האוניברסיטה הפתוחה

המחלקה למתמטיקה ולמדעי המחשב

מימוש אלגוריתם לערפול קוד בינארי

דו"ח פרויקט מתקדם זה מוגש כחלק מהדרישות לקבלת תואר

"מוסמך למדעים" .M.Sc במדעי המחשב

באוניברסיטה הפתוחה

החטיבה למדעי המחשב

על-ידי

שלמה ארצי

בהנחייתו של פרופ' אהוד גודס

דצמבר 2017

תוכן עניינים

1. תקציר

הפרויקט עוסק בערפול (Obfuscation) של תוכנה למטרות הגנה על הקוד. בעבודה המסכמת נבחנו מספר אלגוריתמים לערפול כאשר בפרויקט זה נבחר לממש אלגוריתם אחד מתוך מאמר [1].
דו"ח הפרויקט פותח בהגדרות ומטרות הערפול ולאחר מכן מתאר יישומים והיבטי אבטחה שלו. בהמשך מוצגות מטרות הפרויקט וחשיבותו. אלגוריתם הערפול מתואר לצד סוגי הפיענוחים השונים שהוא מתמודד איתם, ולאחר מכן היבטי מימושו בפרויקט: הארכיטקטורה והאלגוריתם. בהמשך מתוארים מגבלות והאתגרים במימוש האלגוריתם בסביבת הפיתוח וההרצה.
לאחר תיאור הקלט והפלט של התוכנית מוצגים דוגמאות ההרצה שנבחרו, תוצאות ומסקנות של הערפול המלווים בסיכום.

2. מבוא

מגמת המחשוב עדיין נמצאת בעיצומה. שלל המוצרים הכולל בתוכו תוכנה גדל במהירות עצומה. מחשוב ענן, מחשוב נישא, מחשוב במוצרים אלקטרוניים, חשמליים ומכאניים ופריטים לבישים המכילים מחשב ותוכנה. ארגונים משקיעים משאבים רבים כיום בפיתוח מוצרי תוכנה מסחריים, כאשר אחוז ההשקעה בתוכנה הולך וגדל ביחס להשקעה הכוללת במוצר. כתוצאה מכך, הידע והנכסים הנצברים ע"י הארגונים, הבא לידי ביטוי בתוכנה גדל. הארגונים כיום צריכים להתמודד עם הסיכון שהתוכנה שלהם תגיע לידי גורמים לא ידידותיים כמו מתחרים או תוקפים שינסו להשיג את הידע והנכסים הטמונים בה, או למצוא פרצות בכדי לחבל בתוכנה שלהם למטרות שונות.
האמצעים להשגת הקוד והידע כוללים טכניקות לשחזור קוד המקור והארכיטקטורה שלו מתוך תוצר מוגמר, ע"י שימוש בכלים אוטומטים או פיענוח ע"י פורצי תוכנה מתוחכמים.
עם גידול המשאבים המושקעים בתוכנה, חשיבות ההגנה עליהם גדלה. הגנה על הקוד ע"י ערפולו מקשה באופן ניכר על תוקף להבין אותו, לעשות בו שימוש או לשבש אותו לצרכיו.
מאידך נוזקות למיניהם משתמשים בשיטות ערפול בכדי להסוות את עצמן ע"י הפצת גרסה חדשה ליעד תקיפה חדש, ולמערכות הגנה יש צורך להתגבר על ההסוואה ולזהות את סוג הנוזקה.

2.1 הגדרת הערפול

תהליך ערפול מקבל כקלט תכנית P ומוציא כפלט תכנית 'P, כאשר תכנית 'P שומרת על הפונקציונאלית של P, קשה להבנה (unintelligible) [8] ומקשה על הנדסה הפוכה (reverse engineering) [1,2].
מטרות נוספות הרלוונטיות לתהליך הערפול הן שמירה על סדר גודל דומה של נפח התוצר המוגמר (הנספר בבתים) באופן סביר בהתאם לסביבת הביצוע, תרחישי השימוש השונים בו, ושמירה על ביצועים. כלומר, זמן ביצוע תרחישי הזרימה בהתקבל אותו קלט, לא יגדל באופן לא סביר לסביבת הביצוע והתרחישים השונים.
לדוגמא, לתוכנה המפיקה דוחות שעות, ייתכן שגידול נפחה מ- 100MB ל-300MB יתקבל לאור ריבוי המשאבים בסביבת הריצה, וזמן ביצוע איטי ב20% בתרחישי הזרימה הנפוצים מספק את לקוחותיו.

2.2 יישומים והיבטי אבטחה של ערפול

את הערפול ניתן לבצע על כמה מקורות. כשהוא מתבצע על קוד המקור, מייצרים קוד מקור השקול לו מבחינת הפונקציונליות ומהדרים אותו. שיטה אחרת, שעבודה זו מתמקד בה, היא ערפול של קוד אסמבלר המתקבל מתוצר בינארי מוגמר המוכן לריצה.

במקרה של תוצר שקוד המקור שלו הוא שפת ביניים כמו .NET או Java, כלי הערפול ממירים את קוד הביניים לקוד מקור, מחילים את שיטות הערפול עליו ומהדרים אותו שוב. במקרים של קוד סקריפט כמו Java Script שנשלח כמות שהוא ללקוח והערפול יתבצע על קוד המקור (וישלח מעורפל).

שימוש בשיטות ערפול שונות נמצא למעשה בשימוש בתעשייה לצורך הגנת נכסי תוכנה (IP-Intelectual Property), כגון התוכנה "Skype" המשמשת לתקשורת קול, וידיאו והודעות דרך האינטרנט [4], אבל גם ע"י נוזקות במטרה להקשות על זיהוין לפי חתימת הקובץ ע"י אנטי וירוסים, כאשר וירוס תוקף יכול לייצר גרסה חדשה ומעורפלת של עצמו בהתבסס על הקובץ הבינארי ממנו הוא רץ, מבלי להכיל את קוד המקור של עצמו, ולהפיץ אותה ליעד תקיפה חדש.

היבט אבטחה נוסף של מוצרים שלא עברו תהליך ערפול, או שהצליחו לפצח את הערפול שלהם, נוגע לחשיפת פגיעות וחורים באבטחה של מוצרי תוכנה או מערכות הפעלה [6], באמצעות הנדסה הפוכה של תוצרי תוכנה קיימים או עדכוני אבטחה שחושפים את פרצות האבטחה ומאפשרים לתוקפים להשתמש במידע בכדי לפרוץ למערכות בהן טרם הותקן עדכון התוכנה.
הידע המתקבל מקוד התוכנה לאחר הנדסה הפוכה, מאפשר גם להעתיק את ה-IP, לבצע שינויים בתוכנה למשל בכדי להסיר חתימה דיגיטלית, לשנות באופן בלתי חוקי את התוכנה ולהפיץ עותקים פיראטיים ולתקוף פרצות אבטחה כאמור.
תחום הערפול ופענוח הערפול נמצא במרוץ. מחד, מערפלי הקוד הנמנים על מפתחי תוכנה לגיטימיים המגנים על נכסיהם או מפתחי נוזקות ממציאים שיטות חדשות לערפול, ומאידך, חוקרים, מפתחי תוכנה והאקרים המיומנים בפענוח קוד ממציאים שיטות חדשות לפענוח תוצר מעורפל.

2.3 אתגרים ומגבלות הערפול

בדרך כלל, תוכנות מופצות באופן בינארי ללקוחותיהם ללא קוד מקור. להבדיל, גם נוזקות מפיצות את עצמן באותו האופן ללא קוד מקור, בכדי להקשות על זיהוין ע"י חקר הקוד שלהן ובלימתם.
מכאן, שהאתגר הוא לבצע את תהליך ההנדסה ההפוכה לתוצרים בינאריים בעיקר לאחר שלב הפצתם הן למפתחים שרוצים להפריד את שלב הערפול משלב ההידור והן לתוקפים שאינם רוצים לחשוף כיצד הנוזקות פועלות. האתגר בא לידי ביטוי בין היתר מהקושי לבצע דיס-אסמבלר (disassembly), כלומר לפענח נכון רצף בינארי לפקודות אסמבלר - כאשר יש מאות פקודות מעבד ולאלו נוספות פקודות חדשות בגרסאות חדשות, כל פקודה יכולה להתחיל במקום אקראי בזיכרון המיושר לפי b בתים[6], ואותה פקודה יכולה לתפוס לפעמים מספר משתנה של בתים למשל בגלל מספר וגודל משתנה של האופרנדים[2]. הפקודות עצמן מרובות ומורכבות להבנה, לדוגמא התיעוד של פקודות אלו בספר של אינטל IA-32 מתועד בספר השוקל יותר מ-5 קילו. ואכן, בפרויקט הגמר הבנת הפקודות השונות והאופרנדים הוסיפו מורכבות, לרבות פקודות הקפיצה השונות.

בגלל הקושי להבין פקודות אסמבלר ולשנותן ללא שינוי התנהגות התוכנה, יש צורך להמיר אותן לקוד פשוט יותר ברמה סמנטית גבוהה יותר. צורך זה נתקל בקושי לחלץ מפקודות האסמבלר קוד, מבנה נתונים וטיפוסים, המאפשרים להבין את תרשימי הזרימה התנאים ואת שאר הביטויים (predicates). לצורך העניין, ייצוג סמנטי ברמה גבוהה הוא קוד מקור של תוכנית בשפה העילית c.
האתגר בערפול קוד הוא למנוע הנדסה הפוכה של הקוד ע"י טכניקות ניתוח קוד ואנטי-ערפול (deobfuscation) המתפתחות לצד טכניקות הערפול השונות. השאלה האם קיימת טכניקה לערפול שלא תימצא לה טכניקה תואמת לאנטי ערפול נותרה ללא מענה מוחלט, אם כי הוכח באופן מתמטי שיש חסמים לערפול כך שלא ניתן לערפל תכונות מסוימות של משפחת פונקציות בתוכנה [8].
אולם, כלי הערפול השונים לא מתיימרים לערפל את כל תכונות התוכנה, אלא רק את חלק מהן המקשות על ההנדסה ההפוכה. לדוגמא טכניקות מסוימות לא תשננה את הפקודות שמוסיפות ערך 1 לרגיסטר, אלא תנסנה להסתיר את תרשים הזרימה, מבני הבקרה ומיקום הנתונים [1,2,5,7] המקשים באופן ניכר את תהליך ההנדסה ההפוכה.
מגבלה נוספת נובעת ממשפט רייס הטוען כי הבעיה האם תוכנית מקיימת תכונה מסוימת π אינה כריעה. מגבלה זו תואמת את הקושי של כלי הנדסה הפוכה להבין את התכונות הסמנטיות של תוכנה, כלומר היכולת להכריע האם הקוד מקיים תכונה או התנהגות מסוימת. באופן טכני קשה יותר להמיר קוד בקוד שקול לו, כך שהסמנטיקה שלו תשתנה אבל ההתנהגות לא תשתנה, מאשר לבצע שינוי תחבירי בו או טרנספורמציה אחרת [7].

ערפול הקוד הבא להסתיר תכונות או התנהגויות מסוימות של תוכנה, עומד בפני בעיות נוספות של היכולת להריץ אותה בסביבת סימולציה ולבחון את התנהגותה [6], כלומר היכולת להסיק מהם תרשימי הזרימה שלה, מה הקשר בין קלטים מסוימים ותרשימי זרימה, ומה האינטראקציה של התוכנה המורצת עם סביבתה כגון פניה לדיסק קשיח, רשת תקשורת וכו'.
כאמור הערפול יוצר מתוכנית P תוכנית חדשה 'P השקולה בפונקציונאליות אליה. באופן לא פורמאלי ואינטואיטיבי התוכנה תמשיך לפעול ללא שינוי ובאותו האופן, ותמשיך לפלוט את אותו הפלט בהינתן אותו הקלט. כלומר מגבלת ערפול נוספת אינהרנטית להגדרת הערפול היא שלא ניתן לערפל את הקשר בין הקלט והפלט שלה, כי אחרת הפונקציונאליות תשתנה.

בפרק הבא, נדון במטרת הפרויקט, בחשיבותו ונתאר את האלגוריתם שימומש.

2.4 מטרת הפרויקט

מטרת הפרויקט היא לממש אלגוריתם ערפול בהתאם למאמר של Cullen Linn& Saumya Debray[1] בצורה מלאה. המאמר מתאר את המימוש של הטכניקה ומשווה את תוצאותיו בפני מפענחים שונים. מאמר זה הוא בסיס למאמרים ומחקרים נוספים בתחום והוא מצוטט ע"י רבים אחרים.
הפרויקט הוא תוכנה המקבלת תוכנית בינארית P ומייצר תוכנית חדש P_{obf} מעורפלת, שתקינותה אינה נפגעת וזמן הביצועים שלה נפגע באופן זניח. האלגוריתם מורכב משני שלבים: שלב ראשון המגדיל את מספר האפשרויות להכנסת בתים משובשים ע"י טרנספורמציה פקודות של קפיצה על תנאי, ושלב שני המכניס בתים משובשים.
התוכנית המעורפלת, מכילה בתים נוספים של חלקי פקודות אסמבלר נוספות כך שפירושה באופן סטטי ייצור הוראות אסמבלר שאינן חלק מהתוכנית P. התוכנית P נבחרה כתוכנית exe, בפלטפורמה של 32 ביט על מערכת הפעלה Microsoft Windows 10.

2.5 חשיבות הפרויקט

כפי שהוזכר בסעיף המבוא, חשיבות הערפול עולה ככל שגדל הצורך להגן על נכסי תוכנה או להתגונן/לתקוף מפני נוזקות מעורפלות. בפרויקט זה, מתבצע ערפול על תכניות בינאריות מוגמרות על מ"ה Windows, כאשר באינטרנט לא נמצא תיעוד טכני על ערפול תוכנית מוגמרת לריצה. התיעוד שנמצא עוסק בשינויים קלים בקוד האסמבלר לצרכים שונים כגון דריסת פקודת אסמבלר קיימת באחרת. יתרה מכך, טכניקות לשינוי מערך הוראות האסמבלר הכולל שינוי כתובתן (בתוכנית מוגמרת) לא נמצא. מכאן חשיבותו של הפרויקט שמדגים יכולת זו על מספר תוכנות לדוגמא.
כמטרת צד הפרויקט מספק פלטפורמה ליצירת טרנספורמציות על קוד – במילים אחרות פלטפורמה להזרקת קוד אסמבלר לתוכנית מוגמרת. מטרה זו מושגת מכיוון שחלק התוכנה שעוסק בטרנספורמציה של הערפול עצמו הוא קטן וניתן להחלפה בטרנספורמציה שונה וחדשה באופן פשוט יחסית. דוגמאות לטרנספורמציות שיכולות להיות שימושיות הן הוספת מספר בתים המכילים פקודות אסמבלר המבצעות קריאה לפונקציה אחרת לצורך profiling (מספר קריאות לפונקציה, זמן ביצוע וכו'), שינוי תרשים זרימה, hacking, AOP – Aspect Oriented Programming וכו'.

2.6 אלגוריתם הערפול

מטרת האלגוריתם היא לגרום לקוד האסמבלר של התוכנית המעורפלת, להיות מפוענח באופן שונה מהאמיתי ולגרום ככל הניתן לשיעור גבוהה של פקודות שפוענחו לא נכון, בד בבד עם שמירת הפונקציונאליות המלאה של התוכנית. באופן זה, מפענח רצף הביטים לפקודות האסמבלר הכולל גם תוכנות סטנדרטיות לניפוי שגיאות (debugger), יקבל רצף הוראות שונה מהתוכנית האמיתית. מפענחים אלו מוגדרים כמפענחים סטטיים כפי שמבואר להלן:

2.6.1 פענוח סטטי

פענוח סטטי (static disassembly) הוא מונח המתאר משפחת פענוחים של תוצר בינארי מוגמר, הנעשים על תוכנית ללא הרצה בפועל.
שני סוגי פענוחים סטטיים יוצגו להלן: מעבר ליניארי ומעבר רקורסיבי.

2.6.2 מעבר ליניארי

מעבר ליניארי (linear sweep) הוא הנפוץ והפשוט ביותר. עושים בו שימוש רוב תוכנות פענוח אסמבלר ותוכנות לניפוי שגיאות (debugger). המעבר מבוצע ע"י מעבר רציף על שורות התוכנית כמודגם באיור 1, מהשורה הראשונה של הקוד, כאשר כל פקודת אסמבלר מפוענחת באופן עוקב. מעבר ליניארי מתבצע גם בתהליכים של אופטימיזציות בזמן חיבור (link) של התוכנית.
לשיטה זו יתרון על אחרות בכך שהיא מאפשרת זמן ביצוע ליניארי ביחס לגודל התוכנית *O(n)* בזכות מעבר בבת אחת על התוכנית וזיכרון בגודל *O(1)* כפי שמודגם באלגוריתם הבא:

```
global startAddr,endAddr;
proc DisasmLinear(addr)
begin
   while (startAddr ≤ addr ≤ endAddr) do
      I := decode instruction at address addr;
      addr += length(I);
   od
end

proc main()
begin
   ep          := address of the first executable byte;
   startAddr := ep;
   endAddr   := startAddr + text section size;
   DisasemLinear(ep);
end
```

איור *1*: אלגוריתם לפיענוח סטטי

2.6.3 מעבר רקורסיבי

מעבר רקורסיבי (recursive traversal) מבוצע ע"י בחינת תרחישי זרימה מנקודת הכניסה של התוכנית. השיטה בוחנת את פונקציית הכניסה, לאחר מכן את הפונקציות שהיא יכולה להפעיל (b1,b2,..), ובאופן רקורסיבי עבור כל פונקציה bi שנבחנה תבחן את הפונקציות שהיא מפעילה (c1,c2,...).
האלגוריתם הבא מדגים את השיטה:

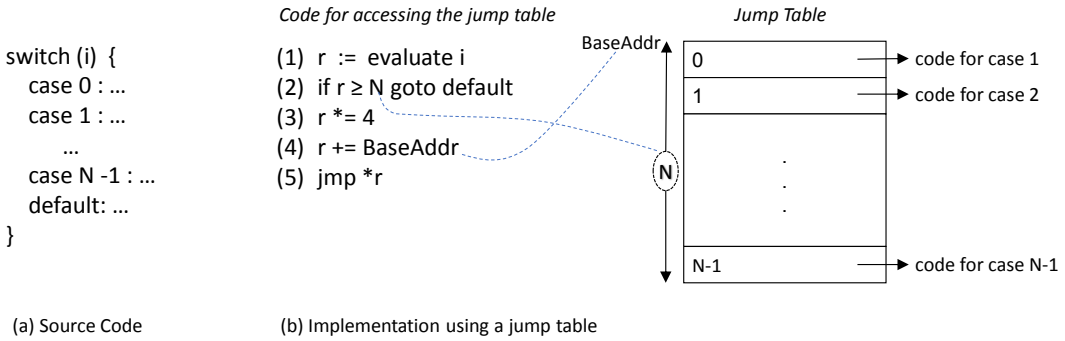
```
global startAddr, endAddr;
proc DisasmRec(addr)
begin
   while (startAddr ≤ addr < endAddr) do
      if (addr has benn visited already)
return;
      I := decode instruction at address addr;
      mark addr as visited;
      if (I is a branch or function call)
         for each possible target t of I do
            DisasmRec(t)
         od
         return;
       else addr += length(I)
    end

proc main()
begin
   startAddr := program entry point;
   endAddr  := startAddr  + text section
size;
   DisasmRec(startAddr);
end
```

איור 2: אלגוריתם לפיענוח רקורסיבי

יתרון השיטה היא היכולת להימנע מפענוח שגוי של רצף ביטים לפקודות אסמבלר עקב שגיאה מכוונת או מידע ששולב בתוכנית. כמו כן, מכיוון שהשיטה פועלת על פקודות בקרת זרימה, ניתן לבנות תרשים זרימה של התוכנית.
חסרון השיטה היא ההנחה שניתן להסיק את כל היעדים האפשריים ($b_1,b_{2,..}$) ממשפטי בקרת הזרימה, המתברר כקושי בקפיצות בקרה לא ישירות כגון טבלת קפיצה האופיינית למשפט switch-case בשפת c. ניסיון לקבוע את היעדים במקרים של קפיצות בלתי ישירות עלול לגרום לשגיאה בפענוח כך שחלקי קוד שהם יעדים של משפט בקרת זרימה לא יפוענחו או להפך, חלקי תוכנית שאינם קוד יפוענחו כקוד שגוי.

בפענוח הרקורסיבי ניתן להיעזר בשיטות שמנסות להתגבר על הקושי לפענח קפיצות בקרה בלתי ישירות. לדוגמה כמודגם באיור 3, בשיטה הבאה לפענוח משפטי switch-case [1], יש מערך של בתים רציפים של N כתובות של יעדי ביצוע, התואמות את יעדי הביצוע של המשפט בהתאם למשתנה כמודגם ב-3.a. בכדי למצוא את יעד הביצוע יש להעריך את ערך האינדקס התלוי בערך המשתנה (b.1), לבצע בדיקה האם ערך האינדקס בתחום המותר (b.2), להוסיף את ערכו המוכפל בגודל כל כניסה לכתובת התחלת הטבלה (b.3,b.4) ולקפוץ לכתובת המתקבלת מהחישוב (b.5).
בכדי לבצע את האלגוריתם נדרש לזהות את הכתובת של התחלת הטבלה ומספר יעדי הביצוע. זיהוי זה יכול לעשות ע"י סריקה אחורה מהפקודה לקפוץ (b.5) לפקודה המוסיפה את הערך המוכפל של האינדקס לכתובת התחלת הטבלה (b.4) המפענחת את התחלת הטבלה, ולאחר מכן סריקה אחורה ל - (b.2) בכדי לגלות את טווח האינדקסים. לאחר שמתגלה טווח האינדקסים, ניתן לקבוע את הכתובת ההתחלתית של טבלת הקפיצות, ולקבוע את רשימת היעדים של משפט הבקרה switch-case.



איור 3: שיטה אפשרית לפענוח טבלת קפיצה של משפט *switch-case*

2.6.4 פענוח דינאמי

פענוח דינאמי (dynamic disassembly) הוא מונח המתאר משפחת פענוחים המריצים תוכנה בפועל ובוחנים את אופן ביצועה. בזמן ביצוע התוכנית ניתן לחלץ את פקודות האסמבלר שהתבצעו. שיטת הערפול בפרויקט זה אינה נותנת מענה לפענוח דינאמי. חסרונות שיטה זו הם:

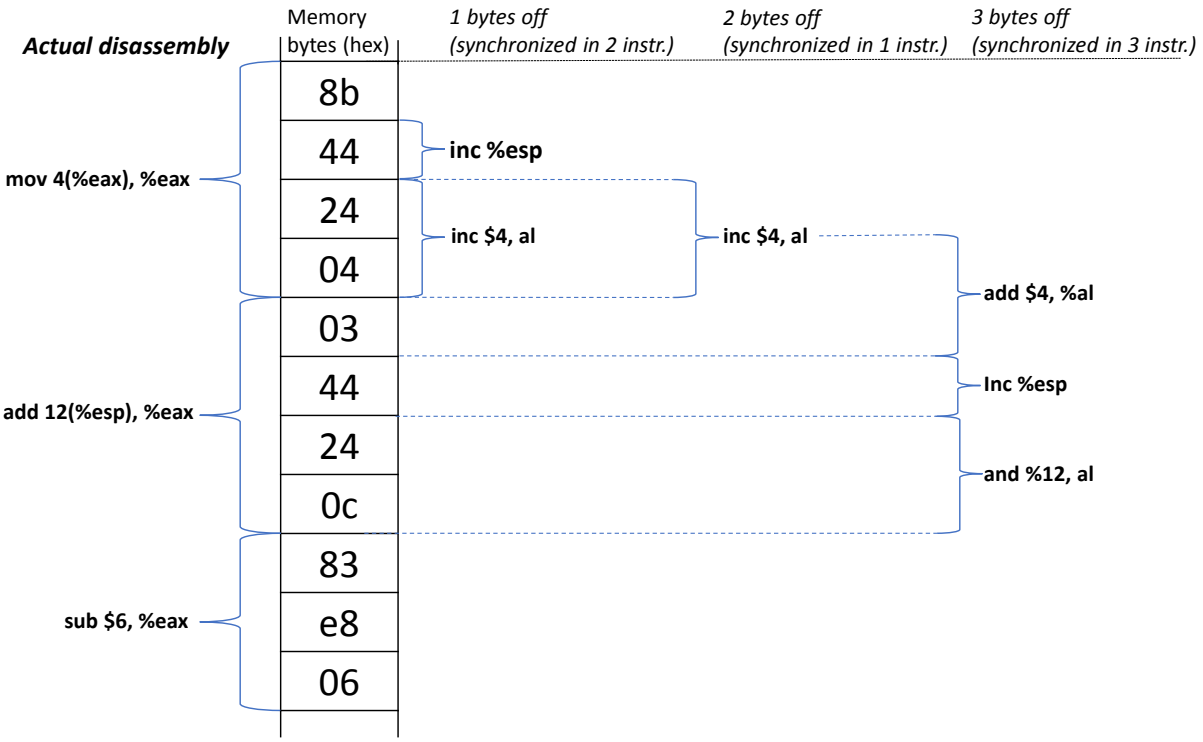
1. נדרש זמן לא ידוע בכדי לפענח את כל התוכנית – ניתן לבנות תוכנית שתריץ לולאה במשך מספר פעמים גדול מאוד (ככל שנרצה) ורק אז תקרא לפונקציה (שנקראת רק על ידה). באופן זה זמן הפענוח יכול להיות גדול באופן לא ידוע.
2. נדרשת סביבת הרצה אמיתית של התוכנה. עלול להיות קשה בגלל שתוכנות מסוימות דורשות סביבה מוגדרת לביצוע: רכיבים שהותקנו, מפתחות registry, הרשאות מסוימות, קבצים נדרשים, חיבור לרשת וכו'.
3. נדרשת הרצה תחת תוכנות אבטחה כמו אנטי וירוס, או הרצה בסביבה מבודדת ("sandbox") שיכולה לשנות גם את התנהגות הריצה אם התוכנה שרצה מזהה שהיא רצה בסביבה כזו.

2.6.5 מנגנון סנכרון המעבד

הערפול מכניס לפי האלגוריתם בתים משובשים לחלק הקוד של התוכנית במקומות שמאפשרים זאת מבלי לפגוע בביצוע התוכנית. בכדי להרחיב את מספר המקומות האלו, האלגוריתם מייצר מקומות נוספים בקוד בהם ניתן להכניס בתים משובשים.

בכדי להבין את תרומתם של הבתים המשובשים נסביר את פעולת המעבד והיכולת שלו להסתנכרן על פקודה נכונה: המעבד ופקודות האסמבלר תוכננו כך שהם יכולים להסתנכרן תוך מספר פקודות על הפקודה הנכונה בהכנסת שיבוש כלשהוא.
מרחק הסינכרון תלוי בגודל הפקודה עצמה, לדוגמא הפקודה *mov 4(%esp),%eax* שגודלה 4 בתים שתוכנם '8b 44 24 04', יכולה להיות מוסטת בבית 1, 2 או שלושה בתים (ראה איור 4):

1. היסט של בית אחד יצור פקודה חוקית, בעלת שלושה בתים עם התוכן '44 24 04' המתפרש כ- 'inc %esp'.
2. היסט של שני בתים ייצור פקודה חוקית עם התוכן '24 04' המתפרש כ- 'and $4,%al'.
3. הסט של שלושה בתים ייצור פקודה חוקית עם התוכן מהבתים של הפקודה הבאה מעבר לארבעת הבתים של הפקודה המקורית '04 03' המתפרשת כ-add$3,%al, כאשר '03' נגזר מהפקודה הבאה.



איור 4: דוגמא לפקודות המתקבלות מהיסטים שונים במקטע של שלוש פקודות אסמבלר

האלגוריתם בעבודה בנוי משני שלבים גסים: בשלב הראשון הרחבת התוכנית לשם יצירת פקודות נוספות שלפניהן אפשר להכניס בתים משובשים, והשני הכנסת בתים משובשים.
אולם, מה הם המקומות שבהם ניתן להכניס בתים משובשים? המקומות הם אלו שהבקרה לא תעבור בהם אף פעם.
בכדי למצוא אותם יש לזהות את הפונקציות ואת הבלוקים הבסיסיים בתוכנית, כפי שיוגדרו להלן:

2.6.6 בלוקים בסיסיים

בלוקים בסיסיים (basic blocks), הם קטעי קוד רציפים, שבהן כל פקודה השייכת להן, תוכל להתבצע רק אם הפקודה שלפניה התבצעה בדיוק לפניה [3] ללא כל פקודה אחרת ביניהן. בבלוקים בסיסיים אין פקודות קפיצה כגון אלו המעבירות את הבקרה ליעד אחר, אלא רק בסופם.
לבלוק הבסיסי יש נקודת כניסה אחת, כך שאף פקודה בתוכו היא אינה יעד של קפיצה ממקום אחר בתוכנית. מכך משתמע שביצוע פקודה אחת יגרום בהכרח לביצוע כל הפקודות העוקבות פעם אחת (לכל היותר, כי במצב של פסיקה למשל בגלל שגיאת חלוקה ב-0 הפקודות שאחרי חילול הפסיקה לא יתבצעו).
הבלוק שאליו מועברת הבקרה לאחר הפקודה האחרונה בבלוק הבסיסי נקרא ה*בלוק העוקב (block successor)*, כאשר הבלוקים שהעברת הבקרה הגיעה מהם נקראים ה*בלוקים הקודמים (predecessor's blocks).*

בלוק בסיסי מתחיל בפקודה אליו מגיעה פקודת העברת הבקרה או הפקודה הראשונה בתוכנית (כגון כניסה לפונקציה), ונגמר בפקודת העברת הבקרה (כגון קפיצה לא מותנית או חזרה מקריאה לפונקציה) הבאה וכולל אותה.

2.6.7 הכנסת בתים משובשים

אם כך, אחרי בלוק בסיסי שמסתיים בפקודת העברת בקרה (JUMP) ללא תנאי ניתן להכניס בתים משובשים שהמעבד לא יקרא באף תרחיש. אולם איך יבחרו הבתים המשובשים? בד"כ בוחרים פקודת אסמבלר ארוכה

ומחסירים ממנה בית אחד. לדוגמא: *and eax* מיוצג ע"י הבתים {81 ,e0 ,03, 04, f1, f2} , ממנו מחסירים את הבית האחרון. לאחר פקודת JUMP מכניסים את הבית הראשון מאלו ובודקים אחרי כמה פקודות המעבד מסתנכרן על הפקודה הנכונה. לאחר מכן מכניסים שני בתים, וכן הלאה עד לאורך המלא של הפקודה פחות אחד. לבסוף, בוחרים את חלק הפקודה שבו הסנכרון יתבצע להוראה הרחוקה ביותר בבלוק הבסיסי הבא (ולא אחריו). בעבודה נבחרו מספר פקודות ארוכות, ובכל פעם באופן אקראי נבחרת פקודה אחרת.

אולם בד"כ שיעור המקומות בתוכנית שבהן יש הוראות JUMP לעומת כלל ההוראות הוא נמוך, עובדה שמאפשרת להכניס מעט בתים משובשים ולהגיע לשיעור נמוך יחסית של פירוש הוראות שגויות לעומת נכונות.

בכדי להתגבר על בעיה זו, בשלב הראשון מעלים את מספר האפשרויות להכנסת בתים משובשים ע"י החלפת קפיצה עם תנאי להוראה מסוימת לקפיצה עם תנאי ההפוך (לדוגמא 'ne'-not equal→'eq'-equal), והוספת JUMP מיד אחר כך, כפי שמתואר באיור 5 כאשר cc הוא תנאי, ו- $\overline{cc}$ הוא התנאי ההופכי.

$$b_{cc}\ Addr \quad \xRightarrow{T} \quad \begin{array}{l} b_{\overline{cc}}\ L' \\ jmp\ Addr \\ L' \end{array}$$

איור 5: טרנספורמציה לקפיצה על תנאי

3. מימוש הפרויקט

הרעיון שעומד בבסיס הפרויקט הוא פיתוח פלטפורמה גנרית לביצוע טרנספורמציות על קוד. הפלטפורמה מאפשרת לספק פונקציית המרה של הוראת אסמבלר להוראה/הוראות אחרת בהינתן ההקשר של הפונקציה והבלוק הבסיסי של ההוראה המומרת. הפלטפורמה דואגת לטפל בשאר ההיבטים הקשורים לשינוי הוראת אסמבלר אחת לאחרת.
במימוש מומשו שתי טרנספורמציות (שהם שני השלבים באלגוריתם הערפול): הראשונה מגדילה את מספר האפשרויות להכנסת בתים משובשים, והשנייה מכניסה בתים משובשים.

הפרויקט מורכב מהחלקים הבאים:

א. ניתוח האלגוריתם ודרישותיו ועיצוב הארכיטקטורה הנדרשת למימושו.
ב. מימוש הארכיטקטורה.
ג. בנית תוכנית המדגימה את האלגוריתם והשימוש בו.

3.1 תכולות העבודה

א. פענוח מבנה של קובץ בינארי מוגמר לריצה.
ב. פענוח חלק הקוד של התוכנית לפקודות אסמבלר.
ג. פענוח פקודות האסמבלר לפונקציות המכילות בלוקים בסיסיים.

ד. ביצוע טרנספורמציה ראשונה.

ה. ביצוע טרנספורמציה שניה.

ו. כתיבת התוכנית מחדש.

3.2 סביבת פיתוח וכלי עבודה

התוכנית פותחה על Windows 10 ו-Visual Studio 2017 על שפת .NET. C# 7.0 – כל הטכנולוגיות העדכניות ביותר נכון לזמן הפיתוח.
בנוסף נעשה שימוש בספריית PeNet לצורך פענוח ה Header של תוצר בינארי מוגמר.
כמו כן התוכנית משתמשת בספריית SharpDisasm בה בוצעו הרחבות, ותיקוני באגים בכדי לאפשר את הערפול.
הפרויקט עושה שימוש גדול בDesign Pattern של Inversion Of Control ומשתמש ב Dependency Injection.
לשם כך נעשתה אבסטרקציה לIoC Container מעל Unity, המאפשר בין היתר רישום שירותים ויצירתם תוך כדי הזרקת תלויות.
יצירת התוכנית המזריקה תלוית אפשרה לבצע בדיקות יחידה (Unit Testing) בקלות, ולשם כך נעשה שימוש ב Microsoft Tests המובנים בכלי הפתוח.
ה-UI פשוט ומבוסס על פלט לתוכנת Console.

3.3 אלגוריתם

נבצע את הפסאודו קוד הבא:

1. בצע פונקציה להכנת הקוד (GetCodeInMemoryLayout).
2. הכן רשימה של שתי טרנספורמציות (הרחבת מקומות עם jump, הכנסת הוראה משובשת).
3. עבור כל טרנספורמציה:
    1. בצע פונקציית טרנספורמציה כללית חלק 1 - Transform.
    2. בצע פונקציית המרה ספציפית.
    3. בצע פונקציית טרנספורמציה כללית חלק 2 - Transform.
4. כתוב לזיכרון :
    1. כתובות אסמבלר חדשות.
    2. טבלת relocation חדשה.
    3. חלק הקבועים בתוכנית.
    4. כתובת הפקודה הראשונה בקוד.
5. כתוב לקובץ חדש את הקוד לאחר ערפול.

פונקציית הכנת הקוד (GetCodeInMemoryLayout):

1. קריאת ה-Header של התוכנית.
2. איתור חלקי התוכנית המכילים את פקודות האסמבלר, טבלת הRelocation, וקבועים.

3. עבור החלק המכיל את פקודות האסמבלר, נתח את פקודות האסמבלר. זהה את הפונקציות ואת הבלוקים הבסיסיים המרכיבים אותם.
4. נתח ושמור את טבלת הRelocation ואת המקומות בחלק הקבועים המכיל כתובות של פקודות אסמבלר.
5. שמור את ההוראה של כניסת הקוד וכתובתה.

פונקציית טרנספורמציה כללית חלק 1- Transform:

1. שמור מפה של כל הפקודות בהן יש אופרנד שהוא כתובת, ואת ערך האופרנדים עצמם.
2. שמור מפה של כתובות התוכנית לפקודות עצמן.
3. הרחב את גודל האופרנדים המכילים בתוכם כתובת ל-4 בתים.

פונקציית טרנספורמציה כללית חלק 2- Transform:

כתוב לזיכרון את הכתובות החדשות:

1. באופרנדים של הפקודות .
2. בטבלת הrelocation.
3. בחלק הקבועים המכיל כתובות,
4. של הכתובת כניסה לתוכנית.

3.4 ארכיטקטורה

העיקרון המנחה היה מימוש כל הנדרש בכדי לבצע את הטרנספורמציות ולגרום לתוכנית לדוגמא לעבוד לאחר הערפול. במהלך העבודה ולאחר כתיבת הטרנספורמציות התגלו עוד ועוד נושאים חסרים שנדרשו לטיפול, כאשר התוכנית המעורפלת קרסה.
כאמור התוכנה מאמצת גישה של Dependency Injection ומשתמשת בIoC Container בכדי להזריק תלויות ולהגדיר את אורך חייהם.

3.4.1 IoC and Dependency Injection

בכדי לעודד Decoupling והפרדה היבטים ככל שניתן למחלקות שונות, נעשה שימוש בתבנית (Design Pattern) של Inversion Of Control (IoC) Container ו-Dependency Injection . הרעיון הוא הגדרת ממשקים לכל מחלקה שאינה מבנה נתונים. כל מחלקה התלויה במחלקות אחרות מקבלת בפונקציית הבנאי (Constructor) את המחלקות ע"י הממשקים שלהם באופן אוטומטי (ה-Container דואג לכך).
במחלקה בה מוגדר ה IoC Container מתבצע רישום כל הצמדים מחלקה-ממשק תוך כדי קביעת סוג אורך החיים – Transient ו Singleton.
התוכנה נדרשת לגשת לContainer במקרים יוצאי דופן כמו באתחול המחלקה המערפלת בתוכנת הConsole, או איתור שירותים ליצירת מחלקות במחלקות Extensions. ככלל העיקרון המנחה הוא שמחלקה ראשית (`ICodeObfuscator`) מכילה בעץ תלויות את כל התלויות הנדרשות לה, ושמוזרקות אליה באופן אוטומטי.

3.4.2 מבנה הספריות

התוכנית בנויה משלושה חלקים:

1. ספרייה בשם ObfuscationTransform המאגדת את כל הפונקציונאליות של אלגוריתם הערפול.
2. הכניסה הראשית לתוכנית שהיא תוכנת Console בשם ObfuscationTransform.Console המפעילה את הספרייה, מספקת את הקלט לספריה ObfuscationTransform, כותבת את הקובץ ומציגה את התוצאות.
3. ספריית בדיקות יחידה בשם ObuscationTransform.Test .

לפרויקט המאגד את הפתרון נוספו שתי ספריות:

1. ספריית PeNet המפענחת את ה Header של כל קובץ ריצה/ספרייה.
2. ספריית SharpDisasm המפענחת רצף בתים לאסמבלר. לספרייה הוכנסו שינויים בכדי לתמוך בדרישות הערפול וגם תיקוני באגים.

3.4.3 מבנה מחלקות

עיקר הפונקציונאליות מומשה בObfuscationTransform. להלן המחלקות המשמעותיות:

- `namespace ObfuscationTransform.Core`
  מאגד בתוכו מספר מחלקות בסיס המהוות שפה משותפת ובסיס לבנייה של שאר המחלקות.

  - `class Code : ICode`
    מחלקה המתארת את הקוד בתוכנית. מחלקה זו הינה מאוד מרכזית שכן היא מועברת לטרנספורמציות כפרמטר היחיד, ועליה מתבצעת הטרנספורמציות, כך שעליה להכיל את כל המידע הדרוש בכדי לבצע טרנספורמציות.
    המחלקה מכילה את רשימת כל הוראות האסמבלר (רשימה דו כיוונית), רשימת הפונקציות ובלוקים בסיסיים. כמו כן המחלקה מכילה מידע על אופן ייצוג הקוד בזיכרון (כתובות וירטואליות, נקודת כניסה, גודל חלק הקוד וכו') ואופן ייצוג הקוד בקובץ (היסט חלק הקוד מהתחלת הקובץ) . מידע זה מיוצג ע"י מחלקת *CodeInMemoryLayout*

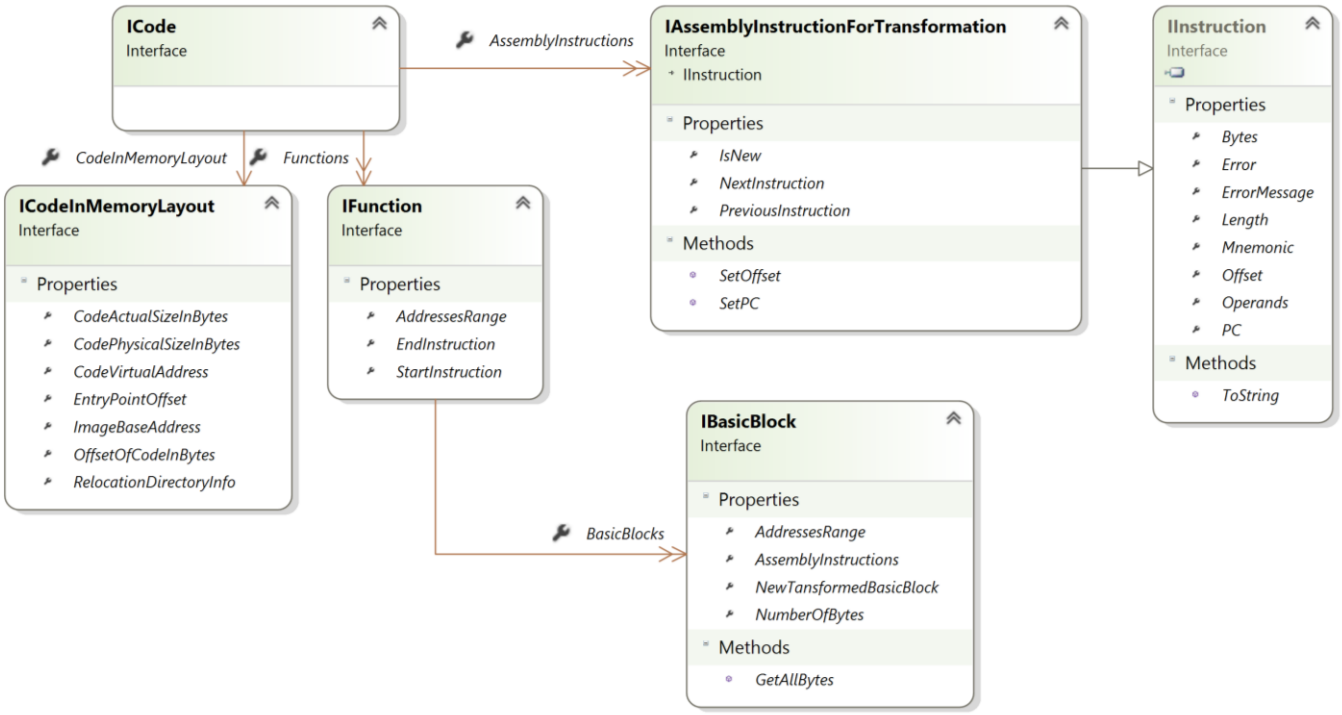
  - `class Function : IFunction`
    מתאר פונקציה – אוסף של בלוקים בסיסיים.

  - `class BasicBlock : IBasicBlock`
    מתאר בלוק בסיסי כרצף הוראות אסמבלר

  - `class AssemblyInstructionForTransformation : IAssemblyInstructionForTransformation`

    מתאר פקודת אסמבלר מורחבת (הרחבת הגדרה מספריית *PeNet*) המכילה מידע האם הפקודה חדשה לטרנספורמציה, ומצביעים לפקודה הבאה והקודמת המאפשרים ליצר רשימה מקושרת דו -כיוונית.

  - `class CodeInMemoryLayout : ICodeInMemoryLayout`
    הגדרה של פריסת הקוד בזיכרון ובדיסק – איפה הקוד מתחיל בכתובת וירטואלית, בדיסק, מה גודלו וכו' בנוסף מכיל מידע על טבלת ה-relocation ע"י מבנה *RelocationDirctoryInfo*.



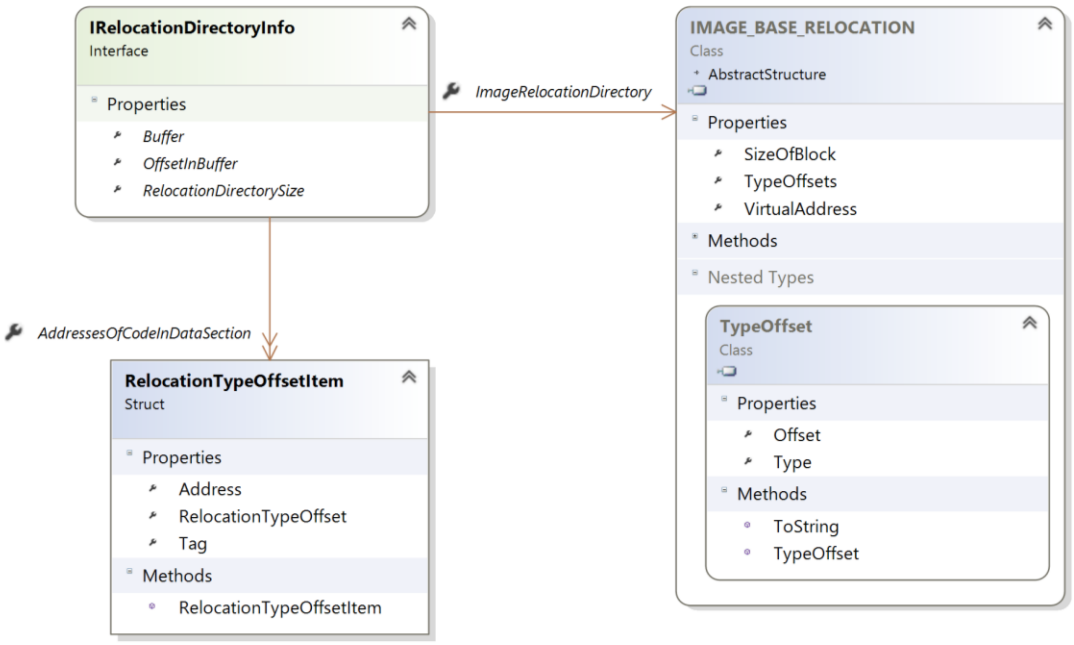
איור 6: מחלקות עיקריות *ICode,IBasicBlock,IFunction,IAssemblyInstructionForTransformation,IInstruction* והקשרים ביניהם.

- `struct RelocationDirectoryInfo : IRelocationDirectoryInfo`

מתארת את טבלת הrelocations המגדירה את כל הכתובות בתוכנית העלולות להכיל כתובות בתוכנית.
לכתובות אלו יתווסף ערך הבסיס של הכתובת הווירטואלית - בעליית הProcess הבסיס לכתובתו הווירטואלית נקבע בזמן עליה באופן אקראי, ובסיס זה מתווסף לכל המקומות בהם יש כתובות בתוכנית.

- `struct RelocationTypeOffsetItem`

כניסה בטבלת הrelocation table המכילה כתובת אבסולוטית של הכתובת בזיכרון היכולה להכיל כתובת.



איור 7: מחלקות המייצגות מידע של טבלת ה-*Relocation*.
המחלקה המיוצגת ע"י *IRelocationDirectoryInfo* מייצגת את כל כניסות הטבלה בעזרת רשימה של פריטים מסוג *RelocationTypeOffset* המכילים מידע על הכתובת האבסולוטית של האופרנדים המכילים כתובת.
המחלקה *IMAGE_BASE_RELOCATION* שייכת לספרייה צד שלישי *PeNet*. המחלקה מייצגת גם אוסף כניסות בטבלת ה-*Relocation*, אולם היא הכניסה היא עבור טווח כתובות של 1000_{16} (*hexadecimal*), וכל שינוי בה מעדכן את מרחב הזיכרון של התוכנית שאותה הוא מייצג.

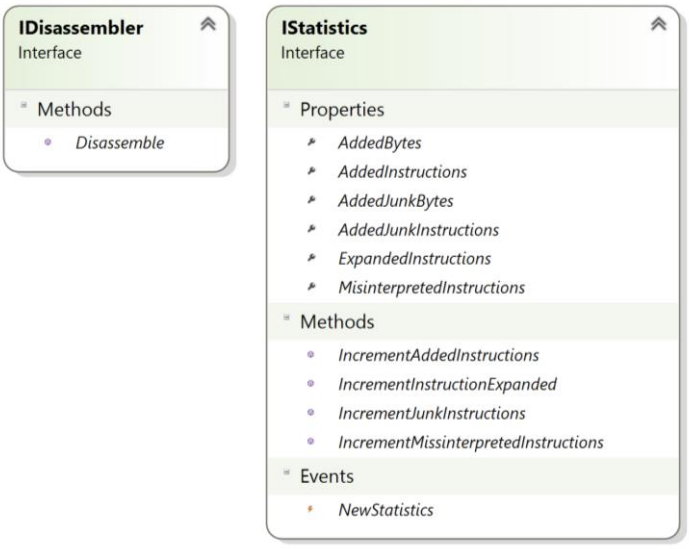
- `class Disassembler : IDisassembler`

מפרש מערך בתים להוראות אסמבלר

- `class Statistics : IStatistics`

מחלקת סטטיסטיקות המאפשרת להזין מידע על:
1. הוראות שנוספו
2. מספר בתים שנוספו
3. מספר הוראות שיתפרשו לא נכון
4. מספר הוראות קפיצה שהורחבו
5. מספר הוראות משובשות שהתווספו
6. מספר בתים משובשים שהתווספו

המידע מאפשר לקבוע את שיעור ההטעיה – כלומר יחס ההוראות השגויות שפורשו לעומת סך ההוראות התקינות שפורשו.



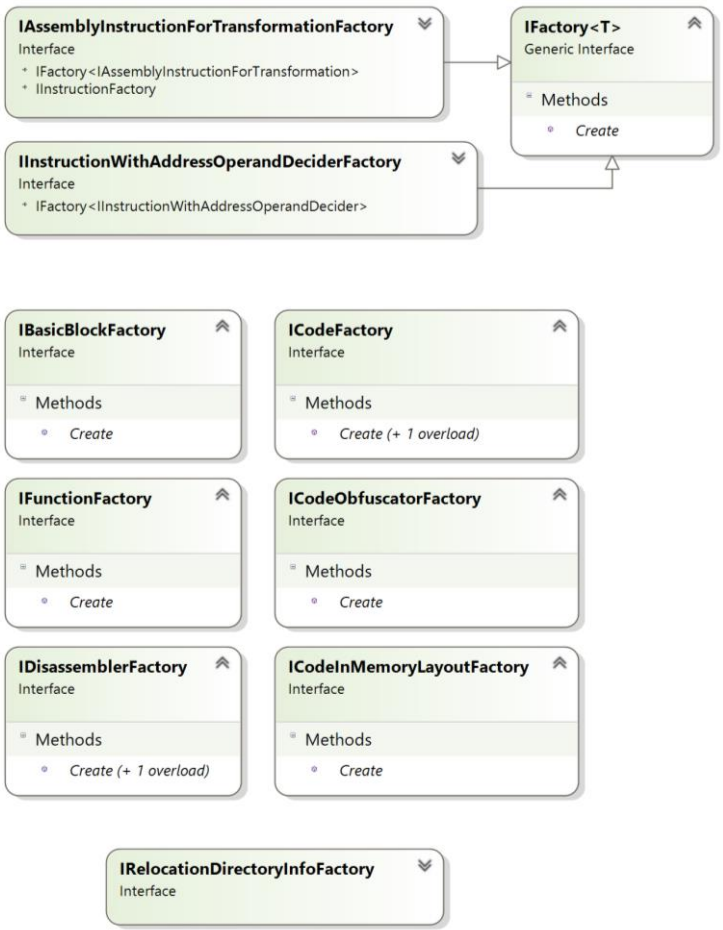
איור 8: ממשק המייצג סטטיסטיקה (*IStatistics*) ומפענח פקודות אסמבלר (*IDisassembler*).

- `class InstructionWithAddressOperandDecider : IInstructionWithAddressOperandDecider`

מחלקה שמאפשרת לבחון האם הוראת אסמבלר מכילה אופרנד עם כתובת. המחלקה נדרשת בכדי למפות את הכתובות שיהיה צריך לעדכן באופרנדים לאחר שינוי הפקודות בתוכנית ועדכון הכתובות של ההוראות בתוכנית.

- `namespace ObfuscationTransform.Core.Factory`
  - `class Factory<T> : IFactory<T>`

  הגדרה של מחלקת ייצור גנרית למחלקות שהוגדרו בקוד. מחלקות הייצור שהוגדרו בתוכנית יורשים את מחלקה זו. לדוגמא `.CodeFactory`



איור 9: ממשקים המייצגים מחלקות יצירה שונות *(Factories)*. הממשקים שיורשים את *IFactory* לא מקבלים פרמטר לפונקציית *Create* לעומת שאר המחלקות הדורשות פרמטרים.

- `namespace ObfuscationTransform.Parser`
  - `class CodeParser : ICodeParser`

מחלקה המפענחת קוד מרצף פקודות אסמבלר. הקוד בנוי מרשימת פקודות ופונקציות.

  - `class FunctionParser : IFunctionParser`

מחלקה המפענחת פונקציה יחידה מרצף הוראות אסמבלר. המחלקה משתמשת בשלושת המחלקת הבאות בכדי לקבוע היכן פונקציה מתחילה ונגמרת, ומהם הבלוקים הבסיסיים שמרכיבים אותה. המחלקה מכילה רשימת בלוקים בסיסיים.

  - `class FunctionPrologParser : IFunctionPrologParser`

מחלקה המפענחת מהם פקודות האסמבלר המייצגות התחלת פונקציה.

  - `class FunctionEpilogParser : IFunctionEpilogParser`

מחלקה המפענחת מהם פקודות האסמבלר המייצגות סוף פונקציה.

  - `class BasicBlockParser : IBasicBlockParser`

מחלקה המפענחת בלוק בסיסי שלם מרצף פקודות אסמבלר. המחלקה מכילה את רצף פקודות האסמבלר המרכיב אותה.

- `namespace ObfuscationTransform.Parser.Factory`
- `namespace ObfuscationTransform.Transformation`
  - `class CodeTransform : ICodeTransform`

מחלקה הממירה קוד לקוד אחר. המחלקה מייצרת מבנה חדש של קוד המכיל רשימת הוראות, פונקציות ובלוקים חדשים.

  - `class InstructionWithAddressOperandTransform : IInstructionWithAddressOperandTransform`

מחלקה המייצרת הוראה חדשה המתבססת על הוראה קיימת.
דוגמאות:

    - הוראת jump חדשה עם ערך אופרנד הקופץ לכתובת מעודכנת.
    - הוראת jump המכילה אופרנד של 4 בתים במקום 2 בתים.
    - הוראת קפיצה על תנאי ההופכית לפקודה עם תנאי אחר. כגון המרת פקודת JL-Jump if less לפקודה JGE-Jump if greater or Equal.

  - `class PeTransform : IPeTransform`

מבצע טרנספורמציה על קובץ בינארי מוגמר (בד"כ dll/exe) המיוצג בזיכרון, לקובץ חדש.
הטרנספורמציה לקובץ חדש תתבצע ע"ס מחלקת `ICode` המייצגת מידע על קוד (חדש שעבר טרנספורמציה).

  - `class RelocationDirectoryFromNewCode : IRelocationDirectoryFromNewCode`

מחלקה המייצרת טבלת relocation חדשה ע"ס טבלה קודמת וקוד חדש.

  - `abstract class TransformationBase`

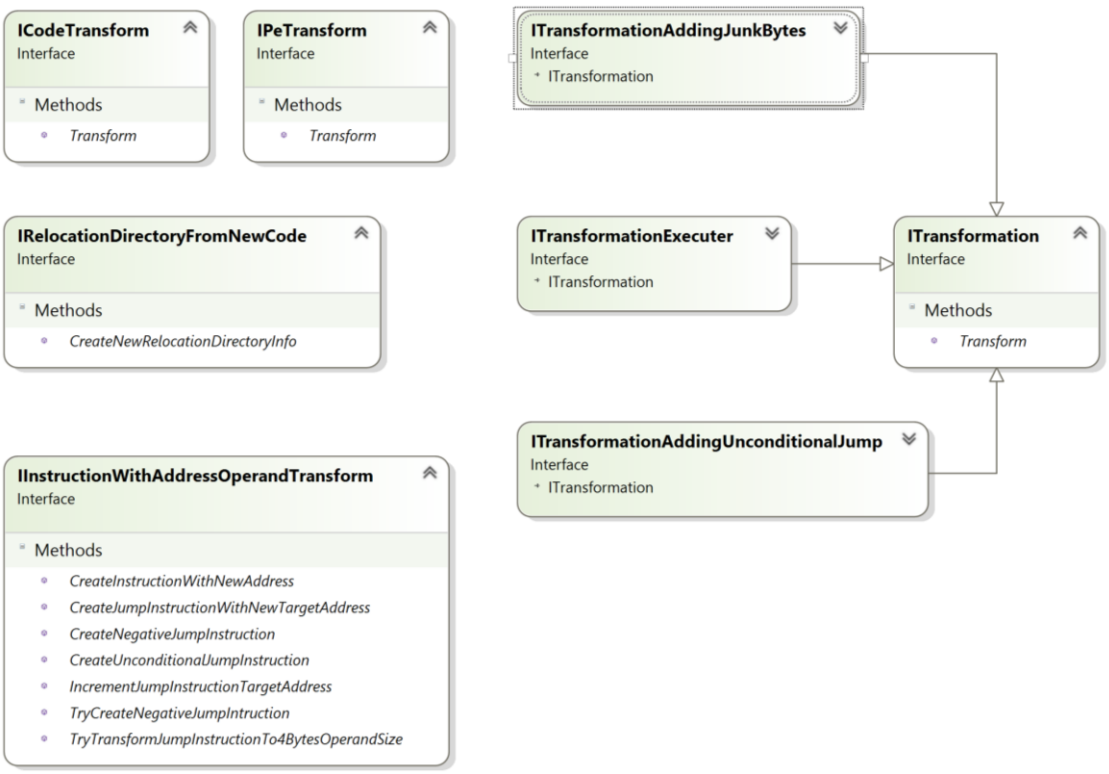
מחלקת בסיס לטרנספורמציות של קפיצות על תנאי והכנסת בתים משובשים. הטרנספורמציה מטפלת בהרחבת הוראות, יצירת מערך חדש של הוראות המכיל כתובות הוראות מעודכנות ואופרנדים מעודכנים ועדכון מידע נוסף בקוד.

  - `class TransformationAddingUnconditionalJump : TransformationBase,ITransformationAddingUnconditionalJump`

טרנספורמציה הממירה קפיצה על תנאי לקפיצה על תנאי הופכית וקפיצה ללא תנאי כפי שמודגם באיור 5.

  - `class TransformationExecuter : ITransformationExecuter`

מחלקה המבצעת רשימת טרנספורמציות על קוד. הרשימה המבוצעת היא של טרנספורמציות מסוג המרת קפיצה על תנאי והכנסת בתים משובשים.



איור 10: ממשקי טרנספורמציות במערכת על הקוד:

- *ITransformationExecuter*
- *ITransformationAddingJunkBytes*
- *ITransformationAddingUndconditionalJump*.

הממשקים האחרים מגדירים טרנספורמציה על תוכנית – *IPeTransform*, טרנספורמציה על קוד *ICodeTransform*, טרנספורמציה על טבלת *Relocation* – *IRelocationDirectoryFromNewCode*, וטרנספורמציה על הוראה – *IinstructionWithAddressOperandTransform*.

- `namespace ObfuscationTransform.Transformation.Junk`
  - `class TransformationAddingJunkBytes : TransformationBase, ITransformationAddingJunkBytes`

    טרנספורמציה המכניסה בתים משובשים בסוף בלוק בסיסי המקיים את התנאים המתאימים לכך.

  - `class JunkBytesProvider : IJunkBytesProvider`

    מספק בתים משובשים שהם חלקי הוראות שונות. ההוראות מהן נגזרים הבתים המשובשים הם אקראיים בכדי שלא יהיה ניתן לזהות תבנית שיבוש.

- `namespace ObfuscationTransform.Transformation.Factory`

  ה- namespace מאגד בתוכו מחלקות יצירה לכל המחלקות ב- namespace של `ObfuscationTransform.Transformation`

- `namespace ObfuscationTransform.PeExtensions`
  - `class ImageBaseRelocationSerializer : IImageBaseRelocationSerializer`

    מבצע סריאליזציה של טבלת ה- relocation לרצף בתים.

  - `struct RelocationTypeOffset`

    מתאר כניסה בטבלת ה-Relocation. כניסה בנויה מסוג הכניסה, והיסט מתחילת קטע הקוד.

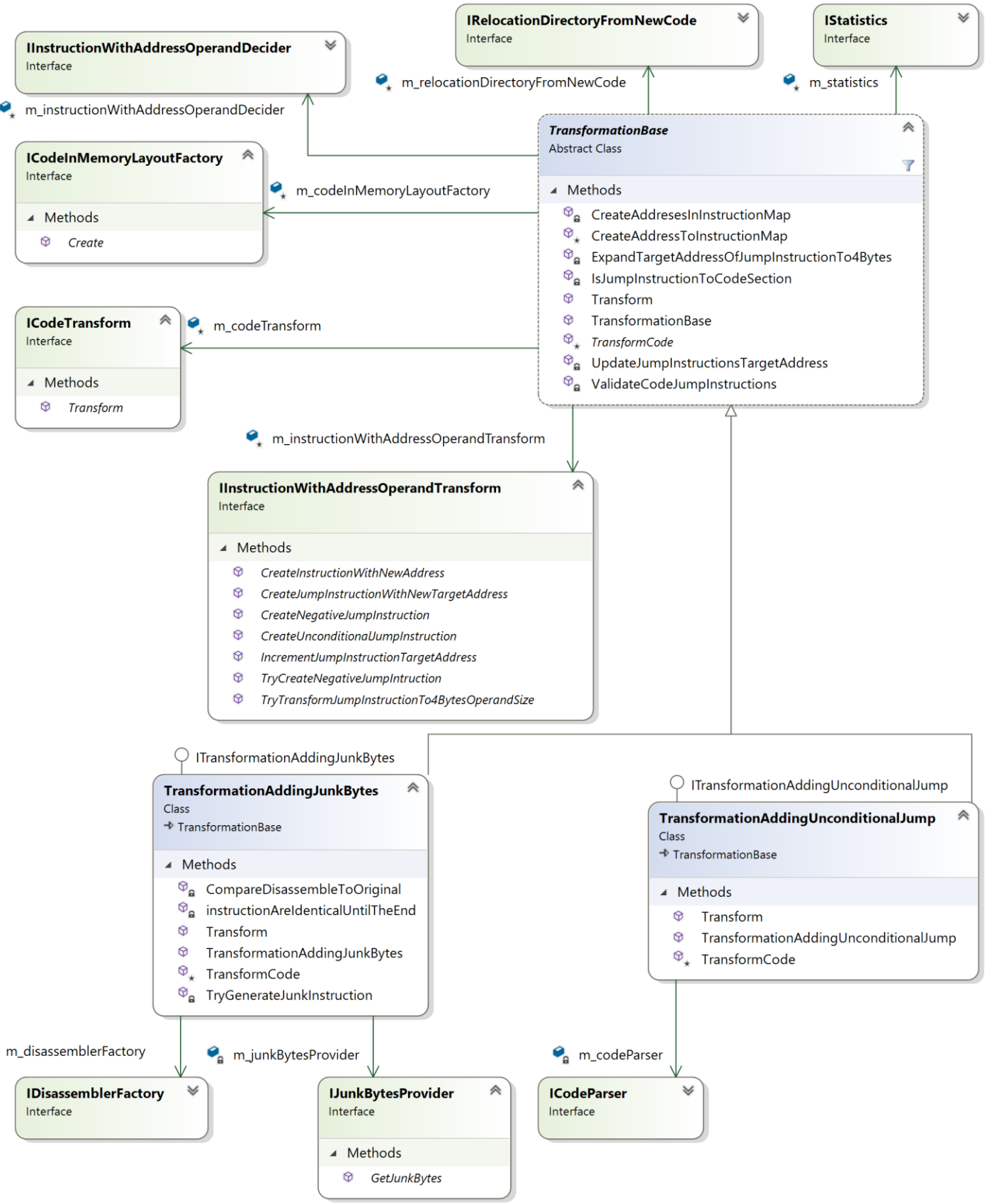
- `namespace ObfuscationTransform.Extensions`

- `static class InstructionExtensions`

  מרחיב פעולות על הוראה אסמבלר, כגון חילוץ ערך אופרנד לקפיצה באופן יחסי ואבסולוטי.

- `static class PeFileExtensions`

הרחבות של מחלקת `PeNet` המוגדרת בספריית צד שלישי *PeNet*. ההרחבות מאפשרות לחלץ מידע המיוצג בממשק `ICodeInMemoryLayout` . המידע המחולץ מקובץ מכיל פרטים כגון נקודת כניסה לתוכנית (Entry Point), מקטע הקוד, טבלת ה-relocation וכו'.



איור 11: מחלקות הטרנספורמציה *TransformationAddingJunkBytes,TransformationAddingUnconditionalJump* המממשות ממשקים מאיור 10.

באיור מתוארים הקשרים עם הממשקים של מחלקות העזר לטרנספורמציות המוזרקים בפונקציית הבנאי למחלקות הטרנספורמציה.
מחלקות הטרנספורמציה יורשות את מחלקת בסיס *TransformationBase* המממשת את רוב הפונקציונאליות הנדרשת מהטרנספורמציות. לכן, הקוד הממומש בטרנספורמציות עצמן קטן. אם תידרש טרנספורמציה נוספת, ניתן יהיה לרשת את *TransformationBase* המטפלת בהיבטים כלליים של טרנספורמציה ולהוסיף קוד המטפל בטרנספורמציה של פקודת אסמבלר. לדוגמא, לצורך ביצוע *profiling* על כל קריאה לפונקציה, תבוצע טרנספורמציה של ההוראה הראשונה בכל פונקציה כך שבמקומה יוכנסו רצף הוראות שיבצעו קריאה לפונקציית ה-*profiling* בנוסף להוראה הראשונה.

- `namespace ObfuscationTransform.Container`
  - `static class Container`

מחלקה המאפשרת את ה-Design pattern של Inversion of control והזרקת תלויות. המחלקה מאפשרת לרשום צמד של ממשק ומחלקה המממשת אותו (הנקראת שירות בעולם המושגים של ה-Container), ולצרוך שירות זה באופן מפורש או ע"י הזרקתו באופן אוטומטי בבנאי של מחלקה אחרת.
המחלקה מספקת אבסטרקציה מעל IoC Container של תשתית Unity של מייקרוסופט. למעשה ניתן להחליף את Unity בתשתית אחרת ללא כל השפעה על שאר התוכנית.
הערה: מומלץ לפנות למחלקה זו במקרים חריגים, מכיוון שתוכנה הבנויה נכון מזריקה את כל התלויות הנדרשות לפונקציות הבנאי. בד"כ קיימת מחלקה/ות עיקריות בודדות היוצרות עץ לוגי של מחלקות אחרות התלויות בהן (שבהן תלויות מחלקות אחרות, וכן הלאה). שימוש בContainer באופן ישיר ונרחב מהווה Anti-Pattern בשם Service Locator.

  - `class Parameter : IParameter`

מייצג פרמטר לפונקציית הבנאי של מחלקה (שירות) שצריך להיות מסופק באופן ישיר ע"י הקוד המשתמש ב-Container.

בפרויקט ObfuscationTransform.Console:

- `namespace ObfuscationTrasnform.Console`
  - `class Program`

מחלקה שהיא נקודת הכניסה של התוכנית, המפעילה את הספרייה הראשית `ObfuscationTrasnform`.
המחלקה מבצעת את הפעולות הבאות:

1. טוענת מקונפיגורציה את הקובץ עליו עושים טרנספורמציה.
2. מפענחת את הקוד ומידע על הקובץ לתוך מחלקה המיוצגת ע"י ממשק `ICode`.
3. קוראת למחלקה מתוך ObfuscationTransform המבצעת את הערפול (`ICodeObfuscator`) על הקוד.
4. כותבת את הקוד החדש לאחר טרנספורמציה לקובץ חדש (בעזרת `IPeTransform`).
5. כותבת את כל ההוראות של הקוד החדש והישן לקבצים המהווים פלט לתוכנית. הקבצים מאפשרים להשוות בין הקוד לפני הטרנספורמציה ואחרי.
6. כותבת את הסטטיסטיקות המיוצגות ע"י מחלקת `IStatistics` למסך.

4. פלט וקלט לתוכנה

הקלט הוא שם הקובץ הבינארי עליו יתבצע הערפול. הקלט מוגדר בקובץ קונפיגורציה נפרד.
הפלט המתקבל הוא שלושה קבצי טקסט:

- הוראות האסמבלר לפני הערפול.

- הוראות האסמבלר אחרי הערפול הכוללות פקודות לא חוקיות המודפסות כ-*Invalid*.
- הוראות האסמבלר המפוענחות מהתוכנית לאחר שעורפלה ומכילה שגיאות רבות.

פלט נוסף הוא סטטיסטיקות על ההמרה המוצגות על המסך תוך כדי ערפול.

5. מגבלות התוכנה

התוכנה נבחנה על מספר תוכניות שהודרו ע"י Visual Studio 2017 וכתובות ב-C++. תוכנות אחרות שעורפלו הפסיקו לעבוד (קרסו).
אולם, ניתן לעשות בתוכניות שהודרו שינויים מסוג כלשהוא שאינו כולל שימוש במחלקות מספריות צד שלישי, כך שהתכנית (שהודרה מחדש לאחר השינויים) תמשיך לעבוד לאחר הערפול.
התוכנה אינה מטפלת בקבצים כגון dll, מכיוון שאין טיפול לחלק המגדיר פונקציות/מחלקות הזמינות למשתמש בהם (exported/import).
כמו כן, ישנם היבטים נוספים (לא ידועים) שבהם נידרש לשנות חלקים אחרים בתוכנית מלבד חלק הוראות האסמבלר וחלקים אחרים שטופלו כבר, כגון כתובות המופיעות בחלק הטקסט המכילים קבועים (consts) בין היתר, טבלת relocation וכו'.
התוכנה לא מערפלת באופן תקין קבצים שבהם אין מקום פנוי בקטע הקוד להוספת פקודות חדשות. בד"כ תוכנית לא תופסת את מלוא המקום המוקצה לה בקובץ, ובנוסף מכילה מספר גבוהה של הוראות לא אפקטיביות (nop). אולם המקום הפנוי לא מספיק תמיד להכנסת הבתים שנוספים בטרנספורמציות. בכדי לאפשר זאת נדרש לחלק תוכנית לחלקים, לבצע ערפול, להרכיב את התוכנית מחדש ולעדכן את מקומות בהם מופיעות כתובות שהשתנו.

6. אתגרים

העבודה הציבה אתגרים רבים, כך שרוב הזמן והמאמץ בפיתוחה נצרכו באיתור תקלות - כאשר תוכנית בינארית לא עובדת, קשה לנתח את הסיבה לכך.
האתגר המשמעותי בעבודה הוא עבודה עם מ"ה (מערכת הפעלה) Windows שהיא מ"ה סגורה ללא קוד מקור – המערכת מבצעת בדיקות לא ידועות ומתועדות על מבנה התוכנית, וכאשר היא מבצעת את התוכנית, היא בודקת ומסיימת את התוכנית במקרים מסוימים שאינם מתועדים. בתהליך פיתוח שבו משנים באופן מכוון את התוכנית, כגון ערפול, יש צורך לבצע איטרציות של שינוי מדורג של התוכנית שיגרום בד"כ לקריסת התוכנית, מציאת הגורם, פתרון, שינוי הבא וכן הלאה.
בנוסף, מ"ה Windows 10 הוסיפה מנגנוני אבטחה שונים בכדי למנוע פעילות של תוכנות ששונו ע"י נוזקות, כך שהשינויים בתוכנה צריכים להתחשב בכך.

לצורך הערפול נדרש לשנות את מערך הפקודות בתוכנית, לדוגמא שינוי פקודה שהייתה בכתובת 0x102 לכתובת 0x106 נדרש בכדי לבצע ערפול. (למשל בכדי להכניס פקודה חדשה ובתים משובשים). תיעוד לשינוי כזה, המתייחס לפקודה ששינתה את כתובתה אינו נמצא בחיפוש באינטרנט.
התיעוד שמופיע בנושא שינוי תוכנית סופית בפורומים של האקרים ומפתחי תוכנה, מסביר כיצד משנים תוצר בינארי מוגמר ע"י דריסת פקודות אסמבלר קיימות, אבל ללא שינוי מקומם וכתובתם.
לעומת זאת, הערפול שמומש בעבודה זו מומש גם ע"י כותבי המאמר, אולם הכותבים ביצעו ערפול על מ"ה UNIX המספקת קוד פתוח, ומשתמשת בכלים שסופקו לה ע"י אינטל המאפשרים לפרק תוכנית לחלקים שונים ולאחדה מחדש, כך שניתן לבצע את הערפול הנדרש ללא צורך לעדכן את שאר התוכנית.

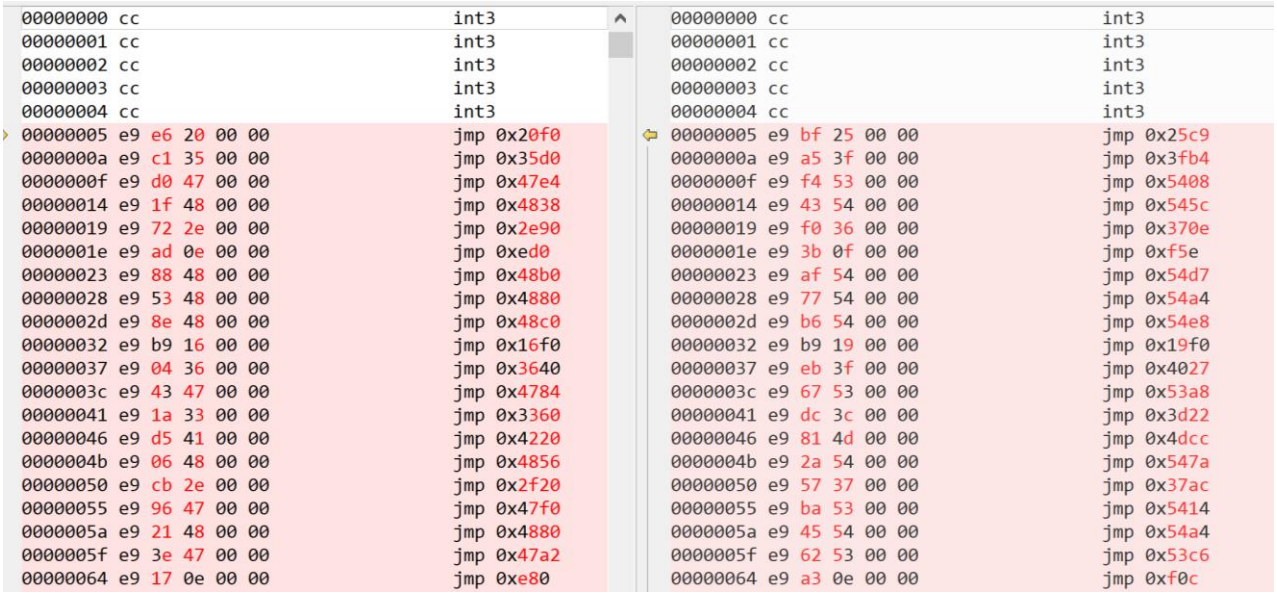
לכך מתלווה אתגר של איתור התקלות בתוכנה מעורפלת שאינה עובדת – שכן סיום התוכנית אינו מלווה במידע מספק על סיבת הסיום (לדוגמא stack trace ברמת פונקציות). איתור התקלה דורש מעקב ריצה של

קוד אסמבלר, שינוי כתובות בזיכרון, ערכי רגיסטרים, והשוואות ריצה של גרסאות שונות של אותה תוכנית.
כלומר – יש ללמוד את התנהגות מ"ה והתוכנית עליה התבצע הערפול בכדי לפתח את האלגוריתם.
דוגמא להתנהגות מסוימת של מ"ה Windows הגורמת לקריסת התוכנית, הוא השמה של ערך בזיכרון לערך שיגרום לסיום התוכנית מיד עם טעינתה. מתברר שהערך (security cookie) מצביע על תוכנית שאינה תקינה, כגון גלישת חלק הקוד מהמקום שהוגדר לו.

לכך נוספו אתגרים אחרים: היכולת לפענח קובץ בינארי של תוכנית P לפקודות אסמבלר, להכיר את מבנה הקובץ הבינארי על מ"ה ואיך הוא נטען לזיכרון התוכנית, למצוא אילו חלקים נוספים של התוכנה מלבד הקוד נדרשים לשינוי, להכיר את מגוון הפקודות והרגיסטרים במעבדים המודרניים, לייצר מיני מהדר שכותב פקודות אסמבלר בתוך התוכנית המקורית וכו'.

7. דוגמאות הרצה

נציג דוגמא של התוכנית Prime Number Calculation שעורפלה.
רקע: התוכנית מדפיסה את כל המספרים הראשוניים בין 2 ל-500000 במספר threads במקביל. מספר הthreads נקבע כמספר המעבדים.
התוכנית בנויה מ-13,119 הוראות אסמבלר שמתוכם רק 4314 הוראות אפקטיביות (לא nop), כך שנשאר מספיק מקום להוספת הבתים שבטרנספורמציה.
דוגמא להשוואת הפקודות הראשונות בתוכנית לפני ואחרי הערפול:

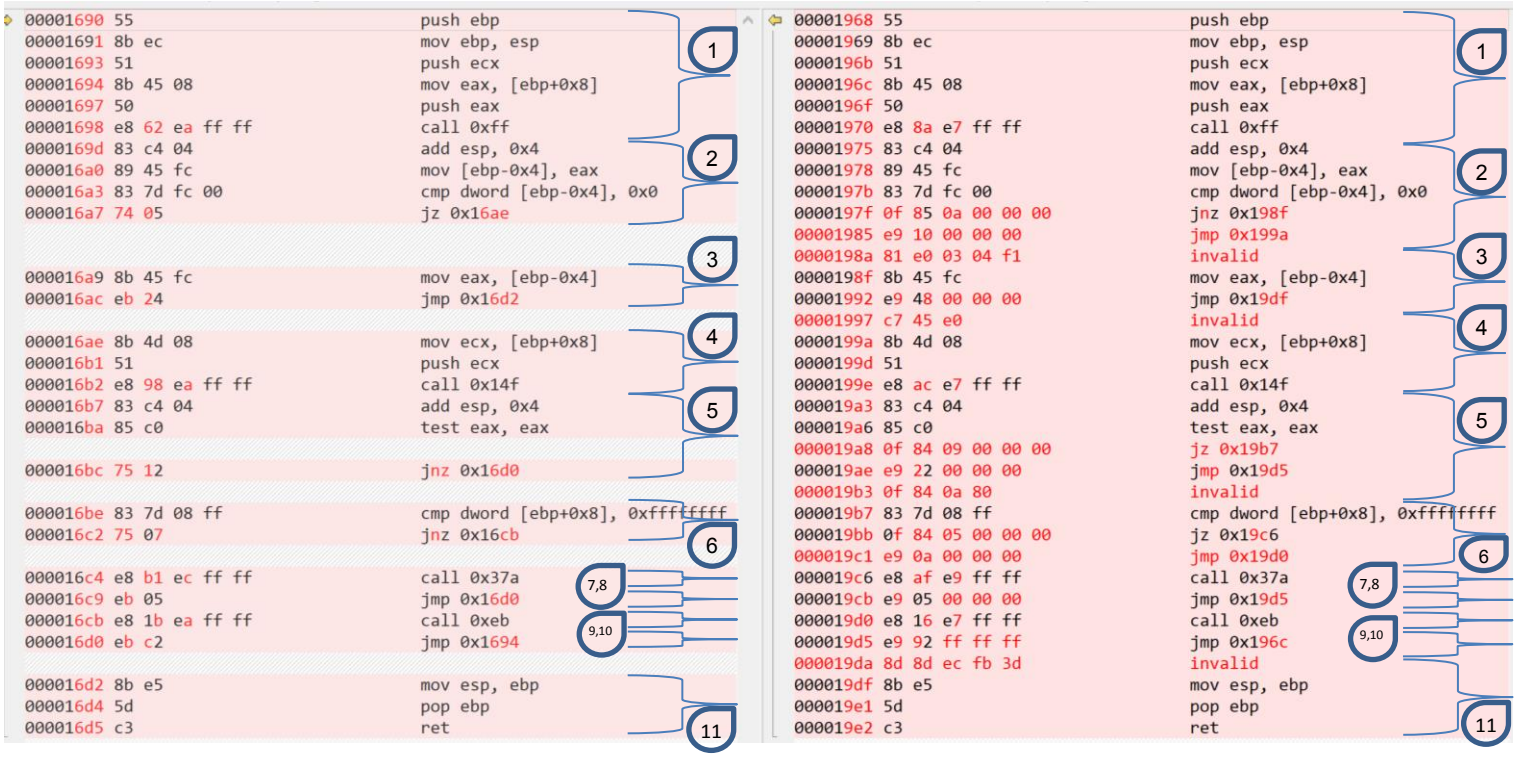


00000000	cc	int3	00000000	cc	int3
00000001	cc	int3	00000001	cc	int3
00000002	cc	int3	00000002	cc	int3
00000003	cc	int3	00000003	cc	int3
00000004	cc	int3	00000004	cc	int3
00000005	e9 e6 20 00 00	jmp 0x20f0	00000005	e9 bf 25 00 00	jmp 0x25c9
0000000a	e9 c1 35 00 00	jmp 0x35d0	0000000a	e9 a5 3f 00 00	jmp 0x3fb4
0000000f	e9 d0 47 00 00	jmp 0x47e4	0000000f	e9 f4 53 00 00	jmp 0x5408
00000014	e9 1f 48 00 00	jmp 0x4838	00000014	e9 43 54 00 00	jmp 0x545c
00000019	e9 72 2e 00 00	jmp 0x2e90	00000019	e9 f0 36 00 00	jmp 0x370e
0000001e	e9 ad 0e 00 00	jmp 0xed0	0000001e	e9 3b 0f 00 00	jmp 0xf5e
00000023	e9 88 48 00 00	jmp 0x48b0	00000023	e9 af 54 00 00	jmp 0x54d7
00000028	e9 53 48 00 00	jmp 0x4880	00000028	e9 77 54 00 00	jmp 0x54a4
0000002d	e9 8e 48 00 00	jmp 0x48c0	0000002d	e9 b6 54 00 00	jmp 0x54e8
00000032	e9 b9 16 00 00	jmp 0x16f0	00000032	e9 b9 19 00 00	jmp 0x19f0
00000037	e9 04 36 00 00	jmp 0x3640	00000037	e9 eb 3f 00 00	jmp 0x4027
0000003c	e9 43 47 00 00	jmp 0x4784	0000003c	e9 67 53 00 00	jmp 0x53a8
00000041	e9 1a 33 00 00	jmp 0x3360	00000041	e9 dc 3c 00 00	jmp 0x3d22
00000046	e9 d5 41 00 00	jmp 0x4220	00000046	e9 81 4d 00 00	jmp 0x4dcc
0000004b	e9 06 48 00 00	jmp 0x4856	0000004b	e9 2a 54 00 00	jmp 0x547a
00000050	e9 cb 2e 00 00	jmp 0x2f20	00000050	e9 57 37 00 00	jmp 0x37ac
00000055	e9 96 47 00 00	jmp 0x47f0	00000055	e9 ba 53 00 00	jmp 0x5414
0000005a	e9 21 48 00 00	jmp 0x4880	0000005a	e9 45 54 00 00	jmp 0x54a4
0000005f	e9 3e 47 00 00	jmp 0x47a2	0000005f	e9 62 53 00 00	jmp 0x53c6
00000064	e9 17 0e 00 00	jmp 0xe80	00000064	e9 a3 0e 00 00	jmp 0xf0c

איור *12*: השוואת הפקודות הראשונות בתוכנית לפני ואחרי ערפול. צד שמאל הם הפקודות לפני הערפול ובימין אחרי ערפול.
העמודה הראשונה מצד שמאל היא כתובת ההוראה *(offset)* כגון "0000000". בעמודה האמצעית הבתים המרכיבים את ההוראה ובעמודה הימנית ביותר תיאור ההוראה והאופרנדים. לדוגמא: jump 0x20f0.

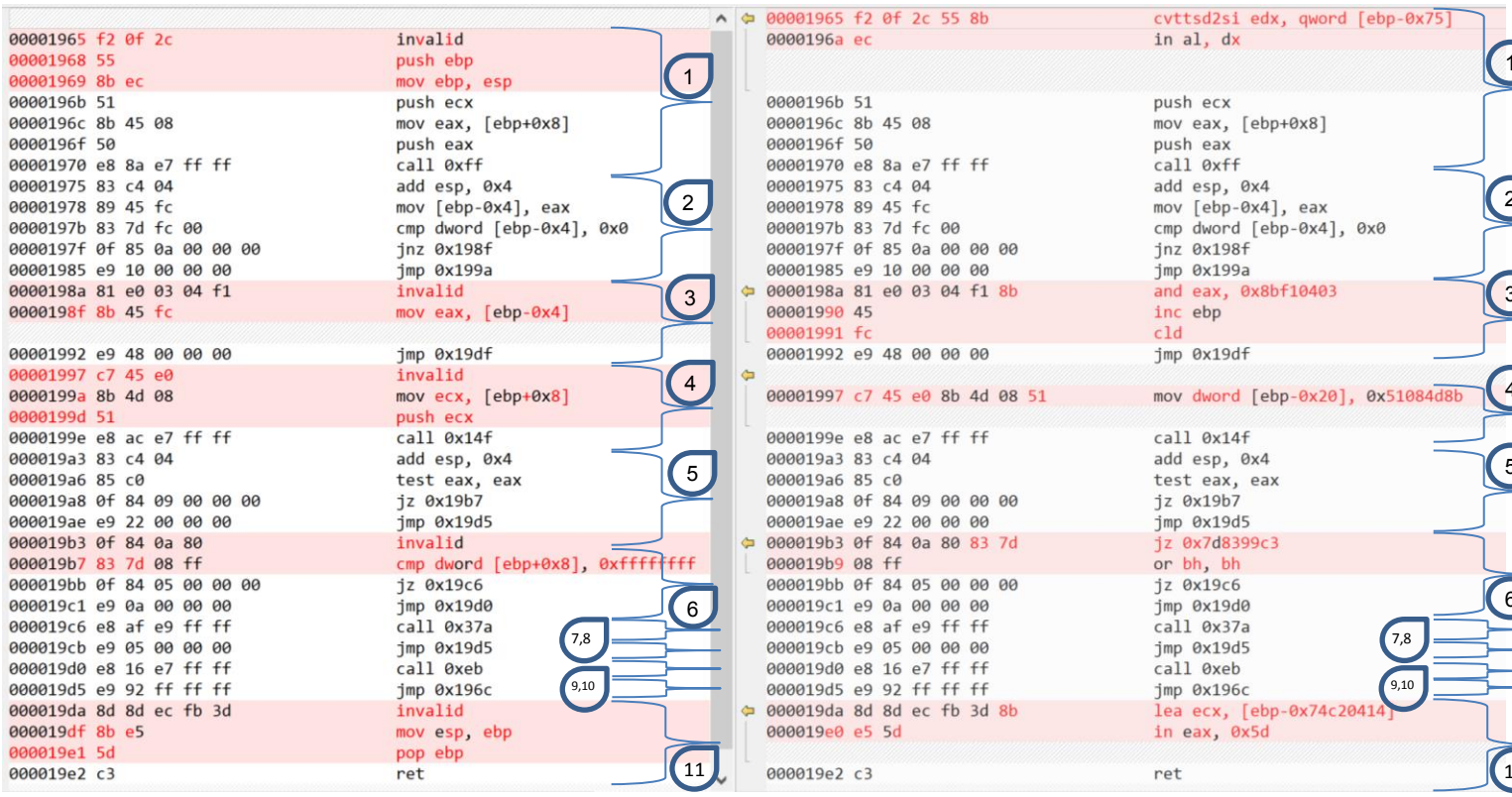
בהשוואה ניתן להבחין שהערך באופרנד הקפיצה של הוראת ה- jump גדל בקובץ המעורפל בערך של כ-0x500 . הערך בקובץ המעורפל מרמז על הכנסת בתים לפני ההוראה ביעד הקפיצה.

באיור 13, מודגמות שתי הטרנספורמציות על בלוקים בסיסיים המרכיבים פונקציה שלמה.



איור *13*: פונקציה לפני ואחרי ערפול. בצד שמאל מוצגות פקודות האסמבלר של פונקציה לפני הערפול ובצד ימין לאחר ערפול, כאשר הקטעים המסומנים מסמלים בלוקים בסיסיים. המימוש של איור מספר 4 מתבטא בטרנספורמציה על בלוקים מספר 2,5,6. לדוגמא בבלוק מספר 2, הוראת 'jz' (Jump if Zero) בצד שמאל עברה טרנספורמציה להוראה עם תנאי נגדי 'jnz' (Jump if Non Zero) בצד ימין בהתאם למוצג באיור. כתובת הקפיצה של 'jnz' היא לפקודה 'mov eax, [ebp-0x4]' הבאה אחרי הפקודה 'jz' בצד שמאל או אחרי 'invalid' בצד ימין. כתובת היעד של הפקודה 'jmp 0x199a' בבלוק 2 בצד ימין, היא אותה כתובת יעד של הפקודה 'jz' בבלוק 2 בצד השמאלי.
אחרי בלוקים בסיסיים מסוימים כגון בלוק 1, לא הוכנסו בתים משובשים (פקודת *'invalid'*), מכיוון ששיבוש ההוראות היה חוצה את גבולות הבלוק שבא אחריו.

באיור 14, מודגם השיבוש הנוצר בפענוח הוראות האסמבלר בעקבות הכנסת הבתים המשובשים המסומנים ע"י *invalid* בקוד המעורפל. פענוח הבתים המשובשים גורם לפענוח פקודות אסמבלר שגויות הנמשכות מספר הוראות לאחר מכן. (הן נבחרות כך שהשיבוש יהיה מספר הוראות מקסימלי אבל שאינו גולש לבלוק הבא).



איור *14*: שיבוש הוראות בבלוקים בסיסיים לאחר הכנסת בתים משובשים ('invalid'): בצד שמאל לפני בלוקים מספר 1,3,4,6,11 הוכנסו בתים משובשים הגורמים לשיבוש הגדול ביותר האפשרי במסגרת אותו בלוק בסיסי ובהסתמך על שבר ההוראה שהתקבל באופן אקראי.
בצד ימין מוצגים הבלוקים לאחר הפענוח השגוי. לדוגמא בצד ימין בבלוק מספר 3 שובשו שלושה הוראות.
שיבוש זה בא לידי ביטוי גם בתוכנה לניפוי שגיאות (debugger), כאשר הוא מציג את הוראות האסמבלר בכתובת של בלוק בסיסי עם בתים משובשים בתחילתו.

8. תוצאות ומסקנות

בכדי לבחון את יעילות הערפול, משתמשים במדד שיעור הטעיה "(confusion factor)" לזיהוי פקודות אסמבלר. שיעור ההטעיה נבדק על ידי אחוז הפקודות שפוענחו נכון, המוגדר כך: A מייצג את סה"כ מספר ההוראות ו-P את מספר ההוראות שפוענחו נכון. A-P הם ההוראות שלא פוענחו נכון.

$$CF = \frac{A - P}{A}$$

8.1.1 תוצאות

תוכנת הערפול קיבלה כקלט את התוכניות הבאות:

1. PrimeNumberCalculation.exe – תוכנה המחשבת את מספר הראשונים בטווח של 1-500,000 בתהליכונים (threads) מקבילים.
2. Strings.exe – תוכנית המדפיסה את כל המחרוזות בתוכנית מסוימת.
3. MaxTreads.exe – תוכנית המייצרת תהליכונים עד לסיום הזיכרון של התהליך.

התוצאות חושבו עבור 10 הרצות של כל תוכנית, כאשר התוצאות שונות בין הרצה להרצה בזכות הבחירה האקראית של הבתים המשובשים.

נציג את תוצאות ההרצה עבור PrimeNumberCalculation.exe כדוגמא מייצגת שאינה שונה מהממוצע באופן משמעותי (סטייה של עד אחוז): הכנסת בתים משובשים (TransformationAddingJunkBytes) בלבד מייצרת שיעור ההטעיה בממוצע הוא 10.85%, אולם אם מבצעים גם את הטרנספורמציה על קפיצות על תנאי (TransformationAddingUnconditionalJump) נעלה את שיעור ההטעיה ל- 18.10%בממוצע.

השיעור חושב מהוראות האפקטיביות (שאינן Non Operation-'nop'). בטרנספורמציה התווספו 3282 בתים בממוצע, ו-781 הוראות לא פוענחו נכונה מתוך 4314 הוראות בסך הכל (שאינן כוללות 'nop'). לתוכנית נוספו 838 הוראות, ו-1087 בתים משובשים בממוצע. מספר ההוראות הכולל בתוכנית לפני ערפול היה 13119, כלומר כמעט 9000 פקודות הם לא אפקטיביות.

מצ"ב טבלה המשווה בין התוכניות השונות, כאשר התוצאה חושבה כממוצע של עשר הרצות.

תוכנית	בתים שנוספו	בתים משובשים	הוראות משובשות	הוראות שנוספו	מספר הוראות כולל	מספר הוראות אפקטיביות	שיעור ההטעיה
PrimeNumberCalculation	8232	1087	781	838	13119	4314	18.10%
Strings	2999	940	667	740	13207	3807	17.52%
MaxThreads	2873	893	636	695	13010	3671	17.32%
ממוצע							**17.64%**

איור 15: תוצאות ערפול תוכניות שונות

8.1.2 מסקנות

האלגוריתם שמבצע את הטרנספורמציות בוצע מתחילתו ועד סופו, תוך כדי דריסת תוכנית קיימת, ללא הידורה מחדש.

התוכנית מהווה הוכחת יכולת למימוש האלגוריתם, אולם אינה עובדת על מגוון גדול של תכניות, מכיוון שישנם קשרים נוספים שלא טופלו בין חלקי התוכנית שאינם קוד לבין חלק הקוד. סיבה נוספת לאי יכולת לבצע ערפול על תכניות מסוימות היא המקום המועט שנשאר בחלק הקוד שהוא פנוי.

בהתאם לאיור 14, נדרש מקום פנוי רב: (בתוכנית PrimeNumberCalcaulation נוספו 8232 בתים בממוצע). בכדי להתגבר על בעיה זו ובעיות אחרות נדרש לפרק את התוכנית לחלקים (sections), להבין את הקשרים עם חלק הקוד, לבצע ערפול ולארוז את התוכנית מחדש.

כמו כן תוצאות שיעור ההטעיה עדיין נמוכות ביחס למאמר המציג הצלחה של כ-39% בממוצע. עובדה זו נובעת מאי מימוש אלגוריתם אחר במאמר שממיר את בקרת הזרימה לפונקציה וכך מאפשר ערפול גדול יותר.

9. סיכום

בדו"ח הוצג מימוש האלגוריתם מתחילתו ועד סופו, היבטים פרקטים של מימושו, ותוצאת ההרצה. השאלה האם הערפול ישים לכל תוכנית נשארת פתוחה עבור מ"ה Windows מכיוון שבהעדר קוד מקור שלה ותיעוד בנושא הקשרים המיוצרים בין הקטעים השונים של התוכנית לקוד, לא ניתן להגיע למסקנה בנושא. ניתן

להתרשם מהעובדה שחיפושים באינטרנט אינם מניבים תוצאות בנוגע לשינוי מערך הוראות האסמבלר של תוצר מוגמר, באופן היכול לרמוז על בעיות רבות שמונעות את הערפול באופן זה.

10. רשימת מקורות

[1] Cullen Linn, Saumya K. Debray: **Obfuscation of executable code to improve resistance to static disassembly.** Proceedings of ACM Conference on Computer and Communications Security 2003: 290-299

[2] Igor V. Popov, Saumya K. Debray, Gregory R. Andrews: **Binary Obfuscation Using Signals.** Proceedings of USENIX Security 2007

[3] Christopher Krügel, William K. Robertson, Fredrik Valeur, Giovanni Vigna: **Static Disassembly of Obfuscated Binaries.** Proceedings of USENIX Security Symposium2004: 255-270

[4] Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, Bart Preneel: **Program obfuscation: a quantitative approach.** Proceedings of the 2007 ACM workshop on Quality of, QoP 2007: 15-20

[5] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, Wenke Lee. **Impeding Malware Analysis Using Conditional Code Obfuscation.** Proceedings of NDSS 2008

[6] Dawn Xiaodong Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager,Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam,Prateek Saxena .**BitBlaze: A New Approach to Computer Security via Binary Analysis**. Proceedings of ICISS 2008: 1-25

[7] Andreas Moser, Christopher Kruegel, Engin Kirda: **Limits of Static Analysis for Malware Detection**. Proceedings of ACSAC 2007: 421-430

[8] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. **On the (im) possibility of obfuscating programs**. Journal of the ACM (JACM), 59(2):6, 2012.