Software Security 2018-A
Assignment 1: Enigma

Deadline: Monday, 12/11/2018

In this assignment, we will implement an exact emulator of German Army Enigma M3 machine. In the next assignment, we will implement this machine's cryptanalysis, discover the hidden machine settings used in *Numberphile* series, and break a real message. For that task to be achievable, however, the emulation must be perfect — fortunately, there are emulators available for debugging.

You should submit a ZIP file that includes the code (in any conventional programming language, but mind the performance requirements), and a PDF with explanations for the grader. We remind you that all assignments in this course must be done alone.

References

While all the necessary details are provided in the assignment, the following references are useful for implementing and debugging an Enigma emulator:

1. Enigma Machine — general background
2. Enigma Rotor Details — we are interested in military Rotors I–V, and Reflector B
3. Visual Enigma M3 Simulator — does not support inner ring setting (it is always `A-A-A`)
4. Navy M3/M4 Enigma Machine Emulator — note that mechanics description is buggy
5. Enigma Simulator (Windows) — see also many more resources on the site
6. Numberphile's Enigma analysis — we are going to emulate this exact Enigma version

Some emulators above use letters in ring offsets — our Enigma version uses numbers. These are interchangeable: `01` corresponds to `A`, `02` to `B`, etc. Note that in order to manipulate these values internally, they need to be converted to 0–25 range for shift operations modulo 26.

Functionality

Functionally, Enigma works as follows: each letter is transformed through the plugboard, right rotor, middle rotor, left rotor, reflector, left rotor (reverse direction), middle rotor (reverse direction), right rotor (reverse direction), and plugboard again. When a key is pressed, the rotors rotate *before* the translation.

An Enigma configuration is defined by the 3 selected rotors (out of 5), their initial offsets (visible in the small windows), their internal setting (functionally similar to the offset), and plugboard configuration.

Each rotor defines a forward (right to left) and a reverse (left to right) permutation. In addition, each rotor has a turnover notch, that influences the rotor to its left during rotation — similar to an analog clock, but not exactly (more on that later).

Here are the 5 rotors we can choose from:

Rotor	**Forward permutation**	**Turnover notch**
I	`EKMFLGDQVZNTOWYHXUSPAIBRCJ`	`Q` → `R`
II	`AJDKSIRUXBLHWTMCQGZNPYFVOE`	`E` → `F`
III	`BDFHJLCPRTXVZNYEIWGAKMUSQO`	`V` → `W`
IV	`ESOVPZJAYQUIRHXLNFTGKDCMWB`	`J` → `K`
V	`VZBRGITYUPSDNHLXAWMJQOFECK`	`Z` → `A`

The reverse permutation can be easily computed from the forward one. For instance, the reverse permutation of Rotor I is `UWYGADFPVZBECKMTHXSLRINQOJ`. The forward permutation shows that letter `A` is translated to `E` (1^{st} letter in the permutation), and the 5^{th} letter in reverse permutation is therefore `A`.

There is only one reflector to use with Enigma M3:

Reflector	**Permutation**
B	`YRUHQSLDPXNGOKMIEBFZCWVJAT`

A reflector is functionally similar to a rotor, but it is static, and its permutation is symmetric — for this reason Enigma's decryption process is identical to encryption. E.g., we see that `A` is translated to `Y`, and `Y` is translated back to `A` — the reverse permutation is identical to the forward permutation.

A plugboard configuration is functionally similar to a reflector, but there can be at most 10 translation pairs. E.g., `A` ↔ `Y` above is one translation that is realized by connecting letters `A` and `Y` on the plugboard. Unconnected letters translate to themselves.

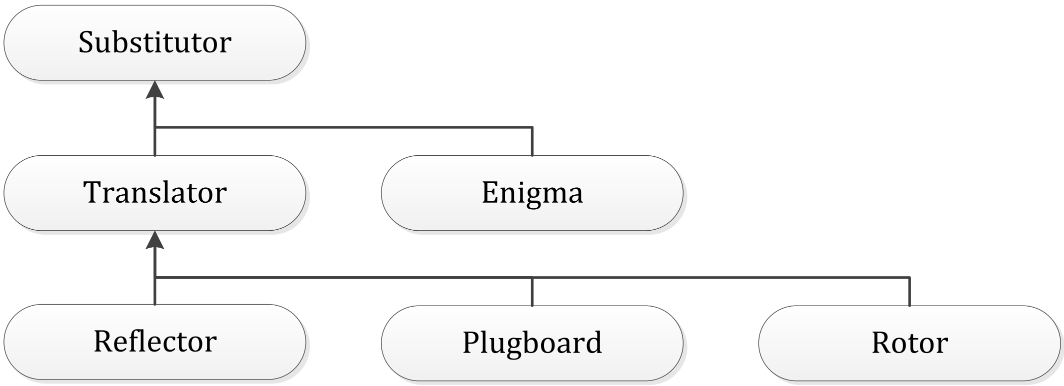
Turnover notches in the rotors table above show ring offsets (visible in small windows) that trigger further steps of rotors to the left of given rotor. However, the exact process is somewhat peculiar. When the rightmost rotor goes through turnover notch during a step, the middle rotor steps as well. The middle rotor affects the leftmost rotor in a similar fashion. However, if the middle rotor is in a turnover position (e.g., `Q` for Rotor I), it *will* step regardless of rightmost rotor offset, and trigger a step of the leftmost rotor — this is called a double step.

Task 1

Define an object-oriented design of the emulator, and summarize it in the submitted document.

One option is to have an abstract `Substitutor` class that defines helper letter-index conversions and circular shifts, an abstract letter translation method, and a default reverse translation method that maps to the regular translation method — we assume a symmetric mapping, which is true for plugboards, reflectors, and Enigma machines.

Then, `Translator` class can inherit from the class described above, and implement simple forward and reverse permutations (the reverse permutation can be precomputed from the forward one). `Reflector`, `Plugboard` and `Rotor` classes should then inherit from this class, and `Enigma` class should inherit from `Substitutor` directly, encompassing three rotors, a reflector, and a plugboard.



1. `Substitutor`: letter-index conversions, circular shifts, abstract forward and reverse translation.
2. `Translator`: simple forward and automatically computed reverse permutations.
3. `Reflector`: a `Translator` with symmetric permutation.
4. `Plugboard`: similar to `Reflector`, but converts each pairs specification into its corresponding permutation.
5. `Rotor`: letter translation via a single rotor, taking into account ring setting and offset.
6. `Enigma`: machine configuration process, complete letter translation, and rotors single and double-stepping mechanism.

Task 2

Implement plugboard (*steckerbrett*) and reflector wheel (*umkehrwalze*) functionalities. Plugboard should support partial configurations (less than 10 pairs) — this will be useful for cryptanalysis later.

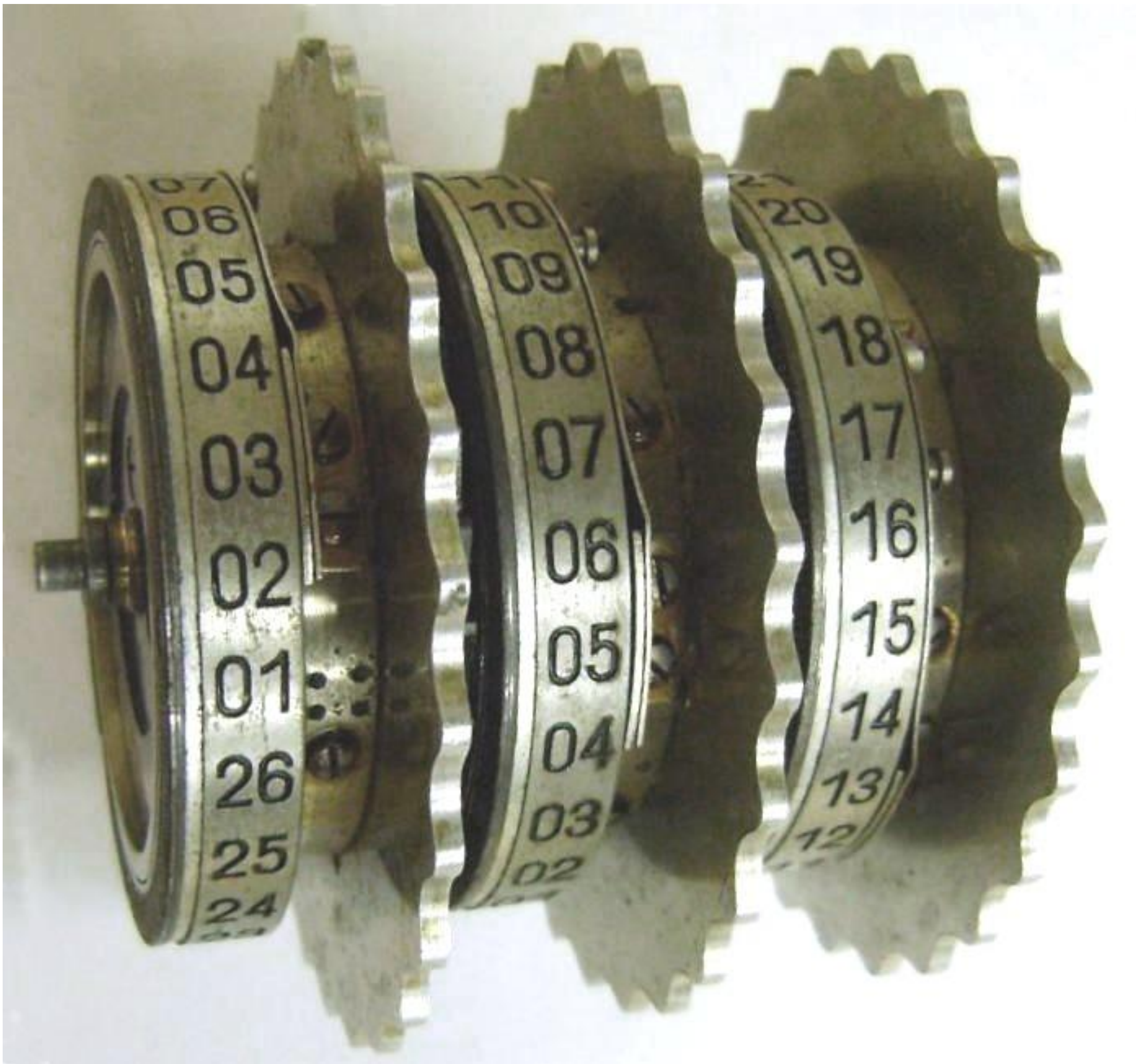
Here are some examples:

Reflector	**Input**	**Output**
B	B	R
B	E	Q
B	Q	E

Plugboard configuration	**Equivalent permutation**	**Input**	**Output**
empty	ABCDEFGHIJKLMNOPQRSTUVWXYZ	T	T
RY BU AS FZ	SUCDEZGHIJKLMNOPQYATBVWXRF	U	B
SW AQ NP FO VY UX MK CL HT ZJ	QBLDEOGTIZMCKPFNARWHXYSUVJ	U	X
SW AQ NP FO VY UX MK CL HT ZJ	QBLDEOGTIZMCKPFNARWHXYSUVJ	B	B

Task 3

Implement rotor (*walzen*) functionality. Unlike reflectors, rotors have a *state*, which is composed of ring offset, and ring setting.



Ring offset, (also ground setting — *grundstellung*) is easily set by rotating the rotors, is advanced automatically via key presses, and is visible through a small window as a letter or as a letter index, depending on the machine version.

Ring setting (*ringstellung*) is a relative shift of rotor's inner output ring, and is a more basic configuration of the Enigma machine, like picking which rotors to use.

Without going into too much mechanical details, the two configurations above are complementary to each other. For an input letter A, ring offset B, ring setting C, and rotor forward or reverse permutation P, the result is given by $P(A+B-C)-B+C$. The principal difference between the two is that ring setting does not affect the turnover notch status. Thus, if the rightmost rotor is Rotor I with offset Q, the next key press will activate the notch regardless of ring setting.

Refer to examples below when implementing rotor functionality. Note that the ring offset shown is after the rotor shifts to that offset due to, e.g., a key press. Notch status shows whether the rotor is pending a notch turnover action — you will use that status for implementing single and double stepping in rotor combinations.

Rotor	**Ring setting**	**Ring offset**	**Input**	**Direction**	**Output**	**Notch**
I	`A (01)`	`A (01)`	`E`	*forward*	`L`	*no*
I	`A (01)`	`D (04)`	`H`	*forward*	`K`	*no*
I	`A (01)`	`W (23)`	`G`	*forward*	`Q`	*no*
V	`E (05)`	`Z (26)`	`P`	*forward*	`X`	*yes*
V	`E (05)`	`Z (26)`	`X`	*reverse*	`P`	*yes*

For example, consider the 4th row. Input letter `P` is shifted 25 positions due to ring offset. The resulting letter `O` could be the actual letter to be translated by rotor's wiring — however, rotor's inner ring setting has an opposite effect: letter `O` is shifted −4 positions, and the actual translation used is `K` → `S`. We then correct the result with opposite shifts: letter `S` is shifted −25+4 positions, and the resulting output is letter `X`. In addition, ring offset `Z` is in turnover notch position — e.g., if it is the rightmost rotor, next key press will trigger middle rotor step.

Task 4

Implement the complete Enigma machine functionality. After configuring which rotors and reflector to use, the machine allows configuration of ring setting and offset for each rotor, and of plugboard letter swaps. Each key press steps at least one rotor before translation is performed. For left, middle and right rotors (L, M, R), the stepping algorithm is as follows:

```
if R.notch or M.notch
    if M.notch
        advance L.offset
    advance M.offset
advance R.offset
```

During translation, each letter passes through plugboard, R-M-L rotors, reflector, L-M-R rotors in reverse, and plugboard again. For identical Enigma machine configurations, translation should be symmetric — this allows for message decryption using the same initial configuration.

You should extensively test your Enigma machine against emulators mentioned in the beginning of the assignment, but the examples below cover various corner cases.

Reflector: B	**Plugboard**: *empty*				
Rotors	**Ring setting**	**Initial ring offsets**	**Final ring offsets**	**Input**	**Output**
I-II-III	`A-A-A (01-01-01)`	`F-D-V (06-04-22)`	`G-F-B (07-06-02)`	`ENIGMA`	`QGELID`
I-II-III	`A-A-A (01-01-01)`	`Q-E-V (17-05-22)`	`R-F-B (18-06-02)`	`KAXMNF`	`ENIGMA`
I-II-III	`A-A-A (01-01-01)`	`X-E-Y (24-05-25)`	`Y-F-E (25-06-05)`	`TURING`	`ACELKT`
I-II-IV	`C-H-F (03-08-06)`	`S-D-I (19-04-09)`	`T-F-N (20-06-14)`	`PEACE`	`ISWAR`
Reflector: B	**Plugboard**: `AT CE RL`				
I-II-IV	`C-H-F (03-08-06)`	`S-D-I (19-04-09)`	`T-F-N (20-06-14)`	`PEACE`	`IRJZU`
Reflector: B	**Plugboard**: `ZU HL CQ WM OA PY EB TR DN VI`				
II-V-IV	`S-I-X (19-09-24)`	`C-O-N (03-15-14)`	`C-O-Q (03-15-17)`	`DOR`	`MLD`
II-V-IV	`S-I-X (19-09-24)`	`C-O-N (03-15-14)`	`C-O-Q (03-15-17)`	`MLD`	`DOR`

Task 5

Consider an actual German Army code sheet for October 1944:

Geheime Kommandosache! Nicht ins Flugzeug mitnehmen

Armee-Stabs-Maschinenschlüssel Nr. 28
für Oktober 1944

Nr. 00008

	Datum	Walzenlage			Ringstellung			Steckerverbindungen										Kenngruppen			
St	31.	IV	V	I	21	15	16	KL	IT	FQ	HY	XC	NP	VZ	JB	SE	OG	jkm	ogi	ncj	glp
St	30.	IV	II	III	26	14	11	ZN	YO	QB	ER	DK	XU	GP	TV	SJ	LM	ino	udl	nam	lax
St	29.	II	V	IV	19	09	24	ZU	HL	CQ	WM	OA	PY	EB	TR	DN	VI	nci	cid	yhp	nip
St	28.	IV	III	I	03	04	22	YT	BX	CV	ZN	UD	IR	SJ	HW	GA	KQ	zqj	hlg	xky	ebt
St	27.	V	I	IV	20	06	18	KX	GJ	EP	AC	TB	HL	MW	QS	DV	OZ	bvo	sur	ccc	lqe
St	26.	IV	I	V	10	17	01	YV	GT	OQ	WN	FI	SK	LD	RP	MZ	BU	jhx	uuh	giw	ugw
St	25.	V	IV	III	13	04	17	QR	GB	HA	NM	VS	WD	YZ	OF	XK	PE	tba	pnc	ukd	nld
St	24.	III	II	IV	09	20	18	RS	NC	WK	GO	YQ	AX	EH	VJ	ZL	PF	nfi	mew	xbk	yes
St	23.	V	II	III	11	21	08	EY	DT	KF	MO	XP	HN	WG	ZL	IV	JA	lsd	nuo	vcr	vex
St	22.	I	II	IV	01	25	02	PZ	SE	DJ	XF	HA	GB	VQ	UY	KW	LR	yji	rwy	rdk	nso
St	21.	IV	I	III	06	22	03	GH	JR	TQ	KF	NZ	IL	WM	BD	UQ	EC	ema	mlv	jjy	iqh
St	20.	V	I	II	12	25	08	TF	RQ	XV	DZ	PY	NL	WI	SJ	ME	GB	xjl	pgs	ggh	znd
St	19.	IV	III	II	07	05	23	ZX	EU	AC	GD	KP	VO	QS	NW	HL	BM	vpj	zqe	jrs	cgm
St	18.	II	III	V	19	14	22	WG	OM	RL	DB	ST	AQ	PZ	XH	YN	IJ	oxd	lnb	iou	ytc
St	17.	IV	I	II	12	08	21	ME	HX	BF	WY	ZD	TR	FJ	AG	IL	KQ	tak	pjs	kdh	jvh
St	16.	I	II	III	07	11	15	WZ	AB	MO	TF	RX	SG	QU	VJ	YN	EL	pzg	avw	wyt	iye
St	15.	III	II	V	06	16	02	GT	YC	EJ	UA	RX	PN	IS	WB	MH	ZV	bhe	xzm	yzk	evp
St	14.	II	I	V	23	05	24	AZ	CJ	WF	HY	SO	QV	MI	NH	DP	GX	fdx	tyj	bmq	typ
St	13.	IV	II	V	03	25	10	CX	KN	JR	DQ	IU	TL	HZ	MF	EP	WB	zfo	bjr	zwx	gvn
St	12.	I	III	II	26	01	18	QB	YE	WN	AI	GJ	TO	HR	FK	PS	CM	upc	anf	tkr	pwz
St	11.	V	I	III	17	13	04	SV	GO	PA	ZR	FN	HI	YK	WT	DE	BJ	vdh	ego	wmy	uti
St	10.	I	V	IV	26	07	16	SW	AQ	NP	FO	VY	UX	MK	CL	HT	ZJ	rpl	anw	vpr	mhn
St	9.	I	III	IV	17	10	18	EH	IR	GK	NZ	SP	UA	LD	CQ	JM	YV	knq	ysq	rhj	tlj
St	8.	V	II	I	23	11	25	QY	OG	ST	HA	CB	WD	KL	JN	VX	IU	lrc	avw	axh	gws
St	7.	II	III	I	06	12	03	BG	FS	TH	JE	VK	PI	CU	QA	OD	NM	aty	mbb	mvo	jmz
St	6.	I	IV	V	24	19	01	IR	HQ	NT	WZ	VC	OY	GP	LF	BX	AK	bhc	iwo	zgz	rnr
St	5.	II	IV	III	05	22	14	MK	GO	RQ	XT	DW	IA	ZL	SY	PJ	ER	bok	rzw	kzo	ryl
St	4.	IV	II	I	15	02	21	KD	PG	CO	FW	HJ	RY	MT	QL	VB	UZ	kpk	php	xmo	pfw
St	3.	III	V	IV	03	23	04	DY	CP	WN	OV	QH	UZ	RA	TI	GL	SM	hjy	nkt	ytn	pvc
St	2.	I	III	V	13	18	01	DR	VJ	FS	EK	TU	HX	AQ	GT	YO	FC	gpq	fqw	oiy	ruj
St	1.	II	IV	I	06	17	26	AC	LS	BQ	WN	MY	UV	FJ	PZ	TR	OK	bol	ooi	ywv	sfb

For October 29th, the configuration consists of rotors II-V-IV, ring setting `19-09-24` (`S-I-X`), and plugboard configuration `ZU HL CQ WM OA PY EB TR DN VI`. One of the four 3-letter identification groups (*kenngruppen*) is sent with a random 2-letter prefix in cleartext in order to help identify the machine configuration in use. According to procedures that were intended to resist code breaking, the radio operator would randomly choose ring offsets (a ground setting) and a random 3-letter message key (*spruchschlussel*). He would then encode the message key, and use this message key as new ring offsets for encoding the actual message. I.e., for ground setting G, message key K, prefixed letter identification group R, and message M, the operator would transmit $G \parallel E(K, G) \parallel R \parallel E(M, K)$, where $E(M, K)$ is Enigma output for ring offsets K and input M. The decoding process is therefore $E(M, E(E(K, G), G))$.

Decrypt the following message using the emulator that you have built:

```
CON MLD
RNYHP UMDPQ CUAQN LVVSP
IARKC TTRJQ KCFPT OKRGO
ZXALD RLPUH AUZSO SZFSU
GWFNF DZCUG VEXUU LQYXO
TCYRP SYGGZ HQMAG PZDKC
KGOJM MYYDD H
```

Note that `RNYHP` is not a part of the encrypted message — `YHP` is an identification group.

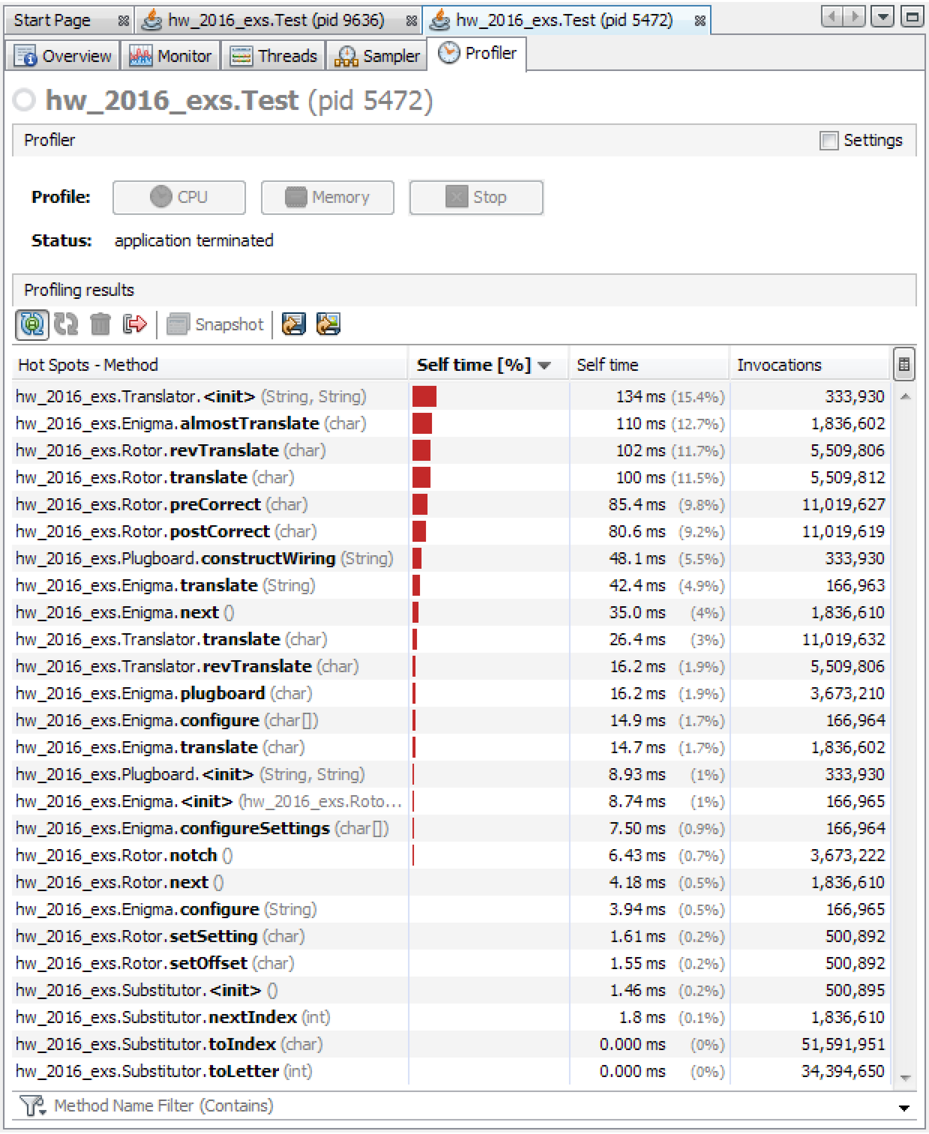
You should describe the decryption process in the submitted file.

Task 6

Since in the next assignment we will cryptanalyze Enigma, you need to make sure that your emulator is as fast as possible. E.g., there should be no routine use of string search operations — table lookups need to be used instead, and all sanity checks should be conditioned by a debugging variable that can be disabled.

Write a loop that repeatedly constructs and configures an Enigma machine (including plugboard configuration), and encodes a short message. Then, profile this loop, and eliminate performance bottlenecks. E.g., VisualVM is a nice option for Java. You need to insert a profiling summary (for instance, a method call histogram) into the submitted file.

For example:



Hot Spots - Method	Self time [%]	Self time	Invocations
hw_2016_exs.Translator.**<init>** (String, String)		134 ms (15.4%)	333,930
hw_2016_exs.Enigma.**almostTranslate** (char)		110 ms (12.7%)	1,836,602
hw_2016_exs.Rotor.**revTranslate** (char)		102 ms (11.7%)	5,509,806
hw_2016_exs.Rotor.**translate** (char)		100 ms (11.5%)	5,509,812
hw_2016_exs.Rotor.**preCorrect** (char)		85.4 ms (9.8%)	11,019,627
hw_2016_exs.Rotor.**postCorrect** (char)		80.6 ms (9.2%)	11,019,619
hw_2016_exs.Plugboard.**constructWiring** (String)		48.1 ms (5.5%)	333,930
hw_2016_exs.Enigma.**translate** (String)		42.4 ms (4.9%)	166,963
hw_2016_exs.Enigma.**next** ()		35.0 ms (4%)	1,836,610
hw_2016_exs.Translator.**translate** (char)		26.4 ms (3%)	11,019,632
hw_2016_exs.Translator.**revTranslate** (char)		16.2 ms (1.9%)	5,509,806
hw_2016_exs.Enigma.**plugboard** (char)		16.2 ms (1.9%)	3,673,210
hw_2016_exs.Enigma.**configure** (char[])		14.9 ms (1.7%)	166,964
hw_2016_exs.Enigma.**translate** (char)		14.7 ms (1.7%)	1,836,602
hw_2016_exs.Plugboard.**<init>** (String, String)		8.93 ms (1%)	333,930
hw_2016_exs.Enigma.**<init>** (hw_2016_exs.Roto...		8.74 ms (1%)	166,965
hw_2016_exs.Enigma.**configureSettings** (char[])		7.50 ms (0.9%)	166,964
hw_2016_exs.Rotor.**notch** ()		6.43 ms (0.7%)	3,673,222
hw_2016_exs.Rotor.**next** ()		4.18 ms (0.5%)	1,836,610
hw_2016_exs.Enigma.**configure** (String)		3.94 ms (0.5%)	166,965
hw_2016_exs.Rotor.**setSetting** (char)		1.61 ms (0.2%)	500,892
hw_2016_exs.Rotor.**setOffset** (char)		1.55 ms (0.2%)	500,892
hw_2016_exs.Substitutor.**<init>** ()		1.46 ms (0.2%)	500,895
hw_2016_exs.Substitutor.**nextIndex** (int)		1.8 ms (0.1%)	1,836,610
hw_2016_exs.Substitutor.**toIndex** (char)		0.000 ms (0%)	51,591,951
hw_2016_exs.Substitutor.**toLetter** (int)		0.000 ms (0%)	34,394,650