RedDye Cipher by Karl Zander (pvial00)

Analysis by Randomiser

INITIAL OBSERVATIONS

I first want to mention something that struck me as odd about the python implementation of the cipher; that being the fact that the counter for the key index is not separate from the mod 26 counter that is added to j. When 26 letters have been encrypted it often causes the index to reset to 0 before the end of the key has been reached, which also means no key can be any longer than 27 characters. I was not sure if this was intentional or not as I saw no mention of it in your YouTube video, which used separate counters. For this demonstration I used a key that was exactly 26 letters long, as the basic idea remains the same whether a single counter is used or not. The resetting effect isn't witnessed at other key lengths that divide evenly into 26, and it doesn't affect the binary version of the cipher at all since the key lengths should always go into 256 bits.

I should also mention that the addition of c being added to j doesn't actually do anything at all to make the cipher more secure. Its effects on the keystream are identical for every ciphertext and thus can be computed ahead of time. If an attacker wanted to remove the variable c they could decrypt the ciphertext with such a keystream consisting of c's iterative effect and they could analyze the resulting text, which leaves only the effects of k and j. c may seem like a psuedorandom number generator, but it's a completely static and therefore useless one.

My experiments and demonstration have used the H4 hand cipher, but from my impression the binary version is similar enough that the same type of attack can be used against it as well.

SUSCEPTIBILITY TO KNOWN PLAINTEXT ATTACKS

While the keystream itself is not periodic, the key has periodic properties. While the key being modified makes it less prone to analysis than a simple periodic vigenere cipher, the fact that a key letter is only modified based on the next letter in the key and j makes it possible to determine other parts of the keystream if part of the key is known.

The formula to generate the keystream from the key is far from irreversible, and with some amount of known plaintext and a reasonably small amount of brute force, part or all of the key can be recovered. I've made the following assessments and back them up with demonstrations further below:

- **If a plaintext section the length of the initial key is known, it is easily possible to reverse the keystream into the key and decrypt the entire message.**
- If the length of known plaintext p is shorter than the key length k, by recovering the partial key it is possible to recover an additional p-1 plaintext characters k characters further into the cipher after only 13 additional guesses.

METHOD

To reverse the math to get the key from the keystream more easily I used a simplified version of the keystream generator in my implementation, and didn't alter k[c] or j until the output keystream character was already generated. Since the keystream output consists mostly of the same values added together multiple times, I rearranged it like this:

```
keystream.append((3*(k[c]) + 3*(k[c+1]) + 4*j + c)%26)
k.append((k[c] + k[c+1] + j) % 26)
j = (j + k[c] + k[c+1] + j + c) % 26
```

(I also appended the modified key values to the end of the current key instead of writing over the original key - this isn't strictly necessary, I just wanted to see how the key changed over time while experimenting.)

Now, if I have a known plaintext character and know or guess the k[c] and j that corresponds to it, I can easily solve for the k[c+i], then use that as the next k[c] to solve for third character in the key and so on for each known pt character.

The following lines take a known ciphertext and plaintext pair, and returns the next letter in the key:

```
next = alphabet.index(letter) - 3*(k[c]) - 4*j - alphabet.index(known[c]) - c
while (next < 26) or (next % 3 != 0): next += 26
next = (next/3) % 26
```

What I do need to bruteforce for this to work is the first letter in the key and the initial value of j. There are 26 values for the first letter, but only 13 values for j; as due to the number of times it gets added to itself and the key encrypting a message with j at 0 through 12 is the same as encrypting with j at 13 through 25 (as 4*13 % 26 = 0). So at most I have to guess 26*13 combinations in order to return the correct full key. For the binary version of the cipher this would be 256*64 (as 4*64 % 256 = 0), which is still quite small for a stream cipher given modern computing.

I've attached a python demonstration of this method when enough plaintext is known to recover the entire key. You can read more about that in the appendix.

But what about situations where we're unable to find the entire key? It's somewhat more complicated, but still possible to glean information on other parts of plaintext. I'll show the following example:

"STRIKETHECASTLEATMIDNIGHT" encoded with the key "mypassword" gives the ciphertext "WVSPCXXDQLCVHIPOFAGZKAXAL." Suppose as the attacker, all we know is that the first word is "strike." If we guess the key is 10 letters long we can still decode other sections of the message.

We can begin by breaking the ciphertext into 10-letter chunks, so that each section starts where the key index resets.

??????????	??????????	?????
WVSPCXXDQL	CVHIPOFAGZ	KAXAL
STRIKE????	??????????	?????

For the first step, much remains the same. We bruteforce the starting letter and j to find possible keys that fit the known plaintext "strike."

c=0, j=0

MYPASSW???	??????????	?????
WVSPCXXDQL	CVHIPOFAGZ	KAXAL
STRIKE????	??????????	?????

new key : XXXZTU????

Eventually, we'll come to firstletter = M and j = 0. With this pair of values the rest of the known key is "MYPASSW", and by following the keystream formula, these letters will become "XXXZTU" for when the key index resets. If we skip ahead to where the key index repeats itself, we can decode another piece of of plaintext. We set "XXXZTU" as the password to decrypt, and now bruteforce j, which once again only has 13 possible values. We no longer need to bruteforce k[c]; we know it's 'X' because we've already computed it when doing our math from the first known plaintext values. We also plug in 10 for c since it's the 11th letter. Thus the possible plaintexts for each j value are:

```
0     KMHNI
1     GERHW
2     CWBBK
3     YOLVY
4     UGVPM
5     QYFJA
6     MQPDO
7     IIZXC
8     EAJRQ
9     ASTLE
10    WKDFS
11    SCNZG
12    OUXTU
```

The best-looking result is for j=9, "ASTLE", the ending of "castle."

c=0, j=0 c=10, j=9

MYPASSW???	XXXZTU????	?????
WVSPCXXDQL	CVHIPOFAGZ	KAXAL
STRIKE????	ASTLE?????	?????

new key : DQTUW?????

Note that solving the second section also gives us the next part of the keystream to use for the third section. With a longer ciphertext, this could be repeated for the rest of the message or until we run out of known plaintext characters.

c=0, j=0	c=10, j=9	c=20, j=6
MYPASSW???	XXXZTU????	DQTUW?
WVSPCXXDQL	CVHIPOFAGZ	KAXAL
STRIKE????	ASTLE?????	NIGH?

Note that the window of recovered text does get smaller by one over time. Even though the known plaintext was 6 characters long, since we don't know how the 7th letter of the key is modified we can only recover 5 letters in the next section, and only 4 in the last.

We still suffer from a side-effect that if any text before or after the recovered sections is obvious enough to guess, we can gain more known plaintext to use once again.

APPENDIX – INFO ON PYTHON DEMONSTRATION

The included file crackdye.py is intended to demonstrate key recovery when enough plaintext is known to find the entire key. It doesn't yet work for partial key recovery, thus has the following limitations; For simplicity's sake, it assumes that the known plaintext is positioned at the beginning of the ciphertext, and that the length of the key is the same as the length of the variable "known". The ciphertext and known plaintext are plugged into the variables "ct" and "known" at the top of the file.

It also does not include the previously mentioned early key resetting effect for key lengths that don't divide evenly into 26.

By default I've supplied the ciphertext "NYUGXANVTTTMRRUPUWFOGDSFFJDQHFMITFKUEBUKPNPRYWUNZTXAQKMVWOYRRMPVRYCADRIEIWEFGSFXLRUUAPGVBDGNDPWOETQUNDR" and 26 letters of known plaintext, for which it should print the 26 letter key and the entirety of the plaintext.

The program currently only "scores" output by limiting to keys where the guessed initial j value is correct for the sum of the key letters. Since we have enough to find the entire key there is only one possible output.

If this cipher were used in the real world I would redesign the program with more variables such as password length and a proper plaintext scoring algorithm similar to the partial key recovery example I demonstrated in the previous section, without having to manually sift through results.