ארגון ותכנות המחשב

תרגיל 2 - חלק רטוב

המתרגל האחראי על התרגיל: בועז מואב.

שאלותיכם במייל בעניינים מנהלתיים בלבד, יופנו רק אליו.

כתבו בתיבת subject: רטוב 2 אתמ.

שאלות בעל-פה ייענו על ידי כל מתרגל.

הוראות הגשה (לקרוא!!!):

- ההגשה בזוגות.
- שאלות הנוגעות לתרגיל יש לשאול דרך הפיאצה בלבד.
- על כל יום איחור או חלק ממנו, שאינו בתיאום עם המתרגל האחראי על התרגיל, יורדו 5 נקודות.
  - ניתן להגיש לכל היותר באיחור של 3 ימים (כאשר שישי ושבת נחשבים יחד כיום אחד בספירה).
  - הגשות באיחור יש לשלוח למייל של אחראי התרגיל בצירוף פרטים מלאים של המגישים (שם+ת.ז).
- הוראות הגשה נוספות מופיעות בסוף בתרגיל.
- לתרגיל שני חלקים. אין קשר בין חלק א' לחלק ב'!

נושא התרגיל: תכנות אסמבלי, קונבנציית קריאות, פסיקות תוכנה ו-IDT.

חומר דרוש: לחלק א' נדרשים הרצאות ותרגולים 1-4. לחלק ב' נדרשים גם תרגול 6 וההרצאה על קידוד פקודות.

חלק א – שגרות, קונבנציות ומה שביניהן

מבוא

בחלק א' של תרגיל בית זה נממש גרסה פשוטה יחסית של הצופן AES[1]. אין צורך בידע מוקדם בשביל להצליח בתרגיל וההוראות בכל שלב יהיו בלתי תלויות בהבנה של מהות הצופן ויתרונותיו. אם הנושא נשמע לך מעניין, יש קורסים מתקדמים בפקולטה בנושא (למשל, קריפטולוגיה מודרנית והגנה ברשתות).

AES הוא צופן בלוקים. המשמעות מבחינת התרגיל היא שכל חלק במידע ("בלוק") בגודל X ביט עובר הצפנה בצורה בלתי תלויה[2]. לדוגמה, אם המידע באורך 4X ביט, נחלק אותו ל-4 חלקים בגודל X וכל אחד מהם יוצפן בנפרד.

גרסת ה-AES אותה נממש בתרגיל היא AES128, כלומר פועלת על בלוקים באורך 128 ביט בכל פעם.

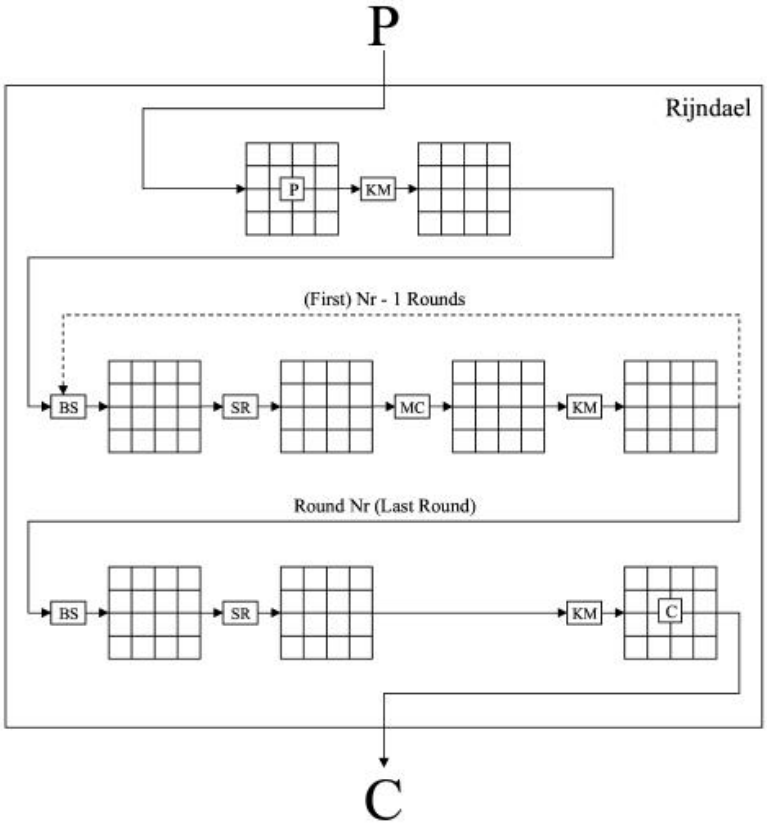
נוח להסתכל על כל בלוק כמערך דו מימדי בגודל 4x4, כשכל תא במערך בגודל בייט אחד בדיוק.

שלבי פעולה ב-AES

הצופן AES (לפי האלגוריתם של Rijndael) מורכב מ-4 פעולות, שחוזרות על עצמן מספר פעמים:

1. Key Mixing (KM) – ביצוע XOR בין מפתח (רצף כלשהו של 128 ביט הידוע לשני הצדדים) והבלוק.
2. Byte Substitution (BS) – כל בייט בבלוק (תא במערך) משנה את ערכו לפי פונקציה חח"ע, הידועה מראש ונתונה לכם. למשל, כל מופע של הבייט שערכו 0x00 יוחלף בערך 0x63 (אבל לא כל מופע של הערך 0x63 יוחלף ב-0x00!)
3. Shift Rows (SR) – השורה ה-i במערך עוברת shift ציקלי i פעמים שמאלה.
4. Mix Columns (MC) – פעולה כלשהי שאליה נתייחס כקופסה שחורה במהלך התרגיל. אין צורך להבין אותה.

להלן תיאור סכמטי של אופן פעולת ההצפנה המלא (נלקח מקורס "הגנה ברשתות". עבורנו Nr=10, P הוא Plaintext, כלומר מידע לא מוצפן ו-C הוא Ciphertext, כלומר מידע מוצפן):



[1] https://he.wikipedia.org/wiki/AES

[2] בפועל זה לא מדויק. יש שיטות הפעלה לצופן שיוצרות תלות (חשובה) בין בלוקים. אבל לא ניכנס לזה בתרגיל.

מה תעשו בתרגיל?

בתרגיל בית זה תממשו את הפונקציות KM, BS, SR ואת הפונקציה הסופית שמבצעת את ההצפנה כולה (המימוש של MC נתון לכם). ההנחיות לכל חלק יינתנו כעת. כל חלק ייבדק בנפרד וכמובן שמימוש 3 הפונקציות הראשונות ישמש אתכם בשלב כתיבת פונקציית ההצפנה המלאה.

את כל המימושים תשלימו בקובץ students_code.S (ואת החתימות המתאימות תוכלו למצוא בקובץ students_code.h) – קובץ S הוא קובץ אסמבלי, המתאים ל-gcc. אין הבדל אמיתי בינו לבין קובץ asm[3] (ויש שיטענו בצדק שדווקא קבצי S מתאימים יותר מ-asm לקורס).

<u>הערות חשובות לגבי מימוש התרגיל:</u>

1. שמרו על הקונבנציות!!! בתרגיל זה נכתוב קוד שמשלב אסמבלי ו-C ולכן קוד שלא ישמור על קונבנציות, ייכשל בטסטים.
2. אסור לכם להוסיף קבצים מלבד students_code.S.
3. אסור לכם להוסיף משתנים ל-section data מלבד sbox (אותו קיבלתם נתון).
   a. אם ברצונכם להשתמש במשתנים מקומיים, אתם כמובן יכולים (ואף מומלץ לעשות זאת) – אך תצטרכו לעשות זאת לפי הקונבנציות שנלמדו בקורס.
4. <u>מומלץ להיעזר ב-GDB בעת דיבוג הקוד</u>. מדריך לשימוש בדיבאגר GDB זמין באתר הקורס.
5. הערות נוספות מופיעות בסוף התרגיל.

שלב ראשון – Key Mixing

תחילה, <u>תממשו באסמבלי</u> את הפונקציה `keyMixing`. חתימת הפונקציה היא:

```
void keyMixing(uint8_t input[4][4], uint8_t key[4][4]);
```

כאשר `input` הוא בלוק אחד של 128 ביט ו-`key` הוא מפתח בגודל 128 ביט. שימו לב כי `input` ו-`key` הן כתובות זיכרון. הפונקציה אינה מחזירה ערך, אלא <u>משנה את תוכן הבלוק `input`</u>, כך שבסופה יכיל הבלוק את ה-XOR בין `input` ובין `key`.

ניתן להניח כי המצביעים `key` ו-`input` אינם NULL.

להלן תיאור סכמטי של פעולת שלב `Key Mixing`:

$$\begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix} \oplus \begin{bmatrix} SK^i_{0,0} & SK^i_{0,1} & SK^i_{0,2} & SK^i_{0,3} \\ SK^i_{1,0} & SK^i_{1,1} & SK^i_{1,2} & SK^i_{1,3} \\ SK^i_{2,0} & SK^i_{2,1} & SK^i_{2,2} & SK^i_{2,3} \\ SK^i_{3,0} & SK^i_{3,1} & SK^i_{3,2} & SK^i_{3,3} \end{bmatrix} = \begin{bmatrix} S'_{0,0} & S'_{0,1} & S'_{0,2} & S'_{0,3} \\ S'_{1,0} & S'_{1,1} & S'_{1,2} & S'_{1,3} \\ S'_{2,0} & S_{2,1} & S'_{2,2} & S'_{2,3} \\ S'_{3,0} & S_{3,1} & S'_{3,2} & S'_{3,3} \end{bmatrix}$$

כאשר $S'_{l,k} = S_{l,k} \oplus SK_{l,k}$ והפעולה $\oplus$ היא פעולת XOR.

<u>טסט לדוגמה</u>: נמצא בקובץ `test_KM.c` עם הקלט `test_KM_1` והפלט הצפוי `test_KM_1_res`. הסבר מפורט על הרצת הטסטים בשלב חמישי של חלק זה.

[3] https://stackoverflow.com/questions/34098596/assembly-files-difference-between-a-s-asm

שלב שני – Byte Substitution

בשלב השני, **תממשו באסמבלי** את הפונקציה `byteSubstitution`. חתימת הפונקציה היא:

```
void byteSubstitution(uint8_t input[4][4]);
```

כאשר `input` הוא בלוק אחד של 128 ביט. שימו לב כי `input` הוא כתובת זיכרון. הפונקציה תבצע החלפה של כל בייט במערך `input` עם הערך המתאים לו לפי פונקציית ההחלפה של AES.
לצורך מימוש פונקציה זו, נתונה לכם ב-`data section` בקובץ שאותו אתם משלימים, טבלה בשם `sbox`. טבלה זו מכילה 256 כניסות, כאשר כל כניסה בגודל בייט, והתוכן של כל כניסה היא ערך ההחלפה המתאים. כלומר, נניח שמערך `input` מופיע הערך `0x30`, אזי בכניסה ה-`48` (`zero base`) בטבלה `sbox` יופיע הערך אליו צריך לשנות את `0x30`.
הפונקציה אינה מחזירה ערך, אלא משנה את תוכן הבלוק `input`, כך שבסופה יכיל הבלוק בכל תא, את ערך ההחלפה המתאימה לאותו `byte` (לפי ערכו המקורי).
ניתן להניח כי המצביע `input` אינו `NULL`. **היזהרו לא לשנות את `sbox`** (הוא חלק מהקובץ שאותו תגישו).

טסט לדוגמה: נמצא בקובץ *`test_BS.c`* עם הקלט *`test_BS_1`* והפלט הצפוי *`test_BS_1_res`*. הסבר מפורט על הרצת הטסטים בשלב חמישי של חלק זה.
נציג כעת הדגמה מודרכת של פעולת *BS* על הקלט הנתון ב-*`test_BS_1`*:
הקלט בטסט, בהקסהדצימלי הוא: *`0x49206c6f7665204154414d21203c330a`* (המרה של *ASCII* להקסה) ולכן זה הבלוק עליו תפעל פונקציית ה-*BS*. נשים לב שהערך בכניסה ה-*`0x49`* בטבלת *sbox* הוא *`0x3b`*, של הכניסה ה-*`0x20`* הוא *`0xb7`* וכך ממשיכים עד שמתקבלת ההמרה הבאה:

49	*20*	*6c*	*6f*
76	*65*	*20*	*41*
54	*41*	*4d*	*21*
20	*3c*	*30*	*0a*

Byte Substitution →

3b	*b7*	*50*	*a8*
38	*4d*	*b7*	*83*
20	*83*	*e3*	*fd*
b7	*eb*	*c3*	*67*

שלב שלישי – Shift Rows

בשלב השלישי, **תממשו באסמבלי** את הפונקציה `shiftRows`. חתימת הפונקציה היא:

```
void shiftRows(uint8_t input[4][4]);
```

כאשר `input` הוא בלוק אחד של 128 ביט. שימו לב כי `input` הוא כתובת זיכרון. אם נסתכל על הבלוק של 128 הביטים הללו כמערך של `4x4` בייטים, נוכל להגדיר את פעולת הפונקציה כהזזה ציקלית של השורה ה-`i` (באשר `i=0,1,2,3`) בדיוק `i` פעמים שמאלה, כך שהשורה הראשונה נשארת במקומה, השניה זזה ציקלית מקום אחד שמאלה וכו'.
ניתן להניח כי המצביע `input` אינו `NULL`.
הפונקציה אינה מחזירה ערך, אלא משנה את תוכן הבלוק `input`, כך שבסופה כל שורה בבלוק עברה את ההזזה המתאימה.
להלן תיאור סכמטי של פעולת שלב `Shift Rows`:

1	*2*	*3*	*4*
5	*6*	*7*	*8*
9	*10*	*11*	*12*
13	*14*	*15*	*16*

Shift Rows →

1	*2*	*3*	*4*
6	*7*	*8*	*5*
11	*12*	*9*	*10*
16	*13*	*14*	*15*

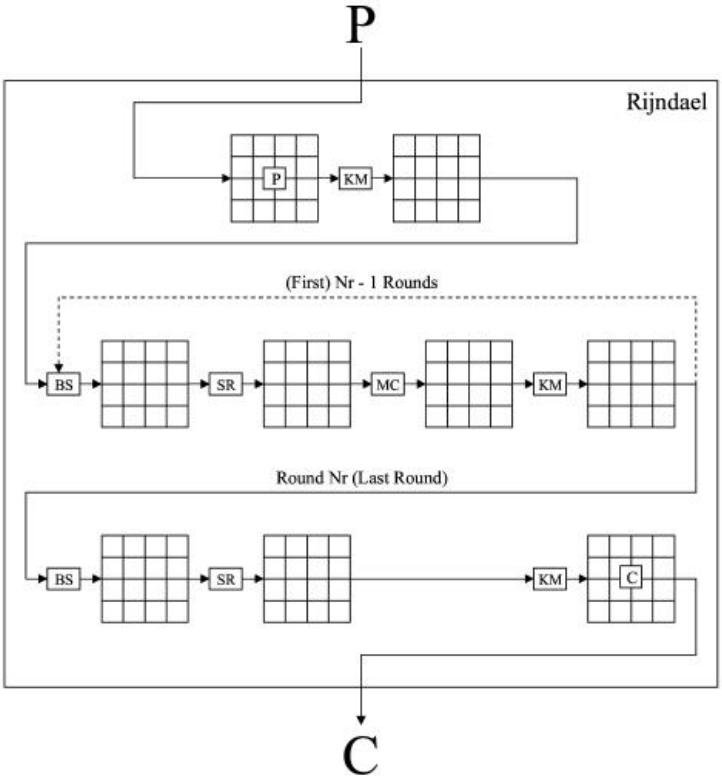
שלב רביעי – מימוש ההצפנה המלא

בשלב הרביעי (והאחרון בשלבי כתיבת הקוד), **תממשו באסמבלי** את הפונקציה `cipher`. חתימת הפונקציה היא:

```
void cipher(uint8_t input[][4][4], uint8_t key[4][4], uint8_t len);
```

כאשר `input` הוא **רצף של `len` בלוקים** בגודל 128 ביט ו-`key` הוא מפתח בגודל 128 ביט. שימו לב כי `input` ו-`key` הן כתובות זיכרון.

בפונקציה `cipher` תצטרכו לממש את הפעולות הבאות, לפי הסדר שמפורט בתרשים הבא (שכבר הוצג בתחילת התרגיל) על כל בלוק בנפרד:



במילים, עליכם לדאוג לכך שבסוף ריצת `cipher`, המידע שהגיע ב-`input` יהיה מוצפן ב-`AES` בצורה הבאה:

1. לכל בלוק בגודל 128 ביט במידע, הריצו את האלגוריתם הבא (באופן בלתי תלוי):
   a. הפעילו `Key Mixing` עם המפתח שהתקבל כקלט
   b. בצעו `9` (נזכיר כי בתרגיל `Nr=10`) פעמים:
      i. `Byte Substitution`
      ii. `Shift Rows`
      iii. `Mix Columns`
      iv. `Key Mixing` עם אותו מפתח שהתקבל כקלט[4].
   c. בצע עוד פעם אחת `Byte Substitution` על התוצאה של שלב `b`.
   d. בצע עוד פעם אחת `Shift Rows` על התוצאה של שלב `c`.
   e. בצע עוד פעם אחת `Key Mixing` על התוצאה של שלב `d`.

שימו לב כי את המימוש של כל השלבים כבר יש לכם: או במימוש שיצרתם בעצמכם בשלבים 1-3 בתרגיל, או כי המימוש נתון לכם בקובץ `aux_code.o` (וחתימת הפונקציה נתונה בקובץ ה-`h` המתאים).

הפונקציה אינה מחזירה ערך, אלא משנה את תוכן המידע `input`, כך שבסופה `input` מכיל את המידע המוצפן.

ניתן להניח כי המצביעים `key` ו-`input` אינם `NULL` וכי `len` גדול מ-0 ואין צורך לבדוק זאת. בנוסף, ניתן להניח כי ב-`input` יש בדיוק `len*128` ביט של מידע.

---

[4] במימוש האמיתי של AES, לא משתמשים באותו מפתח בכל שלב, אלא במפחות שנגזרים מהמפתח האמיתי. בתרגיל אנו ממשים גרסה חלשה יותר של AES.

שלב חמישי – טסטים

לכל חלק בתרגיל יהיו טסטים נפרדים. מצורפים לכם 5 טסטים לדוגמה: טסט ל-KM, טסט ל-BS, טסט ל-SR ושני טסטים להצפנה כולה. כמובן שהציון יינתן על סמך טסטים נוספים.

את הבדיקה תוכלו לעשות באמצעות קובץ ה-bash הנתון check_tests.sh. הפלט הצפוי:

```
student@ubuntu18:~/Winter21-22/HW2-wet$ ./check_tests.sh
1: PASS!
2: PASS!
3: PASS!
4: PASS!
5: PASS!
```

בנוסף לקבצים הללו, תוכלו לבדוק את עצמכם בעזרת המימוש של decrypt שנתון לכם בקובץ `aux_code.o` (וחתימת הפונקציה נתונה בקובץ ה-h המתאים).

הערות נוספות

רגע לפני הסוף, אנא קראו בעיון את ההערות, שנוגעות לדרך בה נבדק התרגיל (תוכלו לראות זאת גם בקובץ check_tests.sh):

1. אנא ודאו שהתוכנית שלכם יוצאת (מסתיימת) באופן תקין, דרך `main` של קובץ הבדיקה שקורא לפונקציה שלכם, ולא על ידי `syscall exit` שלכם, במקרה שבו השתמשתם באחד (וכמובן שגם לא בעקבות קריסת הקוד[5]). הערה זו נכתבה בדם ביטים (של קוד של סטודנטים מסמסטרים קודמים).

על מנת לוודא את ערך החזרה של התוכנית, תוכלו להשתמש בפקודת ה-`bash` הבאה: `echo $?` (תזכורת: ערך החזרה של התוכנית, במידה ויצאה בצורה תקינה, הוא הערך ש-`main` מחזירה ב-`return` האחרון שלה).

2. שימו לב ל-`timeout` איתו הטסטים ייבדקו. כתבו קוד יעיל ככל האפשר.
3. אם הכל עובד כשורה, אתם יכולים לעבור לחלק ב' של תרגיל הבית, ולאחריו לחלק ג', שהוא בסך הכל הוראות הגשה לתרגיל כולו (שימו לב שאתם מגישים את שני החלקים יחד!).

5

חלק ב – פסיקות (אין קשר לחלק א)

מבוא

קראו את כל השלבים בחלק זה, לפני שתתחילו לעבוד על הקוד.
בתרגיל זה נרצה לכתוב שגרת טיפול ב**פסיקת המעבד** המתבצעת כאשר מבצעים פקודה לא חוקית (כלומר, כאשר המעבד מקבל opcode שאינו מוגדר בו).

- כאשר המעבד מקבל קידוד פקודה שאינו חוקי, המעבד עוצר את ביצוע התוכנית וקורא לשגרת הטיפול בפסיקה ב-IDT.
- שגרת הטיפול נמצאת בקרנל, ובלינוקס שולחת סיגנל `SIGILL` לתוכנית שביצעה את הפקודה הלא חוקית. אפשר לראות זאת כאן למשל:

https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/arch/x86/kernel/traps.c#L321

נרצה לשנות את **קוד הקרנל** כך ששגרת הטיפול בפסיקה תשתנה. נעשה זאת באמצעות kernel module.

מה תבצע שגרת הטיפול החדשה?

שגרת הטיפול בפסיקה שלנו (שאותה **אתם הולכים לממש** באסמבלי בעצמכם, בקובץ ili_handler.asm), תיקרא **my_ili_handler** ותבצע את הדברים הבאים:

1. בדיקת הפקודה שהובילה לפסיקה זו. הנחות:
   - הניחו כי הפקודה השגויה היא פקודה של אופקוד בלבד. כלומר, לפני ואחרי ה-opcode השגוי אין עוד בייטים (אין legacy prefix, אין REX).
     - לכן, אורך הפקודה השגויה הוא באורך 1-3 bytes. בתרגיל זה הניחו כי אורך האופקוד השגוי הוא לכל היותר 2 בייטים.
2. קריאה לפונקציה what_to_do עם ה-byte האחרון של האופקוד הלא חוקי, כפרמטר.
   - היזכרו בחומר של קידוד פקודות:
     - אם האופקוד אינו מתחיל ב-0x0F, הוא באורך byte אחד.
     - אחרת (כן מתחיל ב-0x0F), אם הוא אינו מתחיל ב-0x0F3A או 0x0F38, אזי הוא באורך 2 בייטים. לכן, הניחו כי הבייט השני באופקוד אינו 0x3A או 0x38 (אין צורך לבדוק זאת).
   - דוגמאות:
     - עבור האופקוד 0x27, שהינה פקודה לא חוקית בארכיטקטורת x86-64, נבצע קריאה ל-what_to_do עם 0x27.
     - עבור האופקוד 0x0F04, גם לא חוקית, נבצע קריאה ל-what_to_do עם הפרמטר 0x04.
3. בדיקת ערך החזרה של what_to_do
   - אם הוא אינו 0 – חזרה מהפסיקה, כך שהתוכנית תוכל להמשיך לרוץ (תצביע לפקודה הבאה לביצוע מיד לאחר הפקודה הסוררת) וערכו של רגיסטר rdi% יהיה ערך ההחזרה של what_to_do.[6]
     - **שימו לב #1:** שימו לב ש-invalid opcode הינה פסיקה מסוג fault. חשבו מה זה אומר על ערכו של רגיסטר rip% בעת החזרה משגרת הטיפול ושנו אותו בהתאם.
     - **שימו לב #2:** היעזרו בספר אינטל, volume 3[7], עמוד 222, המדבר על הפסיקה שלנו, בכדי לוודא את תשובתכם ל"שימו לב #1" וגם כדי להחליט האם יש error code או לא.
     - **שימו לב #3:** what_to_do הינה שגרה שתינתן על ידנו בזמן הבדיקה. אין להניח לגביה דבר, מלבד חתימתה (כלומר - שם השגרה, טיפוס פרמטר הקלט וטיפוס ערך החזרה).
   - אחרת (הוא 0) – העברת השליטה לשגרת הטיפול המקורית.

---

[6] בעולם האמיתי אסור לשנות ערכים של רגיסטרים וצריך להחזיר את מצב התוכנית כפי שקיבלתם אותו. כאן אתם נדרשים כן לשנות ערך של רגיסטר, כך שמצב התוכנית לא יהיה כפי שהיה כשהתרחשה הפסיקה. זה בסדר, זה לצורך התרגיל 😊

[7] https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325384-sdm-vol-3abcd.pdf

לפני תחילת העבודה – מה קיבלתם?

בתרגיל זה תעבדו על מכונה וירטואלית דרך qemu (בתוך המכונה הוירטואלית - Virtualiception). על המכונה הזו, אנחנו נריץ kernel module [8] שיבצע את החלפת שגרת הטיפול לזו שמימשתן בעצמכן.
היות והקוד רץ ב-0 ring (kernel mode), במקרה של תקלה מערכת ההפעלה תקרוס. אך זה לא נורא! עליכם פשוט להפעיל את qemu מחדש.

לרשותכם נמצאים הקבצים הבאים בתיקייה part 2:

- initial_setup.sh - הריצו סקריפט זה לפני כל דבר אחר. סקריפט זה מכין את המכונה הוירטואלית לריצת qemu. <u>עליכם להריץ אותו פעם אחת בלבד</u> (לא יקרה כלום אם תריצו יותר, אך זה לא נחוץ).
  - יכול להיות שתצטרכו להריץ את הפקודה הבאה, לפני ההרצה (בגלל בעיית הרשאות):

```
chmod +x initial_setup.sh
```

- compile.sh - הריצו סקריפט זה בכל פעם שתרצו לקמפל את הקוד ולטעון אותו (עם המודול המקומפל) למכונה הוירטואלית של qemu (<u>שימו לב:</u> עליכם לצאת מ-qemu קודם).
  - גם כאן ייתכן ותזדקקו להרצה של chmod באותו אופן כמו בסעיף הקודם.
- start.sh - הריצו סקריפט זה כדי להפעיל את המכונה הוירטואלית של qemu, לאחר שקימפלתם את תיקיית code וטענתם אותה אל המכונה הוירטואלית של qemu.
  - גם כאן ייתכן ותזדקקו להרצה של chmod באותו אופן כמו בסעיף הקודם.
- filesystem.img - המכונה הוירטואלית אותה תריצו ב-qemu.
- קבצי הקוד שנכתוב, כחלק מהמודול (והיא זו שתקומפל ותרוץ לבסוף ב-qemu) וה-makefile:
  - ili_handler.asm, ili_main.c, ili_utils.c, inst_test.c, Makefile

איך הכל מתחבר - כתיבת המודול

בתיקיה code סיפקנו לכן מספר קבצים:

- **inst_test.c** – simple code example that executes invalid opcode. Use it for basic testing.
- **ili_main.c** – initialize the kernel module – provided to you for testing.
- **ili_utils.c** – implementation of ili_main's functionality – <u>YOUR JOB TO FILL</u>
- **ili_handler.asm** – exception handling in assembly – <u>YOUR JOB TO FILL</u>
- **Makefile** – commands to build the kernel module and inst_test.

ממשו את הפונקציות ב-ili_utils.c, כך שהשגרה my_ili_handler תיקרא כאשר מנסים לבצע פקודה לא חוקית. איך? Well, זהו לב התרגיל, אז נסו להיזכר בחומר הקורס. כיצד נקבעת השגרה שנקראת בעת פסיקה? פעלו בהתאם.
לאחר מכן, ממשו את הפונקציה **my_ili_handler** ב-ili_handler.asm שתבצע את מה שהוגדר בשלב II.

[8] למי שלא מכיר את המונח kernel module, בלי פאניקה (כי panic זה רע, אבל זה עוד יותר רע בקרנל. פאניקה! בדיסקו זה דווקא בסדר) – מדובר בדרך להוסיף לקרנל קוד בזמן ריצה (ניתן להוסיף לקרנל קוד ולקמפל לאחר מכן את כל הקרנל מחדש, אך כאן לא הזמן ולא המקום לזה). למעשה, נכתוב קוד שירוץ ב-kernel mode ולכן יהיה בעל הרשאות מלאות. אנו נדרש לזה – הרי אנו רוצים לשנות את קוד הקרנל.

זמן בדיקות - הרצת המודול

לאחר שסיימתם לכתוב את המודול, בצעו את השלבים הבאים:

1. הריצו את `./compile.sh` כדי לקמפל את קוד הקרנל ולהכניסו למכונת ה-QEMU
2. הריצו את `./start.sh` כדי לפתוח מכונה פנימית באמצעות QEMU
   a. משתמש: root, סיסמא: root

   כעת אתם בתוך ה-QEMU וכל השלבים הבאים מתייחסים לריצת QEMU.
3. `./bad_inst` כדי להריץ את הקוד `inst_test.asm`, עם הפקודה הלא חוקית (ולקבל הודעת שגיאה בהתאם). ניתן גם להריץ את `bad_inst_2` כדי להריץ את הקוד ב-`inst_test_2.asm`.
4. `insmod ili.ko` כדי לטעון את המודול שלכם (ודאו שהוא נטען ע"י הרצת dmesg)
5. `./bad_inst` כדי להריץ שוב, אך לקבל התנהגות שונה מהקודמת, מכיוון שהפעם השגרה שלכם נקראה.

דוגמת הרצה תקינה ב-QEMU (עם הטסטים inst_test ו-inst_test_2, ומימוש what_to_do שסופק לכם כדוגמה):

```
root@ubuntu18:~# ./bad_inst
start
Illegal instruction
root@ubuntu18:~# insmod ili.ko
root@ubuntu18:~# ./bad_inst
start
root@ubuntu18:~# echo $?
35
root@ubuntu18:~# rmmod ili.ko
rmmod: ERROR: ../libkmod/libkmod.c:514 lookup_builtin_file() could not open buil
tin file '/lib/modules/4.15.0-60-generic/modules.builtin.bin'
root@ubuntu18:~# ./bad_inst_2
start
Illegal instruction
root@ubuntu18:~# insmod ili.ko
root@ubuntu18:~# ./bad_inst_2
start
Illegal instruction
```

(what_to_do מחזירה את הקלט שלה פחות 4. בטסט הראשון הפקודה הלא חוקית היא 0x27, לכן ערך החזרה הוא 0x23, שזה 35. ערך זה הוא גם ערך היציאה של התוכנית, כי כך נכתב הטסט[9], לכן echo $? מדפיס 35. בטסט השני, הפקודה הלא חוקית היא 0x0f04, לכן ערך החזרה של what_to_do הוא 0 והתוכנית חוזרת לשגרה המקורית לטיפול, ששולחת את הסיגנל Illegal Instruction)

פקודות שימושיות

insmod ili.ko

(טוען את המודול ili.ko לקרנל ומפעיל את הפונקציה init_ko שבמודול)

rmmod ili.ko

(מפעיל את הפונקציה exit_ko שבמודול ומוציא את המודול ili.ko מהקרנל)

SHIFT + page up

(גלילת המסך למעלה)

SHIFT + page down

(גלילת המסך למטה)

הערות כלליות

על מנת להבין מה קורה בקרנל – תוכלו להשתמש בפונקציה ()print המוגדרת בקובץ ili_main.c, ולראות את הודעות הקרנל ע"י dmesg.

תיעוד של qemu ניתן למצוא כאן: https://qemu.weilnetz.de/doc/qemu-doc.html

[9] הטסט נכתב כך שמיד לאחר הפקודה הלא חוקית יש ביצוע של קריאת המערכת exit. אתם משנים את %rdi בשגרת הטיפול, לכן ערך היציאה של הטסט ישתנה בהתאם.

חלק ג' - הוראות הגשה לתרגיל בית רטוב 2

אם הגעתם לכאן, זו בהחלט סיבה לחגיגה. אך בבקשה, לא לנוח על זרי הדפנה ולתת את הפוש האחרון אל עבר ההגשה – חבל מאוד שתצטרכו להתעסק בעוד מספר שבועות מעכשיו בערעורים, רק על הגשת הקבצים לא כפי שנתבקשתם. אז קראו בעיון ושימו לב שאתם מגישים את כל מה שצריך ורק את מה שצריך.

עליכם להגיש את הקבצים בתוך zip אחד:

hw2_wet.zip

בתוך קובץ zip זה יהיו 2 תיקיות:

- part1
- part2

ובתוך כל תיקייה יהיו הקבצים הבאים (מחולק לפי תיקיות):

- part1:
  - students_code.S
- part2:
  - ili_handler.asm
  - ili_utils.c

בהצלחה!!!