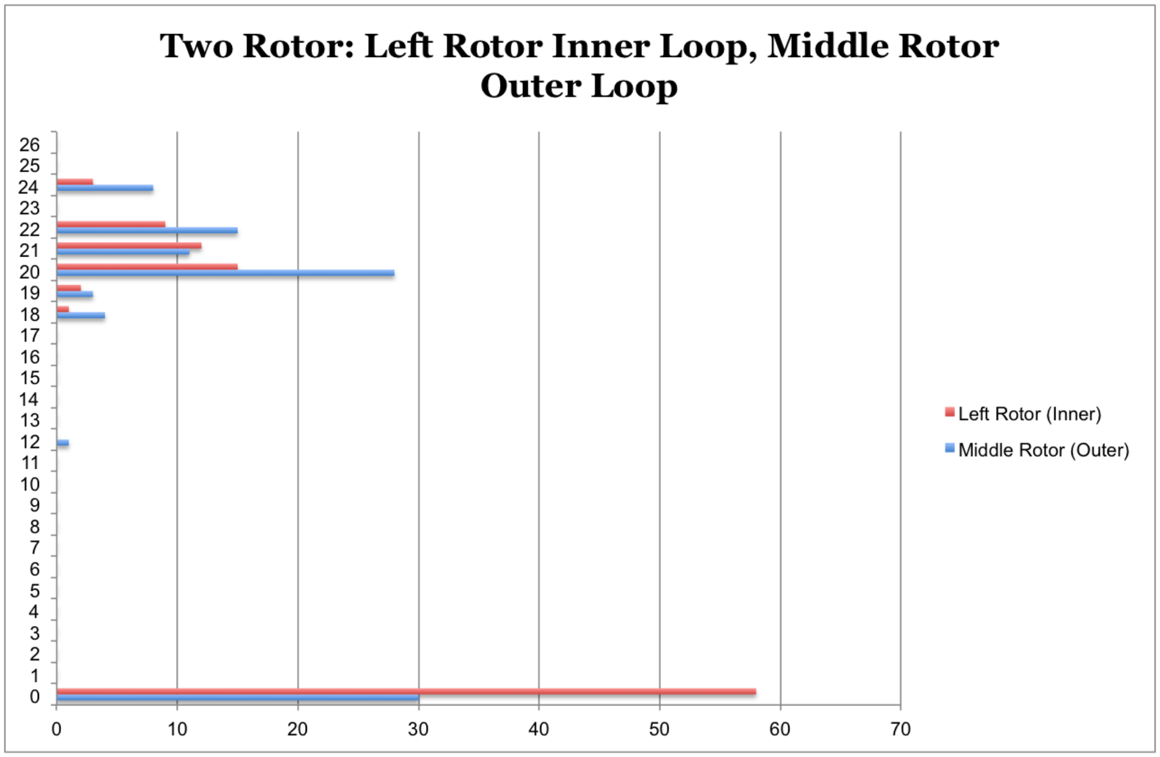


Figure B.14: Two Rotor Results, Left Rotor Inner Loop and Middle Rotor Outer Loop

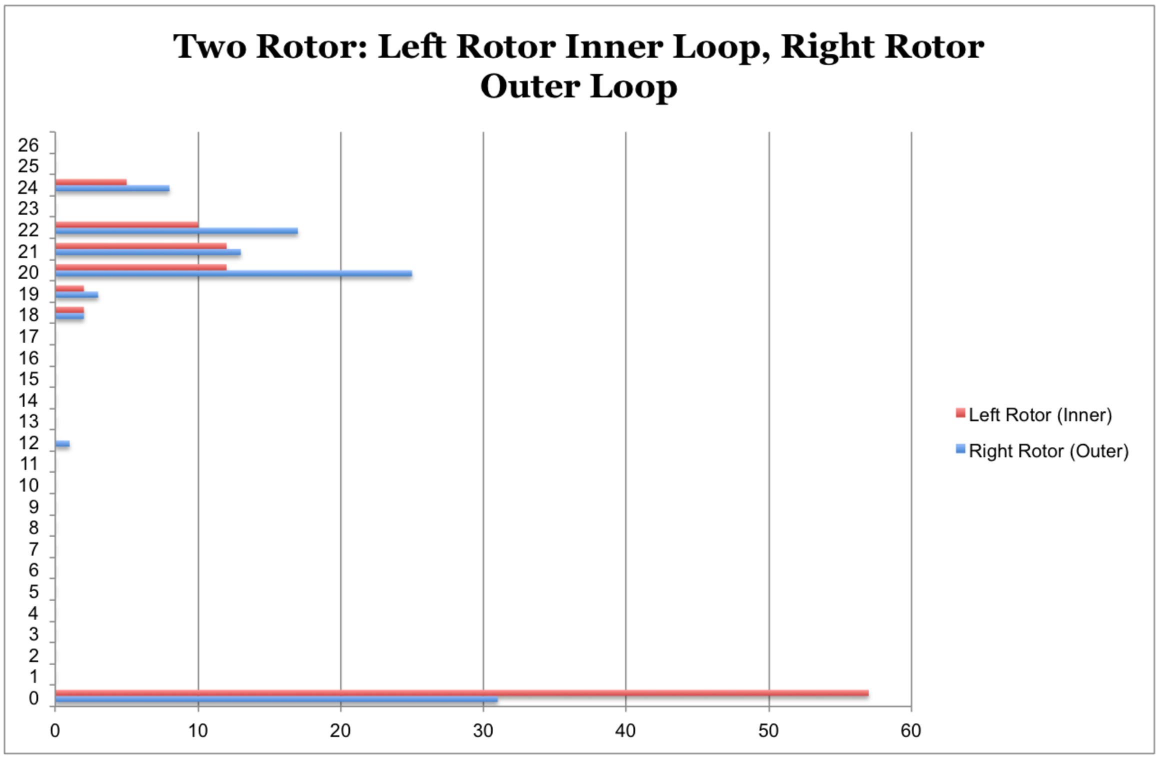


Figure B.15: Two Rotor Results, Left Rotor Inner Loop and Right Rotor Outer Loop

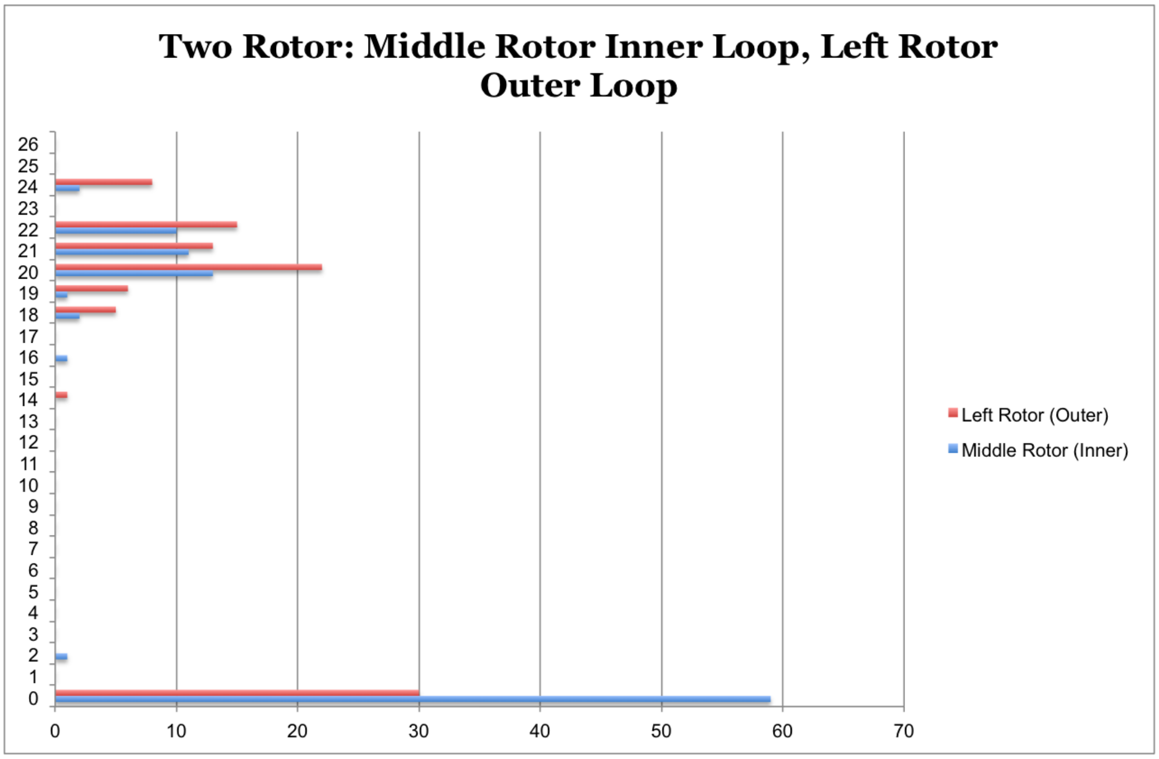


Figure B.16: Two Rotor Results, Middle Rotor Inner Loop and Left Rotor Outer Loop

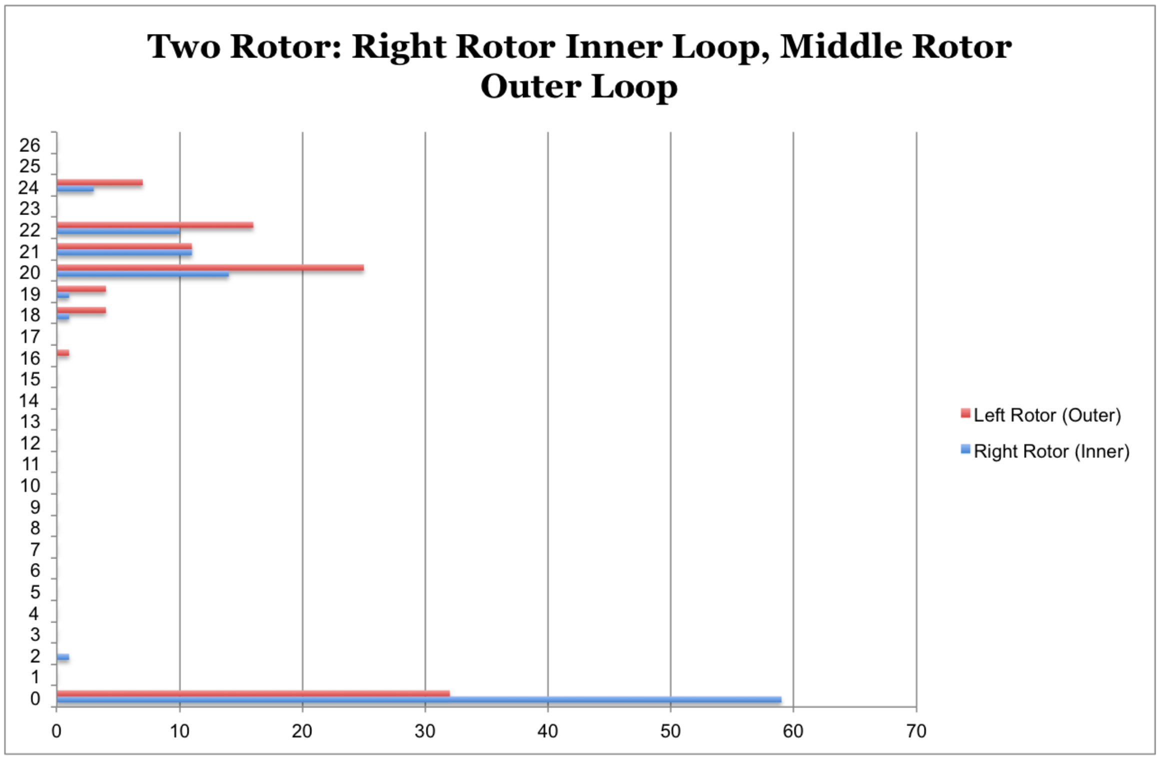


Figure B.17: Two Rotor Results, Right Rotor Inner Loop and Left Rotor Outer Loop

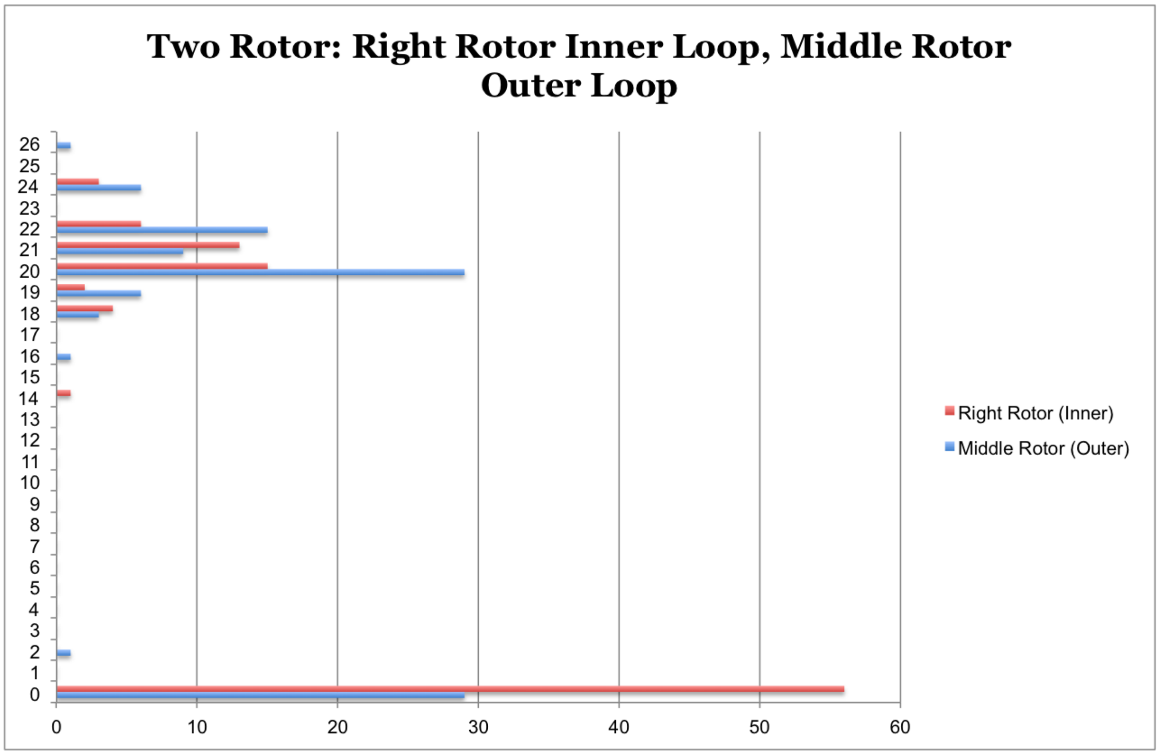


Figure B.18: Two Rotor Results, Right Rotor Inner Loop and Middle Rotor Outer Loop

APPENDIX C

Typex Simulator Code

```
/*
  Program to simulate Typex cipher
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>


// Print flags
// print rotors, reflector, and stator permutations
//#define PR_PERMS
// print output for each step of encryption/decryption
//#define PR_STEPS
// print the key
//#define PR_KEY
// print rotor stepping
//#define PR_ROTORS


FILE *in,
     *out;

int Lrotor[26],
    Mrotor[26],
    Rrotor[26],
    Lstator[26],
    Rstator[26],
    reflector[26],
    notch[9],
    ort[5];

int L[26][26],
    M[26][26],
    R[26][26],
    L_inv[26][26],
    M_inv[26][26],
    R_inv[26][26],
    LS[26][26],
    RS[26][26],
    LS_inv[26][26],
    RS_inv[26][26];

char letter[26] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
     rot[8][26] = {"QWECYJIBFKMLTVZPOHUDGNRSXA",
                   "AJDKSIRUXBLHWTMCQGZNPYFVOE",
                   "BDFHJLCPRTXVZNYEIWGAKMUSQO",
                   "ESOVPZJAYQUIRHXLNFTGKDCMWB",
                   "VZBRGITYUPSDNHLXAWMJQOFECK",
                   "FVPJIAOYEDRZXWGCTKUQSBNMHL",
                   "KZGLIUCJEHADXRYWVTNSFQPMOB",
                   "ZLVGOIFTYWUEPMABNCXRQSDKHJ"},
     ref[26] = "YRUHQSLDPXNGOKMIEBFZCWVJAT",
     step[9] = "ACEINQTVY";


int stepRotor (int pos, int n, int rev) {
    int t;
    if (rev == 1) {
        t = pos - 1;
        if (t < 0) {
            t = 25;
        }
    }
    else {
        t = pos + 1;
        if (t >= n) {
            t = 0;
        }
    }
    return(t);

}// end stepRotor

void reverse (int x) {
    int i,
        j,
        **newrotor;
```

```
newrotor = (int**) calloc(26, sizeof(int*));
for (i = 0; i < 26; ++i) {
    newrotor[i] = (int*) calloc(26, sizeof(int));
}

// reverse Left rotor
if (x == 0) {
    for (i = 0; i < 26; ++i) {
        for (j = 0; j < 26; ++j) {
            newrotor[i][j] = L[i][(26 - j) % 26];
        }
    }
    for (i = 0; i < 26; ++i) {
        for (j = 0; j < 26; ++j) {
            L[i][j] = newrotor[i][j];
        }
    }
}
// reverse Middle rotor
if (x == 1) {
    for (i = 0; i < 26; ++i) {
        for (j = 0; j < 26; ++j) {
            newrotor[i][j] = M[i][(26 - j) % 26];
        }
    }
    for (i = 0; i < 26; ++i) {
        for (j = 0; j < 26; ++j) {
            M[i][j] = newrotor[i][j];
        }
    }
}
// reverse Right rotor
if (x == 2) {
    for (i = 0; i < 26; ++i) {
        for (j = 0; j < 26; ++j) {
            newrotor[i][j] = R[i][(26 - j) % 26];
        }
    }
    for (i = 0; i < 26; ++i) {
        for (j = 0; j < 26; ++j) {
            R[i][j] = newrotor[i][j];
        }
```

```
            }
        }
        // reverse Left Stator
        if (x == 3) {
            for (i = 0; i < 26; ++i) {
                for (j = 0; j < 26; ++j) {
                    newrotor[i][j] = LS[i][(26 - j) % 26];
                }
            }
            for (i = 0; i < 26; ++i) {
                for (j = 0; j < 26; ++j) {
                    LS[i][j] = newrotor[i][j];
                }
            }
        }
        // reverse Right Stator
        if (x == 4) {
            for (i = 0; i < 26; ++i) {
                for (j = 0; j < 26; ++j) {
                    newrotor[i][j] = RS[i][(26 - j) % 26];
                }
            }
            for (i = 0; i < 26; ++i) {
                for (j = 0; j < 26; ++j) {
                    RS[i][j] = newrotor[i][j];
                }
            }
        }

        for (i = 0; i < 26; ++i) {
            free(newrotor[i]);
        }
        free(newrotor);

    }// end reverse

    void getInversePerm (int invPerm[], int perm[], int n) {
        int i;

        for (i = 0; i < n; ++i) {
            invPerm[perm[i]] = i;
        }// next i
```

```
}// end getInversePerm

void initRotors (int numL, int numM, int numR, int numLS, int numRS) {
    int i,
        j;

    // initialize rotor and reflector arrays
    for (i = 0; i < 26; ++i) {
        Lrotor[i] = (int)rot[numL][i] - 65;
        Mrotor[i] = (int)rot[numM][i] - 65;
        Rrotor[i] = (int)rot[numR][i] - 65;
        Lstator[i] = (int)rot[numLS][i] - 65;
        Rstator[i] = (int)rot[numRS][i] - 65;
        reflector[i] = (int)ref[i] - 65;

    }// next i
    // sets initial permutation
    for (i = 0; i < 26; ++i) {
        L[0][i] = Lrotor[i];
        M[0][i] = Mrotor[i];
        R[0][i] = Rrotor[i];
        LS[0][i] = Lstator[i];
        RS[0][i] = Rstator[i];
    }// next i
    // sets permutation for all other letters
    for (i = 1; i < 26; ++i) {
        for (j = 0; j < 26; ++j) {
            L[i][j] = (Lrotor[(i + j) % 26] + 26 - i) % 26;
            M[i][j] = (Mrotor[(i + j) % 26] + 26 - i) % 26;
            R[i][j] = (Rrotor[(i + j) % 26] + 26 - i) % 26;
            LS[i][j] = (Lstator[(i + j) % 26] + 26 - i) % 26;
            RS[i][j] = (Rstator[(i + j) % 26] + 26 - i) % 26;
        }// next j
    }// next i
    // reverse rotors
    for (i = 0; i < 5; ++i) {
        if (ort[i] == 1) {
            reverse(i);
        }
    }
    // find inverse permutation
    for (i = 0; i < 26; ++i) {
        getInversePerm(L_inv[i], L[i], 26);
        getInversePerm(M_inv[i], M[i], 26);
        getInversePerm(R_inv[i], R[i], 26);
        getInversePerm(LS_inv[i], LS[i], 26);
        getInversePerm(RS_inv[i], RS[i], 26);
    }// next i

    // initialize notches
    for (i = 0; i < 9; ++i) {
        notch[i] = (int)step[i] - 65;
    }
}// end initRotors


void simulator (int init_L, int init_M, int init_R, int init_LS,
                int init_RS) {
    int i,
        space,
        temp,
        cur_L,
        cur_M,
        cur_R,
        stepR,
        stepM,
        stepL;

    unsigned char inChar,
                  outChar;

    cur_L = init_L;
    cur_M = init_M;
    cur_R = init_R;
    stepR = 0;
    stepM = 0;
    stepL = 0;

    // encryption/decryption and stepping part
    while (1) {
        temp = fgetc(in);
        if (temp == EOF) {
```

```
        break;
    }
    space = temp;
    temp -= 65;
    space -= 32;
    if (space == 0) {
        temp = 23;
    }
    if (temp < 0 || temp > 25) {
        fprintf(stderr, "\nError --- all input characters must be " +
            "upper case A thru Z or a space\n");
        exit(0);
    }

    inChar = (unsigned char)temp;

    // R is fast rotor
    // M is medium rotor
    // L is slow rotor
    // rotors step _before_ encryption/decryption
    // check all possible notches
    for (i = 0; i < 9; ++i) {
        // all 3 step (step left and middle here)
        if (cur_M == notch[i]) {
            if (ort[0] == 1) {
                cur_L = stepRotor(cur_L, 26, 1);
            }
            else {
                cur_L = stepRotor(cur_L, 26, 0);
            }
            if (ort[1] == 1) {
                cur_M = stepRotor(cur_M, 26, 1);
            }
            else {
                cur_M = stepRotor(cur_M, 26, 0);
            }
            stepL++;
            stepM++;
        }
        else {
            // M and R both step (step middle here)
            if (cur_R == notch[i]) {
                if (ort[1] == 1) {
                    cur_M = stepRotor(cur_M, 26, 1);
                }
                else {
                    cur_M = stepRotor(cur_M, 26, 0);
                }
                stepM++;
            }
        }
    }
    // step right (fast) rotor --- always steps
    if (ort[2] == 1) {
        cur_R = stepRotor(cur_R, 26, 1);
    }
    else {
        cur_R = stepRotor(cur_R, 26, 0);
    }
    stepR++;

    // Typex transformation
#ifdef PR_ROTORS
    printf("[%d]", cur_R);
    printf("[%d]", cur_M);
    printf("[%d]\n", cur_L);
#endif

#ifdef PR_STEPS
    printf("\ninChar = %c\n", letter[inChar]);
    printf("RS[%d][%d] = ", init_RS, inChar);
    temp = RS[init_RS][inChar];
    printf("%c\n", letter[temp]);
    printf("LS[%d][%d] = ", init_LS, temp);
    temp = LS[init_LS][temp];
    printf("%c\n", letter[temp]);
    printf("R[%d][%d] = ", cur_R, temp);
    temp = R[cur_R][temp];
    printf("%c\n", letter[temp]);
    printf("M[%d][%d] = ", cur_M, temp);
    temp = M[cur_M][temp];
    printf("%c\n", letter[temp]);
    printf("L[%d][%d] = ", cur_L, temp);
    temp = L[cur_L][temp];
```

```
        printf("%c\n", letter[temp]);
        printf("reflector[%d] = ", temp);
        temp = reflector[temp];
        printf("%c\n", letter[temp]);
        printf("L_inv[%d][%d] = ", cur_L, temp);
        temp = L_inv[cur_L][temp];
        printf("%c\n", letter[temp]);
        printf("M_inv[%d][%d] = ", cur_M, temp);
        temp = M_inv[cur_M][temp];
        printf("%c\n", letter[temp]);
        printf("R_inv[%d][%d] = ", cur_R, temp);
        temp = R_inv[cur_R][temp];
        printf("%c\n", letter[temp]);
        printf("LS_inv[%d][%d] = ", init_LS, temp);
        temp = LS_inv[init_LS][temp];
        printf("%c\n", letter[temp]);
        printf("RS_inv[%d][%d] = ", init_RS, temp);
        temp = RS_inv[init_RS][temp];
        printf("%c\n", letter[temp]);
#endif

        outChar = RS_inv[init_RS][LS_inv[init_LS][R_inv[cur_R][M_inv[cur_M]
                     [L_inv[cur_L][reflector[L[cur_L][M[cur_M][R[cur_R]
                     [LS[init_LS][RS[init_RS][inChar]]]]]]]]]]];

#ifdef PR_STEPS
        printf("letter[temp] = %c, letter[outChar] = %c\n", letter[temp],
            letter[outChar]);
#endif

        printf("%c", letter[outChar]);
        fprintf(out, "%c", letter[outChar]);
    }// end while
    printf("\n\n");

#ifdef PR_ROTORS
    printf("stepR = %d\n", stepR);
    printf("stepM = %d\n", stepM);
    printf("stepL = %d\n", stepL);
#endif
}// end simulator
```

```
int main (int argc, const char *argv[]) {
    int i,
        j,
        n,
        init_L,
        init_M,
        init_R,
        init_LS,
        init_RS,
        numL,
        numM,
        numR,
        numLS,
        numRS;

    char infname[100],
         outfname[100];

    if (argc != 6) {
oops:   fprintf(stderr, "\n\nUsage: %s rotors orientation init infile "+
                "outfile\n\n"
                ,argv[0]);
        fprintf(stderr, "where rotors == [L][M][R][LS][RS] rotors\n"+
            "            (0 thru 7, no space, no repeats)\n");
        fprintf(stderr, "      orientation == rotor orientations, "+
            "0 = forward, 1 = reverse\n");
        fprintf(stderr,"           (binary 5-tuple)\n");
        fprintf(stderr, "      init == initial position for "+
            "[L][M][R][LS][RS] rotors\n");
        fprintf(stderr,"           (A thru Z, no space)\n");
        fprintf(stderr, "      infile == input file name\n\n");
        fprintf(stderr, "      outfile == output file name\n\n");
        fprintf(stderr,
            "For example: %s 27016 01100 ZXWAK plain.txt out.txt\n\n",
            argv[0]);
        fprintf(stderr, "Note 1: Input file must contain only upper "+
            "case A thru Z. Spaces are OK.\n\n");
        exit(0);
    }

    if (strlen(argv[1]) != 5) {
        fprintf(stderr, "\nError --- must specify 5 rotors\n");
        goto oops;
    }
    numL = argv[1][0] - 48;
    if (numL < 0 || numL > 7) {
        fprintf(stderr, "\nError --- left rotor must be 0 thru 7\n");
        goto oops;
    }
    numM = argv[1][1] - 48;
    if (numM < 0 || numM > 7) {
        fprintf(stderr, "\nError --- middle rotor must be 0 thru 7\n");
        goto oops;
    }
    numR = argv[1][2] - 48;
    if (numR < 0 || numR > 7) {
        fprintf(stderr, "\nError --- right rotor must be 0 thru 7\n");
        goto oops;
    }
    numLS = argv[1][3] - 48;
    if (numLS < 0 || numLS > 7) {
        fprintf(stderr, "\nError --- left stator must be 0 thru 7\n");
        goto oops;
    }
    numRS = argv[1][4] - 48;
    if (numRS < 0 || numRS > 7) {
        fprintf(stderr, "\nError --- right stator must be 0 thru 7\n");
        goto oops;
    }
    if ((numL == numM) || (numL == numR) || (numL == numLS) ||
    (numL == numRS) || (numM == numR) || (numM == numLS) ||
    (numM == numRS) || (numR == numLS) ||(numR == numRS) ||
    (numLS == numRS))  {
        fprintf(stderr, "\nError --- rotors must be distinct\n");
        goto oops;
    }

    // orientation check
    if (strlen(argv[2]) != 5) {
        fprintf(stderr, "\nError --- rotor orientation must be "+
            "5 characters, each 0 or 1\n");
        goto oops;
    }
```

```
for (i = 0; i < 5; ++i) {
    ort[i] = argv[2][i] - 48;
    if (ort[i] < 0 || ort[i] > 1) {
        fprintf(stderr, "\nError --- rotor orientation must be 5 "+
            "characters, each 0 or 1\n");
        goto oops;
    }
}

// init input check
if (strlen(argv[3]) != 5) {
    fprintf(stderr, "\nError --- init must be five characters, "+
        "each upper case A thru Z\n");
    goto oops;
}
init_L = argv[3][0] - 65;
if (init_L < 0 || init_L > 25) {
    fprintf(stderr, "\nError --- init must be five characters, "+
        "each upper case A thru Z\n");
    goto oops;
}
init_M = argv[3][1] - 65;
if (init_M < 0 || init_M > 25) {
    fprintf(stderr, "\nError --- init must be five characters, "+
        "each upper case A thru Z\n");
    goto oops;
}
init_R = argv[3][2] - 65;
if (init_R < 0 || init_R > 25) {
    fprintf(stderr, "\nError --- init must be five characters, "+
        "each upper case A thru Z\n");
    goto oops;
}
init_LS = argv[3][3] - 65;
if (init_LS < 0 || init_LS > 25) {
    fprintf(stderr, "\nError --- init_s must be a five characters, "+
        "each upper case A thru Z\n");
    goto oops;
}
init_RS = argv[3][4] - 65;
if (init_RS < 0 || init_RS > 25) {
    fprintf(stderr, "\nError --- init must be a five characters, "+
                "each upper case A thru Z\n");
            goto oops;
        }

        // filename input check
        sprintf(infname, argv[4]);
        in = fopen(infname, "r");
        if (in == NULL) {
            fprintf(stderr, "\n\nError opening file %s\nTry again\n\n",
                infname);
            goto oops;
        }

        // filename output check
        sprintf(outfname, argv[5]);
        out = fopen(outfname, "w");
        if (out == NULL) {
            fprintf(stderr, "\n\nError opening file %s\nTry again\n\n",
                outfname);
            goto oops;
        }

        initRotors(numL, numM, numR, numLS, numRS);

    #ifdef PR_PERMS
        printf("L perms\n");
        for (i = 0; i < 26; ++i) {
            printf("L[%2d] = ", i);
            for (j = 0; j < 26; ++j) {
                printf("%c", letter[L[i][j]]);
            }
            printf("\n");
        }
        printf("\n");
        printf("M perms\n");
        for (i = 0; i < 26; ++i) {
            printf("M[%2d] = ", i);
            for (j = 0; j < 26; ++j) {
                printf("%c", letter[M[i][j]]);
            }
            printf("\n");
        }
```

```
printf("\n");
printf("R perms\n");
for (i = 0; i < 26; ++i) {
    printf("R[%2d] = ", i);
    for (j = 0; j < 26; ++j) {
        printf("%c", letter[R[i][j]]);
    }
    printf("\n");
}
printf("\n");
printf("LS perms\n");
for (i = 0; i < 26; ++i) {
    printf("LS[%2d] = ", i);
    for (j = 0; j < 26; ++j) {
        printf("%c", letter[LS[i][j]]);
    }
    printf("\n");
}
printf("\n");
printf("RS perms\n");
for (i = 0; i < 26; ++i) {
    printf("RS[%2d] = ", i);
    for (j = 0; j < 26; ++j) {
        printf("%c", letter[RS[i][j]]);
    }
    printf("\n");
}
printf("\n");
printf("L_inv perms\n");
for (i = 0; i < 26; ++i) {
    printf("L_inv[%2d] = ", i);
    for (j = 0; j < 26; ++j) {
        printf("%c", letter[L_inv[i][j]]);
    }
    printf("\n");
}
printf("\n");
printf("M_inv perms\n");
for (i = 0; i < 26; ++i) {
    printf("M_inv[%2d] = ", i);
    for (j = 0; j < 26; ++j) {
        printf("%c", letter[M_inv[i][j]]);
        }
        printf("\n");
    }
    printf("\n");
    printf("R_inv perms\n");
    for (i = 0; i < 26; ++i) {
        printf("R_inv[%2d] = ", i);
        for (j = 0; j < 26; ++j) {
            printf("%c", letter[R_inv[i][j]]);
        }
        printf("\n");
    }
    printf("\n");
    printf("LS_inv perms\n");
    for (i = 0; i < 26; ++i) {
        printf("LS_inv[%2d] = ", i);
        for (j = 0; j < 26; ++j) {
            printf("%c", letter[LS_inv[i][j]]);
        }
        printf("\n");
    }
    printf("\n");
    printf("RS_inv perms\n");
    for (i = 0; i < 26; ++i) {
        printf("RS_inv[%2d] = ", i);
        for (j = 0; j < 26; ++j) {
            printf("%c", letter[RS_inv[i][j]]);
        }
        printf("\n");
    }
    printf("\n");
    printf("reflector\n");
    for (i = 0; i < 26; ++i) {
        printf("%c", letter[reflector[i]]);
    }
    printf("\n\n");
#endif

#ifdef PR_KEY
    printf("\nKey:\n");
    printf("rotors (L,M,R,LS,RS) = (%d,%d,%d,%d,%d)\n", numL, numM, numR,
             numLS, numRS);
    printf("initial rotor positions (L,M,R,LS,RS) = (%c,%c,%c,%c,%c)\n",
        (char)(init_L + 65), (char)(init_M + 65), (char)(init_R + 65),
        (char)(init_LS + 65), (char)(init_RS + 65));
    printf("\nOutput:\n");
#endif

    simulator(init_L, init_M, init_R, init_LS, init_RS);

    fclose(in);
    fclose(out);

}// end main
```