האוניברסיטה הפתוחה
המחלקה למתמטיקה ולמדעי המחשב
החטיבה למדעי המחשב

עקרונות, טכניקות, ואלגוריתמים בתחום "סימן מים" בתוכנה

עבודה זו מוגשת כחלק מהדרישות לקבלת תואר "מוסמך (M.Sc.) במדעי המחשב" במחלקה למדעי המחשב, האוניברסיטה הפתוחה.

מגיש

יוחאי ליברידר

בהנחייתו של פרופ' אהוד גודס

מאי 2017

תוכן העניינים

תקציר 1

1. מטרת העבודה 2

2. מבוא 3

2.1 תיאור הבעיה 3

2.2 מטרות ושימושים עיקריים 6

2.3 תבניות בהטבעת סימן "מים בתוכנה" 6

2.4 מדדים להערכה של "סימן מים" בתוכנה 7

3. התקפות 9

3.1 התקפות חיבור (Additive Attacks) 11

3.2 התקפות חיסור (Subtractive Attacks) 12

3.3 התקפות עיוות (Distortive Attacks) 12

3.4 התקפת התאמה (Collusive Attacks) 13

3.5 התקפת פרוטוקול (Protocol Attacks) 13

4. גישות עיקריות ואלגוריתמים קיימים 15

4.1 הגישה הסטטית 15

4.2 הגישה הדינמית 20

5. כלים קיימים 22

5.1 SandMark 22

6. אלגוריתם CT (COLLBERG AND THOMBORSON) 25

6.1 תיאור האלגוריתם 28

6.1.1 מימוש בסיסי 28

6.1.2 כתיבת הערות או סימון (annotation) 29

6.1.3 מעקב (tracing) 30

6.1.4 שיבוץ (embedding) 34

6.1.5 בניית הגרף 34
6.1.6 חילוץ (extraction) 39

6.2 שיפורים מוצעים ועתידיים לאלגוריתם 42
6.2.1 שיפור העמידות 42
6.2.2 הגדלת גודל טביעת האצבע 46

6.3 הערכה של האלגוריתם (שבוצעה על ידי Collberg ו-Thomborson) 51
6.3.1 חשאיות 51
6.3.2 יחס נתונים 57
6.3.3 כושר התאוששות ועמידות 60

7. דיון ומסקנות 62

8. תוצאות ומסקנות ממימוש האלגוריתם בפרויקט המסכם 63

9. סיכום 66

10. רשימת מקורות 67

11. נספחים 71

11.1 SandMark – מעקב (trace) 71

11.2 SandMark – הטבעת הגרף המייצג. 71

11.3 Rank1 / Unrank1 [32] 72

11.4 בעיית ההכרעה של המצביעים [13] 72

רשימת האיורים

איור 1 – דוגמה לתהליך הטבעת "סימן מים" בתוכנה 5
איור 2 - הטבעת "סימן מים" בתוכנה 10
איור 3 - התקפות חיבור (Additive Attacks) 11
איור 4 – התקפת חיסור (Subtractive Attacks) 12
איור 5 – התקפת עיוות (Distortive Attacks) 13
איור 6 – אלגוריתם SHKQ - דוגמה לזיהוי שכיחויות כ"סימן מים". 18
איור 7 – שלבים עיקריים באלגוריתם CT 26
איור 8 – דוגמא – מחלקה לפני ואחרי אלגוריתם CT בסיסי 27
איור 9 – דוגמא – רישום נקודות מעקב בזמן ריצה. 31
איור 10 – גרפי קידוד לסימן המים. 35
איור 11 – גרף PPT בעל 4 צמתים 38
איור 12 – התאמת הבנאי של מחלקת הבסיס ל-CT 40
איור 13 – אלגוריתם נאיבי לחילוץ סימן המים ב-CT 41
איור 14 – אלגוריתם נאיבי לחילוץ סימן מים המיוצג על ידי גרף מזיכרון הערימה 42
איור 15 – התקפות ערפול כנגד גרפים המייצגים סימן מים 44
איור 16 – שימוש במחלקות LinkedList ו-Event לבנית גרף 49
איור 17 – הגדרה – חשאיות מקומית 52
איור 18 – הגדרה – חשאיות סטנוגרפית 52
איור 19 – דיאגרמת שכיחויות עבור סימן מים 32 סיביות ב-CT 54
איור 20 – דיאגרמת הערכה של חשאיות סטנוגרפית 57
איור 21 – גודל הגרף בזמן הריצה כתלות בגודל טביעת האצבע, לפי סוג גרף 58
איור 22 – גודל הגרף (בקוד) כתלות בגודל טביעת האצבע, לפי סוג גרף 59
איור 23 – גודל גרף בסיס (בקוד) כתלות במספר הרכיבים אליהם פוצל, עבור סימן מים בגדלים שונים 60
איור 24 – גודל התכנית וגודל הערימה בזמן ריצה עבור גרפי בסיס מעגליים, עבור סימן מים בגדלים שונים 61
איור 25 – חילוץ מוצלח של סימן המים מהתוכנית Calculator לאחר הזנת הקלט הסודי 65

תקציר

בעולם האינטרנטי של ימינו ישנה חשיבות הולכת וגדלה לטיפול בנושא גניבת תוכנה. הקלות שבהורדת תוכנות לא מורשות מהרשת מאפשרת ליותר ויותר אנשים לעשות בהם שימוש (במודע או שלא במודע). הסכום המוערך של שווי תוכנות אלה עומד על 52 מיליארד דולר בשנת 2016 [23]. סקר של ה-BSA מ-2016 [23] מראה בנוסף גם על קשר הדוק בין תוכנה שאינה מורשית לתוכנות זדוניות ולמתקפות סייבר.

אחד האמצעים המוצעים להקטנת תופעה זו הינו באמצעות מנגנוני "הוכחת בעלות" (Ownership Authentication). תחום זה נחקר רבות ומאגד בתוכו שיטות ואלגוריתמים רבים, אחת השיטות המוצעות בתחום זה הינה הטבעת "סימן מים" בתוכנה (Watermark embedding).

הטבעת "סימן מים" בתוכנה לצורך אימות זיהוי היא מעין "טביעת אצבע" המשובצת בצורה נסתרת בקוד, ולמעשה משמשת כרישיון שימוש בתוכנה. רישיון זה ניתן לשלוף כאשר נדרש ולטעון על אותנטיות (או חוסר אותנטיות) של התוכנה.

עבודה זו תציג סקירה של טכנולוגיות "סימן מים" שונות, תעמוד על יתרונותיהם וחסרונותיהם ותסקור את עמידותם בפני התקפות שונות, בהמשך תסקור מספר כלים מסחריים ומחקריים הנמצאים בשימוש בתחום, סקירה זו משתמשת במאמרים שונים, אך בעיקר במאמרים [19] , [20], [21].

עיקר העבודה יופנה למאמר של Collberg and Thomborson (2007) [19], המציג אלגוריתם מלא להטבעת "סימן מים" דינמי בתוכנה - CT Algorithm, והממומש על ידי שימוש בגרף דינמי Dynamic Graph Watermarking (DGW). העבודה תציג את האלגוריתם, תדון בטכניקות לשיפור ההסתרה של טביעת האצבע (Stealth), כושר התאוששות (Resilience) ועמידותו בפני התקפות שונות ותציג ניתוח של יעילות האלגוריתם בהיבטים אלו.

בפרויקט המסכם שבצעתי מימשתי את אלגוריתם CT. המימוש בנה יישום כללי, מעין כלי תשתיתי, כך שבקלות ובצורה אוטומטית ניתן לבנות ולהטמיע, ולאחר מכן לחלץ את סימן המים ל/מכל תכנית מארחת, למעשה יצרתי כלי שימושי ליוצרי תוכנות בסביבת Microsoft .Net המאפשר להם להטמיע סימן מים, בתוכנית שלהם בצורה אוטומטית מלאה וללא כל התערבות ידנית בקוד שיצרו ועל סמך קלט סודי. כמו כן, מאפשר הכלי לחלץ את סימן המים מהתוכניות שבהם הוא הוטמע, גם כן בצורה אוטומטית.

1. מטרת העבודה

נושא זכויות יוצרים מעוגן בישראל ובעולם במספר רב של חוקים. בתחום התוכנה ישנן פרצות רבות אך ברור הוא כי סכומי הכסף שהתעשייה העולמית מאבדת בעקבות פירטיות תוכנה מגיעים לעשרות מיליארדי דולרים. קיים כאן גם אספקט נוסף – שימוש בתוכנה פיראטית עלול לחשוף את המשתמש לסיכונים נוספים הקשורים לאיומי אבטחה [23].

הרעיון של זיהוי בעלות של תוכנה אינו חדש ואף נמצא בשימוש נפוץ בתעשיות אחרות כמו צילום, מוסיקה וקולנוע.

קימות טכניקות רבות ומגוונות המתיימרות לטפל בנושא זה. הטבעת "סימן מים" בתוכנה הינה אחת הטכניקות, מטרתה העיקרית היא לא לנסות למנוע פירטיות מלהתרחש, אלא להוכיח שהיא אכן התרחשה.

עבודה זו סוקרת את נושא הטבעת "סימן מים" בתוכנה, ומתארת לפרטיו אלגוריתם עיקרי בתחום.

מבנה שאר העבודה הוא כך: פרק 2 הינו פרק המבוא בו אנו מציגים את תיאור הבעיה, מטרות ושימושים עיקריים ב"סימן מים" בתוכנה, תבניות בהטבעת "סימן מים" בתוכנה ומדדים להערכה של "סימן מים" בתוכנה, פרק 3 עוסק ומציג התקפות קימות ואפשריות על "סימן מים " בתוכנה, פרק 4 מציג סקירה של גישות עיקריות ואלגוריתמים קיימים בנושא. פרק 5 מציג מספר כלים קיימים בנושא. פרק 6 עוסק בצורה מורחבת באלגוריתם CT, הפרק כולל תיאור מפורט של האלגוריתם, שיפורים מוצעים ועתידיים עבורו והערכה שלו במספר היבטים, וכל זאת מניסויים שביצעו Collberg ו- Thomborson. פרק 7 דן במסקנות בנושא "סימן מים" בתוכנה. פרק 8 מציג את התוצאות והמסקנות שהתקבלו ממימוש אלגוריתם CT בפרויקט המסכם שבצעתי, ופרק 9 הינו פרק הסיכום.

2. מבוא

המונח "סימן מים" מוכר לנו כטכניקה המיושמת בשטרות כסף, ניירות ערך ובולים ואשר שימש ומשמש כאמצעי לזיהוי זיופים על ידי יצירת סימון שאינו נראה ממבט ראשון ויצירתו מצריכה טיפול בנייר לפני הדפסתו. סימני מים בנייר נוצרו לראשונה בבולוניה ב-1242 [22] ונמצאים בשימוש גם בימינו.

הטבעת "סימן מים" בתוכנה מנסה ליישם טכניקה זו לשם אותה המטרה בעולם התוכנה. התחום אליו משתייכות טכניקות אלו בתוכנה הוא תחום הסטנוגרפיה (Steganography) תחום זה עוסק בטכניקות של החבאת מסר סודי בתוך מסר מארח כאשר המטרה היא לאפשר לשני צדדים לתקשר בחשאיות ומבלי לעורר חשד מצד מצוטטים פוטנציאלים. הפרסום הראשון בנושא הטבעת "סימן מים" בתוכנה הופיע לראשונה ב-1996 במאמרם של Davidson and Myhrvold [6] (ואף נרשם כפטנט). בשנת 1999, Collberg ואחרים הציעו לראשונה אלגוריתם מפורט. מאז פורסמו מספר רב של מאמרים ואלגוריתמים בנושא, ופותחו מספר כלים מסחריים ולמטרות מחקר.

הערה: בעבודה זו, המונח "סימן מים" בתוכנה משמעו שיבוץ של "טביעת אצבע" בתוכנה, ולכן השימוש בשני המונחים בצורה תחליפית.

2.1 תיאור הבעיה

ניתן לתאר את בעיית הטבעת (או שיבוץ) "סימן מים" בתוכנה בצורה הבאה:

- P – הוא אוסף התוכניות המיועדות לסימון.
- W – הוא אוסף הסימונים.

א. אלגוריתם להטבעת "סימן מים" בתוכנה משבץ w (השייך ל-W) בתוכנית p (השייכת ל-P) כך שנוצרת תוכנית 'p.
ב. ניתן בצורה אמינה לאתר ולחלץ את w מ-'p גם אם 'p הייתה חשופה לשינויים בקוד כמו תרגום (Translation), מיטוב (Optimization) וערפול (Obfuscation).
ג. w – ניתנת להסתרה (Stealth).
ד. ל – w יש תכולת נתונים גבוהה (High Data Rate).
ה. הטבעת w ב-p אינה פוגעת בביצועים ו/או בנכונות של p.
ו. ל – w יש תכונה מתמטית המאפשרת לטעון שנוכחותה ב-'p אינה מקרית אלא מכוונת.

ובשפה תיאורית:

א. לפני שאליס מוכרת את התוכנית שלה p, היא משבצת בה מזהה ייחודי עבור בוב, כך שנוצרת תוכנית חדשה $P_{\text{id=Bob}}$:

$$P \xrightarrow{Bob,\ key} P_{\text{id=Bob}}$$

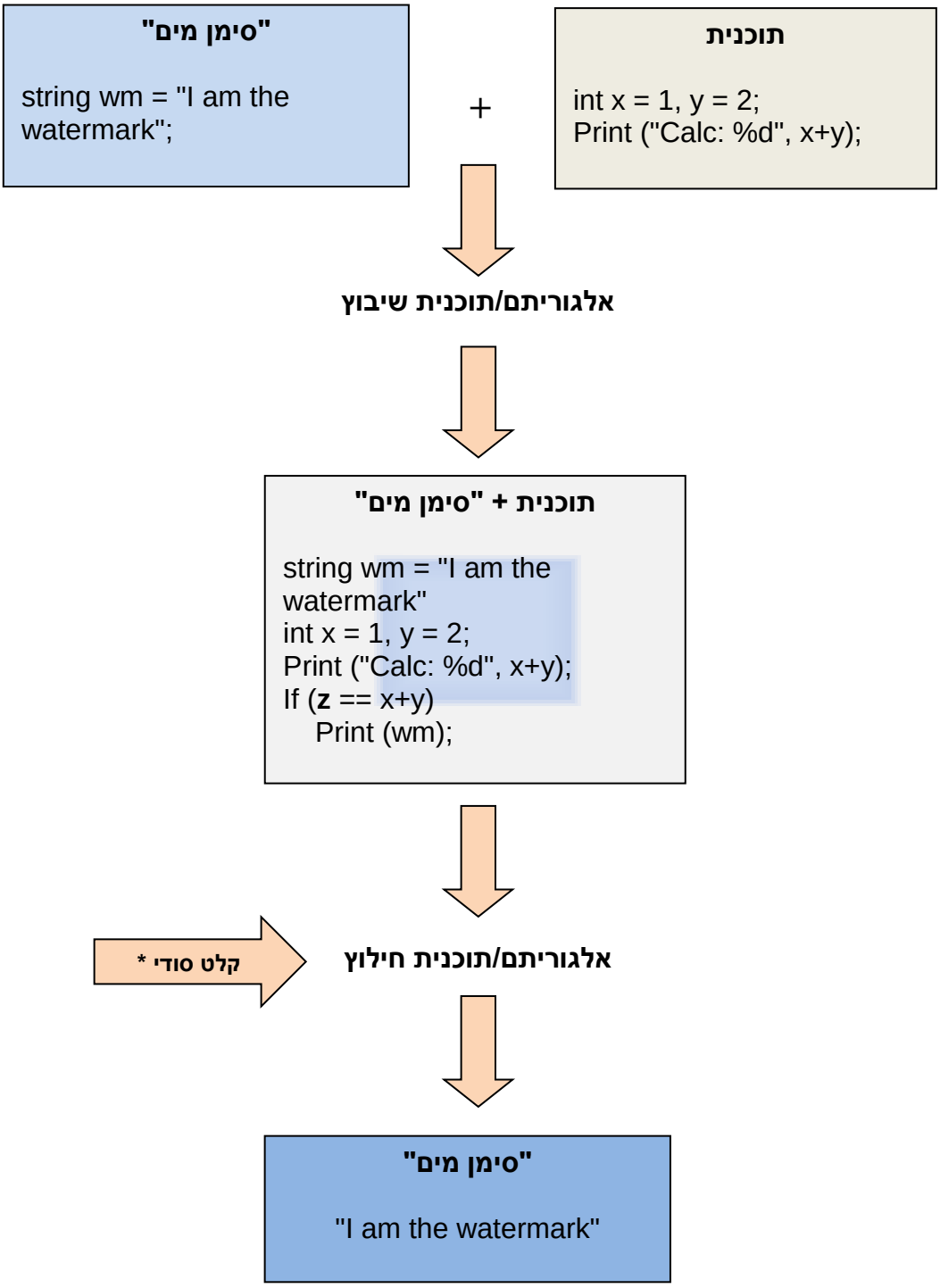
ב. כאשר אליס מאתרת עותק של p החשוד כמועתק היא מחלצת את הזהות של הרוכש המקורי מהתוכנית החשודה ויכולה להשתמש בזה כטיעון משפטי נגדו.

$$P_{\text{id=Bob}} \xrightarrow{key} Bob$$

ג. במשחק הזה מניחים שבוב יודע את אלגוריתם השיבוץ והחילוץ שבשימוש על ידי אליס אך אינו יודע את סימן המים וגם לא את המפתח הסודי. המטרה של בוב היא לייצר עותק $p'_{id=Bob}$ מ-$p_{id=Bob}$ מבלי לפגוע בישימותה של התכנית כך שאליס לא תוכל לחלץ ממנה את המזהה הייחודי שלו.

$$P_{\text{id=Bob}} \longrightarrow P'_{\text{id=Bob}} \xrightarrow{key} \not\longrightarrow Bob$$

שוב, חשוב להדגיש כי הטבעת "סימן מים" בתוכנה אינה מנסה למנוע פירטיות מלהתרחש, אלא להוכיח שהיא אכן התרחשה.



*המשתנה **Z** – הוא הקלט הסודי.

איור 1 – דוגמה לתהליך הטבעת "סימן מים" בתוכנה

2.2 מטרות ושימושים עיקריים

הטבעת "סימן מים" בתוכנה נבדלת מטכניקות אחרות להגנה מפני פירטיות במספר דרכים משמעותיות. ההבדל הראשון הוא, שלא כמו דונגלים (רכיבי חומרה נתיקים או קבועים המכילים אמצעי זיהוי ו/או אימות) וטכניקות אחרות היא אינה נסמכת על הימצאות של חומרה ייחודית ומאובטחת. הבדל נוסף, הטבעת "סימן מים" בתוכנה אינה נועדה למנוע פירטיות של התוכנה אלא לתת את היכולת לעקוב בדיעבד אחר פיראטיות בתוכנה. ולבסוף, שלא כמו מנגנוני רישוי תוכנה שלמעשה משולבים בתוך הקוד והתוכנית עצמה (וכך מספקים למעשה רמזים בעל ערך לתוקפים על מיקום הסימן), מחלץ "סימן המים"' בתוכנה הוא חיצוני לחלוטין ואינו מסופק עם התוכנית המסומנת.

להטבעת "סימן מים" בתוכנה יכולים להיות מספר שימושים או מטרות פונקציונליות אותם הוא נועד לשמש.

1. הטבעת "סימן מים" למניעה (Prevention) – מיועד למניעת (מניעה הרתעתית בלבד) שימוש לא מורשה בתוכנה.
2. הטבעת "סימן מים" להוכחת טענה (Assertion) – מיועד על מנת להוכיח בפומבי בעלות על התוכנה.
3. הטבעת "סימן מים" לצורך מתן רשות (Permission) – מיועד על מנת לאפשר שימוש בעותקים של התוכנה (רישיון שימוש).
4. הטבעת "סימן מים" לצורך חיוב (Affirmation) – מיועד להבטיח שמשתמש הקצה הוא זה שרכש את התוכנה.

2.3 תבניות בהטבעת סימן "מים בתוכנה"

רעיונית, הטבעת "סימן מים" בתוכנה מורכבת מהפונקציות הבאות:

embed (P, *w*, key) -> P_w
extract (P_w, key) -> *w*
recognize (P_w, *key*, *w*) -> [0.0, 1.0]

כאשר *embed* הופך תוכנית P לתוכנית P_w על ידי שיבוץ "טביעת אצבע" ("סימן מים") *w* תוך שימוש במפתח הסודי key. *extract* מחלצת את *w* מתוך P_w, ו-*recognize* נותנת את ההסתברות להימצאות W בתוכנית P_w ובהינתן המפתח הסודי key. הפונקציונליות של P ו-P_w חייבת להיות זהה, כלומר שיבוץ סימן מים חייב לשמר את הסמנטיקה של התוכנית. ישנם ואריאציות קטנות אבל רבות על

הגדרות אלה. למשל, *extract* יכולה להחזיר רשימה של "טביעות אצבע" ובנוסף את רמת הביטחון (confidence level) עבור כל "טביעת אצבע" שברשימה. ישנם מנגנונים אשר בהם לא נעשה שימוש במפתח (בכל אחת מהפונקציות בתהליך) וישנם כאלה שבהם המפתח מכיל עותק שלם של התוכנית הלא מסומנת.

מאחר שהטבעת "סימן מים" בתוכנה נעשה עבור תכניות שככל הנראה יותקפו, יש למדל גם את המטרות והיכולות של התוקף. התוקף ינסה לסכל את מטרות ההטבעה של "סימן המים" על ידי גילוי הפונקציות מהצורה הבאה:

detect (P_w) -> [0.0, 1.0]
attack (P_w) -> P'_w

הפונקציה הראשונה מדגימה את עקרון החשאיות (סטנוגרפיה), כאשר התוקף לא מסוגל לאתר האם הוטבע או לא הוטבע "סימן מים" בתוכנית. הפונקציה השנייה מדגימה את עקרון הזמינות, כאשר התוקף לא מסוגל להתקיף תוכנית שבה הוא חושד שמוטמע "סימן מים" כך ש"סימן המים" יהיה בלתי ניתן לחילוץ או בלתי ניתן לזיהוי (לאחר ההתקפה) למעט מתקפה שמשנה את התוכנית ופוגעת בה כך שהיא הופכת לבלתי שמישה.

נושא ההתקפות מובא בהרחבה בהמשך העבודה (פרק 3).

2.4 מדדים להערכה של "סימן מים" בתוכנה

ישנם שלושה מדדים מקובלים [19], אך הם אינם מדדים מדויקים, וזאת בהיעדרו של מודל מדויק ואנליטי של יכולות התוקף, ולכן שני המדדים הראשונים הינם מדדי איכות בלבד.

מערכת להטבעת "סימן מים" בתוכנה תהיה:

1. חשאית (stealthy) - אם היא עמידה לניסיונות התוקף לגלות את פונקציית הגילוי (detect).
2. מתאוששת מהר ועמידה (resilient) - אם היא עמידה לניסיונות התוקף לגלות את פונקציית ההתקפה (attack).
3. מדד יחס הנתונים (data rate) – מבטא את יעילות כמות המקום. למדד זה שני מדדי משנה – מדד יחס נתונים דינאמי ומדד יחס נתונים סטאטי. בשניהם המונה הוא מספר הביטים של "סימן המים" והמכנה מודד את התקורה (בבתים) שנוספה לתוכנית עקב הוספת סימן המים. במדד הסטאטי מודדים את הגידול בקובץ הרצה המהודר (ה-exe) ואילו במדד הדינאמי

מחשבים את הגידול בצריכת הזיכרון כפונקציה של מספר הביטים ב"סימן המים" בזמן ריצתה של התוכנית המסומנת.

מערכת (או אלגוריתם) אידיאלית להטבעת "סימן מים" בתוכנה תהיה בעלת מדד יחס נתונים מקסימלי, חשאית ובעלת כושר התאוששות. במציאות, לא ניתן למקסם אף אחד ממדדים אלו מבלי להזיק לאחד או יותר מהמדדים האחרים. לדוגמא, הגדלת יחס הנתונים נוטה להפחית את החשאיות מכיוון שהוספת קוד של "סימן המים" תיתן לתוקף יותר רמזים על פונקציית הגילוי (detect). דוגמא נוספת, הגדלת כושר ההתאוששות תקטין את החשאיות ואת יחס הנתונים, אם למשל ננסה להסתיר את "סימן המים" תוך שימוש במספר שיטות שונות מספר פעמים באותה התוכנית, כושר ההתאוששות יעלה מכיוון שעל התוקף להסיר או לקלקל את כל סימני המים מצד אחד, אך מצד שני יחס הנתונים ירד וכך גם ככל הנראה החשאיות מכיוון שהתוקף ידע שהתוכנית מסומנת אם הוא מצליח לאתר מי מ"סימני המים".

3. התקפות

תוכנות עלולות לארח קוד זדוני, בשל כך קיים מגוון רחב של טכנולוגיות להתקפות כנגד "סימן מים" המוטבע בתוכנית. בין הטכנולוגיות הללו ניתן למנות: הנדסה לאחור (reverse engineering) ניתוח קוד מקור/byte code, ניתוח פלטי תוכנה, ניתוח עקבות תוכנה (trace analysis), ניתוח המחסנית (stack) וניתוח הערימה (heap).

למעשה פונקציית התוקף detect היא גרסה גלויה (ללא מפתח) של פונקציית recognize (המחזירה הסתברות להמצאות w ב-P_W בהינתן המפתח הסודי, סעיף 3.3). היכולת לנטרל את הצורך במפתח משקפת את ההנחה הבסיסית בהוכחת אבטחה של מערכת: המגן לא יחשוף באופן ישיר את המפתח לתוקף.

מאחר ואנו מניחים כי פונקציית recognize אינה סודית, תוקף יכול לנסות פונקציות detect טריוויאליות ו/או רנדומלית כדי לבצע פריצה בכוח (brute force). פונקציה כזו היא סוג של פונקציית ()recognize שבה המפתח key והערכים של w נבחרים הסתברותית על סמך צמדים של key-w שמכיר התוקף מהעבר. מכאן אפשר להבין כי חוסר הידע של התוקף על המפתח ועל מרחב החתימות האפשרי הינו נדבך חשוב בהערכה של החשאיות.

פונקציית ה-detect המסוכנת ביותר, במבט מעיני המתגונן, היא אחת מדויקת בצורה מושלמת, ללא תוצאות שליליות שגויות (false negatives) וללא תוצאות חיוביות שגויות (false positives) ניתן לתארה בעזרת שתי המשוואות הבאות:

$$\forall P, w : detect(embed(P, w)) \geq 0.5 \qquad \text{(false negative)}$$

$$\forall X : detect(X) < 0.5 \quad \Rightarrow \quad \forall S, P, w : embed_S(P, W) \neq X \qquad \text{(false positive)}$$

בתיאוריה, ככל הנראה לא קיים מגלה "סימן מים" אידיאלי עבור כל אובייקט x סופי שכזה, וזאת עקב הטווח הלא מוגבל של המשתנים S, P, w במשוואת ה-false positive. אך במציאות לתוקף עלול להיות מידע מקדים על S, P, w שבהם נעשה שימוש למשל בעבר, אצל התוקף גם עלולה להימצא הידיעה באם תוכנית מסוימת X עברה הטבעה של סימן מים. תוקף כזה עלול לנצל את המידע שבידיו על מנת ליצור התקפה הרסנית.

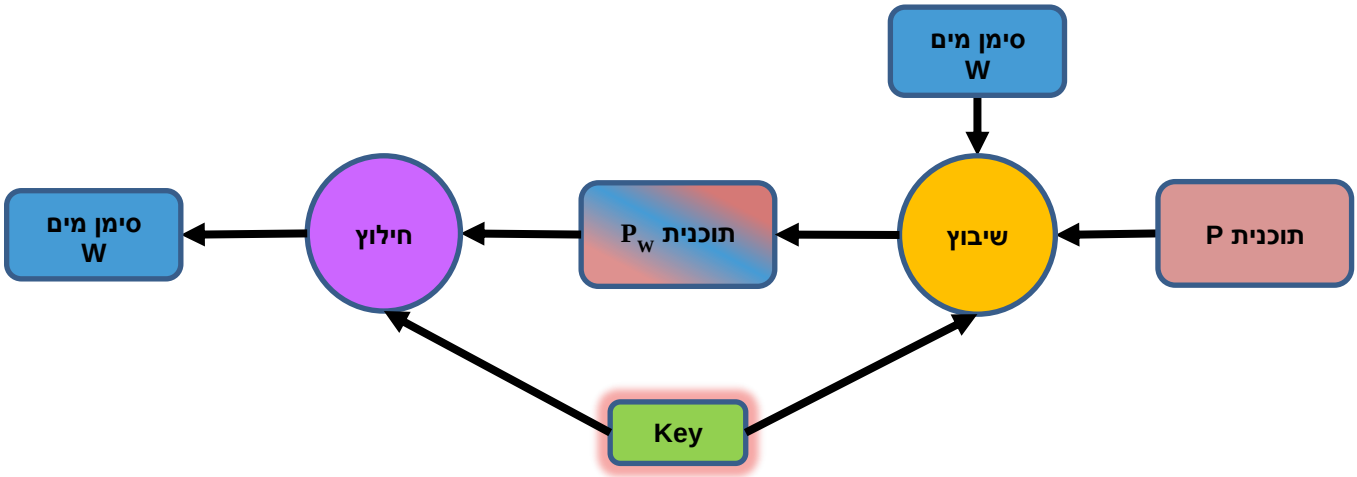
על מנת להעריך את העמידות של אלגוריתם לסימן מים, עלינו לדעת כמה הוא עמיד בפני התקפות מסוגים שונים, זהו כמובן טיעון לא מלא וזאת מכיוון שאין אנו יכולים לחזות את כל ההתקפות האפשריות על ידי כל סוגי התוקפים האפשריים, וכמובן שאיננו יכולים לצפות מכל מערכת אמיתית לספק הגנה "מושלמת" נגד התקפות שאינן טריוויאליות. Bender et al. [1996] [29] אפיין את

האבטחה של מערכות בתחום המדיה עם טביעת אצבע כך: *"... לכל השיטות המוצעות יש מגבלות. המטרה של השגת הגנה על כמות גדולה של מידע משובץ כנגד ניסיונות מכוונים להסרתה אינה ניתנת להשגה."*

להלן דוגמאות למספר התקפות אפשריות שונות, ואף ניתן לחשוב על התקפות המשלבות כמה מהם יחד, בכולם נניח את התרחיש הבא בו אליס חותמת תוכנית מארחת P עם סימן מים w ומפתח key ואז מוכרת את P_w לבוב:

$$P_w = embed\ (P, w, key)$$

לפני שבוב יוכל למכור מחדש את P_w עליו להבטיח ש"סימן המים" יהפוך לחסר תועלת, אחרת אליס תוכל להוכיח שזכויות היוצרים שלה בתוכנית זו הופרו.



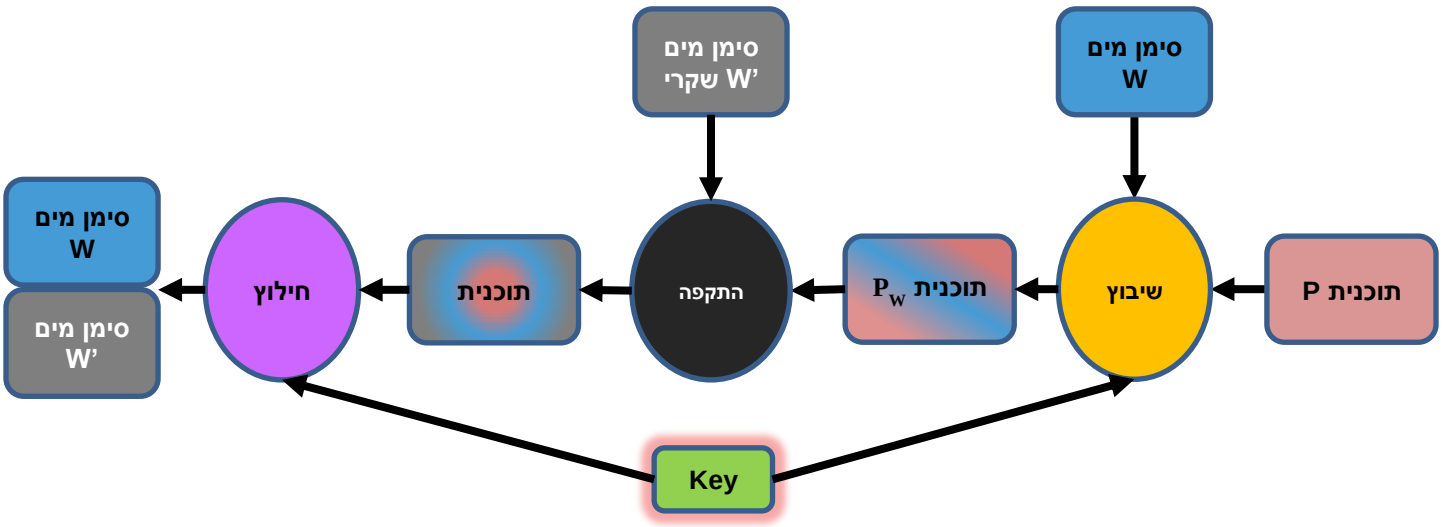
איור 2 - הטבעת "סימן מים" בתוכנה (2)

3.1 התקפות חיבור (Additive Attacks)

בוב יכול להגדיל את P על ידי הוספת "סימן מים" w משלו (או אפילו כמה כאלה) וכך טוען שסימן המים האותנטי הוא זה שהוא הכניס (המזויף). מתקפה יעילה מסוג זה היא כזאת שבה סימני המים המזויפים של בוב ניתנים להצגה בתוכנית ששונתה. בצורה פורמלית, בוב מכניס ואחר כך מחפש w' מזויף, ייתכן שמבלי לדעת את המפתח key של אליס עבור התוכנית P_w עם התכונות הבאות:

recognize (attack (P_w), key, w') > 0.5
recognize (attack (P_w), key, w) > 0.5

אפשר לראות כי הסימן w' של בוב, כמו גם הסימן w של אליס ניתנים לזיהוי בתוכנית שהותקפה (החדשה). אפשר לשים לב שעבור החילוץ של w' המפתח יכול להיות זהה לזה של אליס, אבל לא הכרחי.

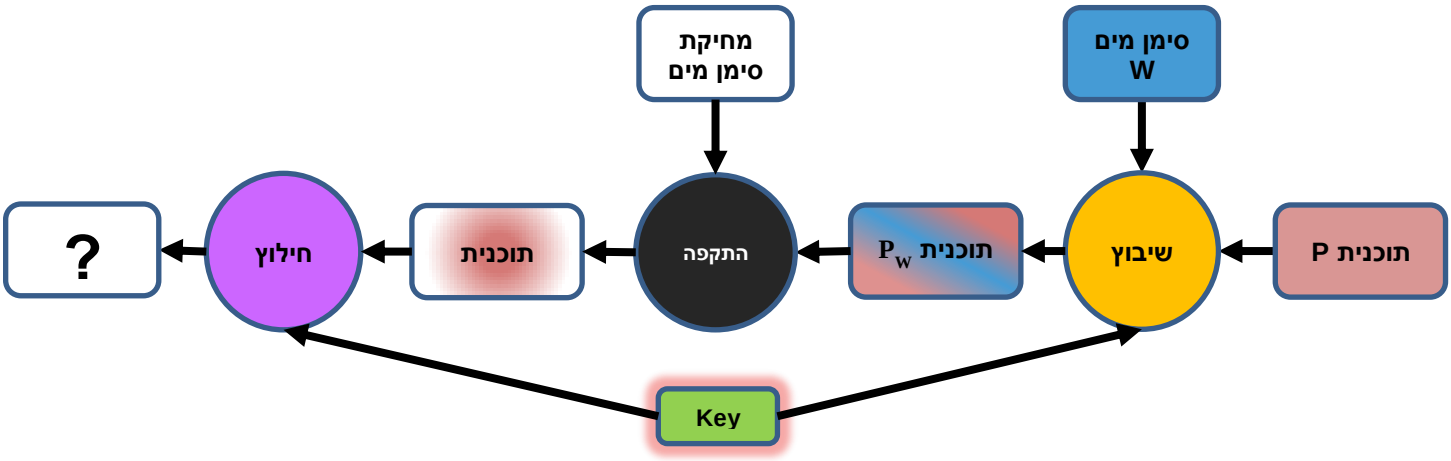


איור 3 - התקפות חיבור (Additive Attacks)

3.2 התקפות חיסור (Subtractive Attacks)

במידה ובוב מצליח לאתר את הנוכחות ואת המקום (המשוער) של "סימן המים" w הוא ינסה למחוק אותו מ-P_w. מתקפה אפקטיבית שכזו היא כזאת שהתוכנית שממנה נמחק "סימן המים" מכילה עדיין מספיק מתוכנה המקורי כך שהיא ממשיכה להיות בעלת ערך עבור בוב. ניסוח פורמלי של התקפה זו הוא כזה שבוב מחפש לבצע על התוכנית עם "סימן המים" טרנספורמציה משמרת סמנטיקה כך ש:

$$recognize\ (attack\ (P_w),\ key,\ w) < 0.5$$

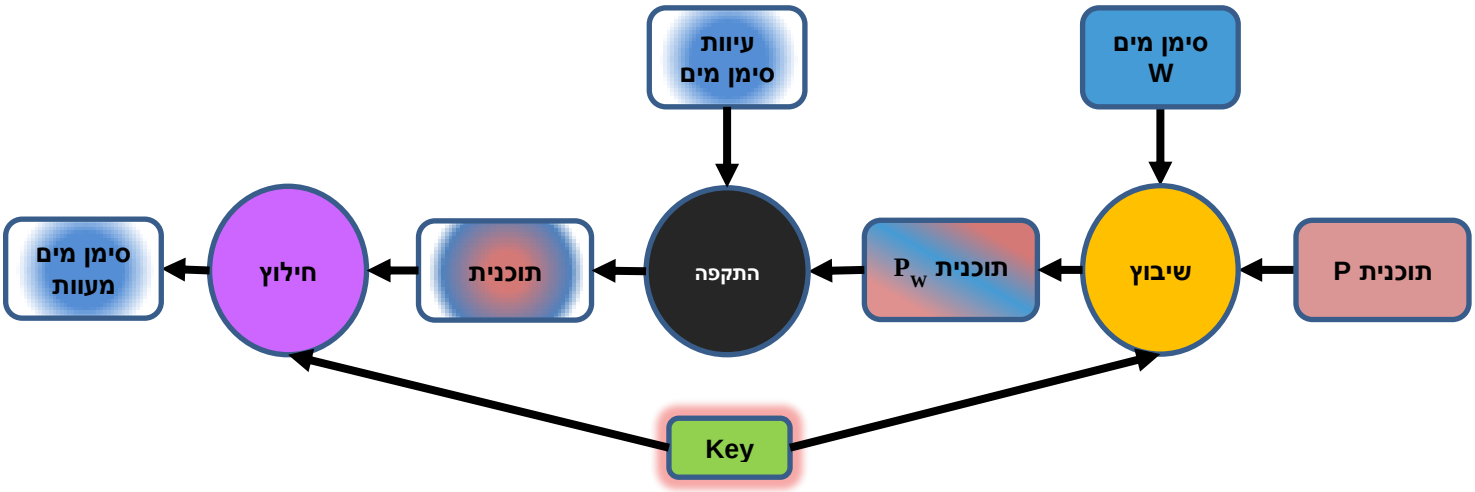


איור 4 – התקפת חיסור (Subtractive Attacks)

3.3 התקפות עיוות (Distortive Attacks)

אם בוב אינו יכול לאתר את w והוא מוכן לקבל ירידה מסוימת באיכות של P, הוא יכול להפעיל טרנספורמציה מעוותת (למשל ביצוע ערפול) על התוכנית, וכתוצאה מכך, גם על כל "סימן מים" ששובץ בתוכנית. מתקפת עיוות כזו הינה יעילה אם אליס לא תוכל עוד לחלץ את סימן המים שלה בתוכנית המעוותת, אבל לתוכנית זו יש עדיין ערך עבור בוב. למעשה יש דמיון רב בין התקפה זו להתקפת חיסור, ההבדל הוא שהתקפת חיסור איננה התקפה ניאותה (sound attack) בעוד שכאן ההתקפה היא ניאותה וזאת על סמך הגדרה שהציעו Madou et al. [2005] [30]. התקפה היא

ניאותה אם לאחריה ריצת התוכנית מובטחת בכל הרצה, לעומתה התקפה שאינה ניאותה תגרום לתוכנית שהותקפה להתנהג אחרת עבור קלטים מסוימים.



איור 5 – התקפת עיוות (Distortive Attacks)

3.4 התקפת התאמה (Collusive Attacks)

מאחר ובכל עותק של התוכנית שמוכרת אליס מוטמע "סימן מים" סודי שונה חשופות תכניות אלה למתקפה זו. הרעיון הוא שהתוקף משיג מספר עותקים שונים של התוכנית, הוא מניח שהם שונים בסימן המים, משווה ביניהם וכך יכול למצוא את המיקום של "סימן המים" בתוכנית. כתוצאה מכך הוא יכול להפעיל התקפות מסוג חיבור חיסור או עיוות.

3.5 התקפת פרוטוקול (Protocol Attacks)

במקרים בהם אליס לא מאבטחת בצורה טובה ומושלמת את מנגנוני השיבוץ או החילוץ או את המפתח, בוב יכול לנצל זאת לטובת התקפת פרוטוקול. ישנם מספר ואריאציות בהם ניתן לתקוף. בוב יכול לייצר מפתח מזויף 'k אשר מתיימר להוכיח את הנוכחות של הסימן המזויף שלו 'w בתוכנית שלא שונתה P_w וזאת על ידי שימוש בפונקציית החילוץ של אליס:

recognize (Pw, k', w') > 0.5

לחילופין, בוב יכול לייצר פונקציית חילוץ מזויפת שנועדה ל"הוכיח" את קיומו של "סימן המים" המזויף שלו 'w בתוכנית שלא שונתה P_w:

recognize' (Pw, key, w') > 0.5

כלומר בוב יטען שפונקציית החילוץ המזויפת היא האמתית וכי 'w הוא סימן המים האמתי, בהינתן המפתח key.

בוב גם יכול לשנות מספר אלמנטים בו זמנית:

extract' (Pw, k') → *w'*

התקפות פרוטוקול ניתן לשלב עם התקפות חיבור, חיסור או עיוות על התוכנית עצמה. לדוגמה, בוב יכול לפתח התקפת חיבור שעובדת בצירוף עם התקפת פרוטוקול בכדי לבלבל כאשר ישנה טענה משפטית לבעלות על התוכנית מצד אליס, כאשר אליס לא תוכל לדרוש את השימוש אך ורק בפונקציית החילוץ שלה:

extract' (attack (Pw), k') → w'

4. גישות עיקריות ואלגוריתמים קיימים

קימות שתי גישות עיקריות לביצוע הטבעת "סימן מים" בתוכנה (המבוססות על טכניקת ההטבעה והחילוץ של "סימן המים" ב\מהתוכנית) - גישה סטטית וגישה דינמית.

4.1 הגישה הסטטית

באלגוריתמים/שיטות אלו המסר מאוחסן בתוך קובץ ההרצה עצמו ואינו משתנה במהלך ההרצה. קימות מספר רב של שיטות ואלגוריתמים. סוג זה נפוץ מאוד מכיוון שהוא קל לבנייה ולשליפה (חילוץ), יחד עם זאת שיטה זו אינה יעילה וחשופה למניפולציות והתקפות. ניתן לחלק אלגוריתמים אלו לשני סוגים עיקריים – "סימן מים" סטטי בנתונים ו"סימן מים" סטטי בקוד.

סימן מים" סטטי בנתונים מאחסן את המסר בתוך נתוני התוכנית ויכול להתאכסן בכל מקום בתוך התוכנית כמו במשתנים. סימן מים" סטטי בקוד לעומתו מאוחסן במקטע הקוד של התוכנית ובדרך כלל תלוי ברצף של הוראות. דוגמא ל"סימן מים" כזה יכולה להיות הסתעפות במשפט switch / case כך ש-case מסוים (אשר הינו חסר משמעות לוגית בתוכנית) ייצג את "סימן המים".

בין האלגוריתמים הקיימים בגישה הסטטית ניתן למצוא את הסוגים הבאים:

א. Code Replacement

האלגוריתם מחליף קטע קוד או נתונים ידוע/ים מראש בסימן המים. אלגוריתם זה חשוף להתקפות מסוג התקפות התאמה (Collusive attacks) – התוקף משווה שתי תכניות או יותר בהם הוטמע "סימן המים" ומאתר את ההבדלים ביניהם, שם כנראה נמצא סימן המים.

Monden et al. [1] [2] [3] הציג טכניקה בשם MON עבור java (והממומשת בכלי שנקרא jmark). Fukushima and Sakurai [39] [40], שילבו את MON עם טכניקת האפלה (Obfuscation) בכדי להתגבר על התקפות התאמה.

Stern et al. [4] הציג טכניקה בשם ROW מבוססת spread spectrum ששימשה עבור "סימן מים" בתחום המולטימדיה. טכניקה זו עמידה בפני התקפות התאמה שכן סימן המים בה "מפוזר" בכל התוכנית.

חוקרים רבים נוספים הציעו מודלים נוספים ושיפורים למודלים קיימים [20].

ב. Code Re-Ordering

[5] Davidson and Myhrvold [1996b] שהיו מחלוצי התחום, הציעו ב-1996 אלגוריתם בשם DM המשבץ את סימן המים ע"י סידור מחדש בסיסי של בלוקים בתוכנית. בלוק תוכנה הוא אוסף של הוראות סדרתיות עם נקודת כניסה אחת ונקודת יציאה אחת. תחילה האלגוריתם בוחר קבוצה של בלוקים כאלה ומסדר אותם מחדש כך שייצגו "סימן מים" שנבחר. כמובן שהסידור מחדש שומר על זרימת הנתונים והפקודות המקורית כך שהפונקציונליות של התוכנה נשארת ללא שינוי. בדיקה של סידור הבלוקים הללו בתוכנית מאפשרת לחלץ את "סימן המים". אלגוריתם זה קל להתקפה על ידי התקפת עיוות, התוקף צריך רק "לסדר" מחדש את הבלוקים כך שלא ניתן יהיה לאתר את "סימן המים" המקורי.

[6] Shirali-Shahreza and Shirali-Shahreza , הציעו שיטה בה סימן המים מבוסס על סידור מחדש של פעולות במשוואות מתמטיות.

[7] Gong et al. הציע אלגוריתם ליצירת סימן מים על ידי סידור מחדש של מאגר הקבועים בקבצי המחלקות (Class files).

ג. Register Allocation-based Watermarking

[8] Qu and Potkonjak הציעו ב-1998 טכניקה לשיבוץ סימן המים על סמך מיקום הרגיסטרים של התוכנית תוך הסתמכות על בעיית צביעת גרפים. בעיית צביעת הגרפים מדברת על חלוקת צבעים מינימלית האפשרית לקדקודים בגרף כך שלא יהיו 2 קדקודים המחוברים ביניהם ושלהם אותו הצבע, כלומר הרעיון הוא לשבץ סימן מים בצורה אופטימלית בדומה או בתוך גרף הקצאות הרגיסטרים של התוכנית שמתבצע על ידי הקומפיילר (שיטה זו נמצאת בשימוש של קומפיילרים לצורך קביעת אופן השימוש ברגיסטרים בתוכנית ואופן חלוקתם עבור משתנים כך שניתן יהיה להשתמש בהם בו-זמנית – ומכאן שם האלגוריתם. הקדקודים הם המשתנים, וקשת מחברת 2 קדקודים אם ורק אם הם נדרשים לשימוש באותו הזמן. שימוש בצביעת גרפים מאפשר להפחית את מספר הרגיסטרים למינימום). אלגוריתם זה הוא אלגוריתם רגיש שכן אם אליס מסוגלת להטמיע סימן מים על ידי שינוי מבנה התוכנית, בוב יוכל בקלות להסיר את סימן המים על ידי שינויו שוב. בנוסף האלגוריתם סובל מתכולת הנתונים נמוכה.

[9] Myles and Collberg שיכללו את הטכניקה (QPS) לשלשות של קדקודים, אך תוצאות ניסויים חשפו חולשות רבות באלגוריתם, בעיקר חוסר האמינות וחוסר העקביות בזיהוי של "סימן המים" שהוטבע.

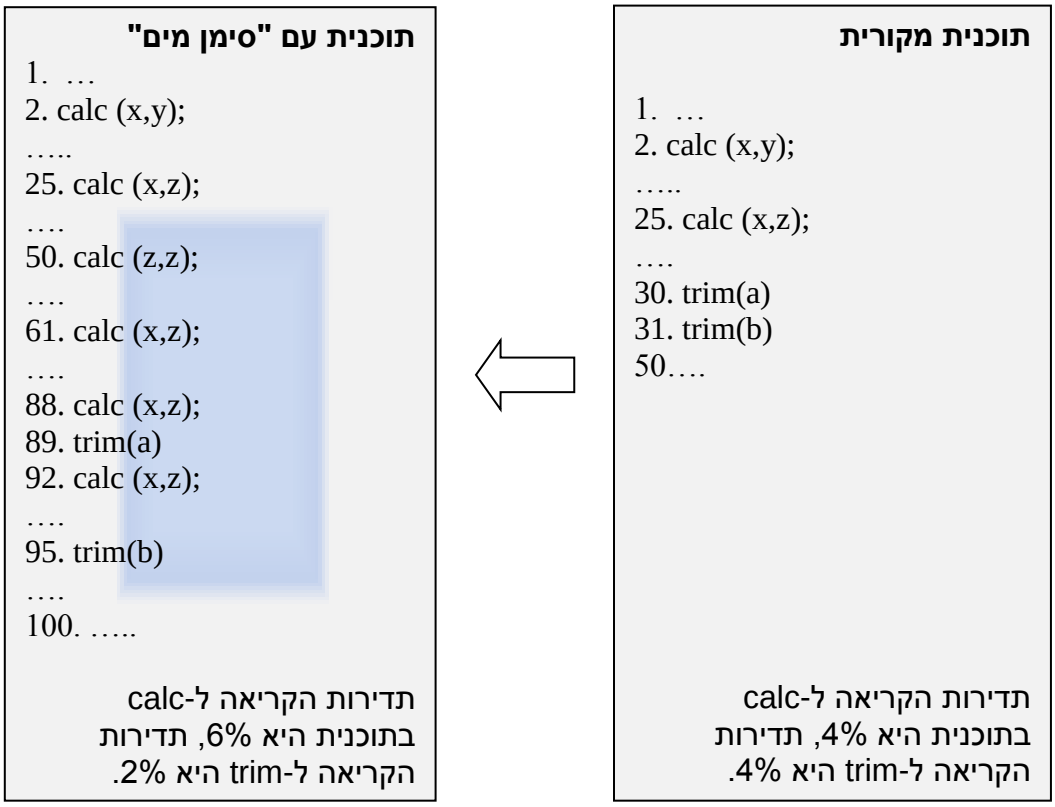
Zhu and Thomborson [10] הציעו ב-2006 את QPI המהווה שיפור נוסף, ובהמשך הוצעו שיפורים נוספים, כולל ורסיה המשלבת RSA public key encryption עם QPI (Jiang et al. [11] 2009)

ד. Spread spectrum

שיטה זו פותחה במקור עבור תעשיית המדיה הדיגיטלית. היא מייצגת נתונים במסמך כווקטור ומשנה כל יחידה בווקטור עם קורטוב של אקראיות – וזהו למעשה "סימן המים". ניתן לחלץ את "סימן המים" על ידי השוואה של הנתונים עם "סימן המים" שנוצר (בזמן היצירה). טכניקה זו נבדלת מטכניקות אחרות לשיבוץ סימן מים בכך שהיא מתבוננת על כל התוכנית כעל אובייקט אחד סטטיסטי ולא על מקבצים של הוראות, מסיבה זו טכניקה זו עמידה יותר למתקפות מסוג התאמה, שכן היא מקשה על איתור סימן המים.

Stern et al. [25] הציעו את אלגוריתם SHKQ אשר בו ההתייחסות לקוד היא לא כהוראות סדרתיות אלא כעצמים סטטיסטיים. סימן המים הוא בעצם התדירויות של קבוצות של הוראות עוקבות. האלגוריתם מחלץ קבוצות של הוראות מייצגות כאלה מהקוד לווקטור ואז מבצע עליהם את טכניקת ה-Spread spectrum לצורך שיבוץ "סימן המים" (איור 6 – אלגוריתם SHKQ - דוגמה לזיהוי שכיחויות כ"סימן מים".), בצורה זו הוא משנה את התדירויות של קבוצות של הוראות וקובע את סימן המים. האלגוריתם מומש הלכה למעשה בכלי SandMark (פרק 5.1).

Curran et al [26] הציעו ב-2004 אלגוריתם spread spectrum המשתמש במניה של עומק הקריאות לפונקציות בתוכנית כ"סימן מים". הם הציעו לייצר ווקטור בזמן ריצת התוכנית. הווקטור מייצג את עומקי הקריאות לפונקציות בנקודות מסוימות ובהינתן קלט ייחודי, לאחר מכן משנים את עומקי הקריאות כך שייצגו את "סימן המים" שרוצים לשבץ.



השכיחויות 6% ו-2% של המופעים להוראות שנבחרו בתוכנית עם "סימן המים" - הם "סימן המים".

איור 6 – אלגוריתם SHKQ - דוגמה לזיהוי שכיחויות כ"סימן מים".

ה. Graph Watermarking

טכניקות ואלגוריתמים אלו מסתמכים על העובדה שקוד המייצר גרפים קשה לניתוח וזאת עקב אפקט המצביעים ([12] Ghiya and Hendren) ואשר ידועה כבעיה לא כריעה ([13] Ramalingam) , בעיית ההכרעה מנסה להחליט האם 2 ביטויים מסוג L-VALUED (משתנים, בצד שמאל של סימן = בתוכנית) עלולים/יכולים (may) או חייבים (must) להכיל את אותו ערך בנקודה מסוימת בתוכנית.

בצורה לא פורמלית, נאמר שני ביטויים מסוג L-VALUE הם כינויים אחד לשני (alias) בנקודה ספציפית בזמן ריצת תוכנית אם שניהם מתייחסים (מצביעים) לאותו מיקום.

בבעיית ה-*may-alias* (ייתכן כינוי) מעוניינים לזהות כינויים שעלולים להתרחש בזמן חלק מהריצות של התוכנית. בעוד שבבעיית ה-*must-alias* (בהכרח כינוי) מעוניינים בזיהוי הכינויים בכל הריצות של

התוכנית. [13] Ramalingam הוכיח כי שתי בעיות אלו אינן כריעות עבור שפות עם משפטי תנאי (if) לולאות (loops), זיכרון דינאמי ומבני נתונים רקורסיביים, [42] Venkatesan הוכיח זאת גם למקרה שהתנאים כולם לא-דטרמיניסטיים ומבנה הנתונים היחיד הינו רשימות מקושרות. הרחבה בנושא ניתנת בנספח 11.4 .

[14] Collberg et al סווג את יצירת הגרפים הללו לשלושה סוגים: enumeration, radix ו-permutation.

[15].Venkatesan et al היה הראשון להציע את המודל הראשון לסימן מים מבוסס גרף סטטי – Graph Theoretic Watermarking (GTW) המקודד ערך בטופולגיה של גרף זרימת התוכנית, מודל זה נרשם כפטנט עבור חברת מייקרוסופט, מיושם בכלי SandMark (פרק 5.1) ואף נבדק כנגד מספר התקפות, בהם התגלה שהחולשה העיקרית של האלגוריתם הייתה ביכולתו להסתיר את סימן המים.

ו. Opaque Predicates

פרדיקטים אטומים, אלו פרדיקטים שתוצאתם ידועה אפריורית, ואשר קשה לתוכנה אוטומטית למצוא את ערכם, לכן קשה להסירם או לשנותם. זוהי אחת הטכניקות הפופולאריות ביותר גם בתחום ערפול תוכנה (Obfuscation).

[2][3] .Monden et al הציעו לשבץ "סימן מים" בתוך פונקציית דמה שלא מיועדת להרצה כלל והיא מוגנת על ידי פרדיקטים אטומים. האלגוריתם שהציעו הותאם ל-java וכולל שלושה שלבים:

1. הזרקת פונקציות דמה – בשלב זה מוסיפים לקוד את פונקציות הדמה ולאחר מכן מוסיפים קריאות לפונקציות אלו כאשר הם עטופות בפרדיקטים אטומים.

```
If (Opaque predicate) DummyCall ();
```

2. הידור (קומפילציה).
3. הזרקת "סימן מים" חבוי – כתיבת סדרת ביטים בתוך פונקציית הדמה (החלפת opcode ב-class file).

[16] Arboit פרסם אלגוריתם משופר שמאוחר יותר יושם בכלי SandMark (פרק 5.1) אך בניתוח שעשו [17] Myles and Collberg נמצא שהוא חשוף להתקפות התמרה משמרות סמנטיקה.

ז. Abstract Interpretation

שיטה זו משמשת בין היתר לאימות תוכנה. הרעיון הבסיסי הוא להחביא את סימן המים כך שהדרך היחידה לחלץ אותו היא באמצעות פירוש מופשט של הקוד, למעשה מדובר על "הרצה" שונה של הקוד ע"י מפרש שיודע לחלץ מהתוכנית את סימן המים .[18]Preda et al.

[27] Cousot and Cousot תיארו ב-2003 אלגוריתם לשיבוץ "סימן מים" כאשר הערכים משובצים במשתנים מסוג integer ייעודיים ומקומיים, גם כאן החילוץ נעשה על ידי מפרש ייחודי.

ח. Threading algorithm

[28] Nagra and Thomborson הציעו אלגוריתם ל"סימן מים" ומימשו אותו בשפת ג'אווה. האלגוריתם מנצל את העובדה שישנו אלמנט רנדומלי מובנה בריצת תהליכון (thread) בתוכנית מרובת תהליכונים (multithreaded program). מאחר שקשה מאוד לנתח תוכנית כזו האלגוריתם טוען להיות עמיד. בשלב ראשון, האלגוריתם יוצר מספר תהליכונים כך שמספר נתיבי ההרצה בתוכנית גדל (משמעותית) אך בד בבד מכניסים גם מנגנוני נעילה כך שהפונקציונליות המקורית של התוכנית תשמר. לאחר מכן משבצים נעילות כך שמבטיחים שרק תת קבוצה קטנה של נתיבי הרצה של התוכנית תוכל להתרחש בפועל בזמן ריצה. בשלב האחרון משבצים את "סימן המים" בנתיבים אלו.

4.2 הגישה הדינמית

באלגוריתמים/שיטות אלה המסר נוצר ונשלף בזמן ריצת קובץ ההרצה. הרעיון הוא להריץ את האפליקציה עם קלט ייחודי וסודי כך שבזמן ריצה ובהתאם לקלט האפליקציה תיצור טביעת אצבע ייחודית בזיכרון. שיטות אלו נפוצות הרבה פחות שכן הם קשות יותר למימוש מצד אחד אך הן עמידות יותר בפני התקפות מאידך.

ניתן למנות את השיטות הבאות:

א. Dynamic Easter Egg Fingerprints

הסוג הנפוץ ביותר והפשוט ביותר. חושף למשתמש את סימן המים על סמך קלט סודי.

דוגמאות למכביר ניתן למצוא ב-www.eeggs.com

דוגמא ל-Easter egg ב-Microsoft office word (www.eeggs.com):

1. *פתח מסמך Word חדש.*
2. *הקלד "=rand(200,99)" (ללא המרכאות)*
3. *לחץ Enter, והמתן מספר שניות.*
4. *נסה להקליד "=rand" עם ווריאציות שונות מ (1 ,1) עד (200 ,99). המספר הראשון מייצג את מספר החזרות והשני את מספר הפעמים בכל שורה.*

ב. Dynamic Execution Trace Fingerprints

בזמן ריצה "סימן המים" מוצג בהוראות או ב-trace של ריצת התוכנית.

ייתכן שנדרש קלט מיוחד. חילוץ סימן המים נעשה על ידי ניתוח כתובות ה-trace.

ג. Dynamic Data Structure Fingerprints

אלגוריתמים המתבססים על בניית מבני נתונים דינמיים ב-heap על סמך ריצת האפליקציה ועל סמך קלט סודי. האלגוריתם הבולט בנושא הוא – Dynamic Graph Watermarking (DGW) , CT algorithm של Collberg and Thomborson [19], עליו נדון בהרחבה בהמשך העבודה.

5. כלים קיימים

בשוק קיימים מספר פתרונות תוכנה התומכים באלגוריתמים להטבעת "סימן מים" בתוכנה, אנו נסקור רק את הכלי הראשון (SandMark) ברשימה זו:

- **Sandmark** – פיתוח למטרות מחקר שבוצע באוניברסיטת אריזונה [Coolberg Townsend et al. 2001], עבור אפליקציות ג'אווה ותומך ב-Software Watermarking ב- Obfuscation ובטכניקות נוספות. פיתוח המוצר הופסק בשנת 2004.
- **Allatori** – http://www.allatori.com, כלי מסחרי למטרת Obfuscation תומך גם ב-Watermarking, שחררו לאחרונה את גרסה 6.1.
- **DashO** – https://www.preemptive.com/products/dasho/features , פתרון אבטחה עבור אפליקציות ג'אווה המכיל פונקציות Watermarking.

SandMark 5.1

SandMark הינו כלי שפותח באוניברסיטת אריזונה ושמטרתו המוצהרת היא מחקר בהגנה של תוכנה על תוכנה [21]. הכלי שפותח בג'אווה, מודד את האפקטיביות של שיטות שונות מבוססות תוכנה להגנה מפני פיראטיות תוכנה, חבלה בתוכנה והנדסה לאחור. מטרת צוות הפיתוח הייתה לפתח טכניקות שיעזרו למשתמשים להעריך בצורה אמפירית אלו מהאלגוריתמים הקיימים הם בעלי ההשפעה הזניחה ביותר על ביצועי התוכנית מצד אחד, ומצד שני העמידים ביותר בפני התקפות.

הכלי והקוד זמינים להורדה ללא עלות באתר: http://sandmark.cs.arizona.edu, אך הוא אינו מתוחזק ולא משתחררות עבורו גרסאות חדשות.

מודל האיום ו-SandMark

על מנת להעריך את החוזק של טכניקות להגנת תוכנה יש צורך להגדיר בצורה טובה את מודל האיום. מודל זה מתאר למעשה את הכלים והטכניקות שסביר שתוקף יעשה בהם שימוש.

בהינתן מספיק זמן ומניע תוקף יכול לערער כל מודל הגנה קיים בתוכנה, לכן רוב התפיסות בהגנה מתמקדות בלהעלות את העלות של התקפה מוצלחת לגבוהה ככל שניתן. אלגוריתם מוצלח יהיה בנוסף גם כזה שעבור כל התקפה מוצלחת יגרום גם לירידה משמעותית בביצועים של האפליקציה המותקפת או שתגרום לה לא לעבוד כלל.

מפתחי SandMark רצו לפתח טכניקות הגנה בתוכנה ש:

- יעשו את העלות של מתקפה לגבוהה במידה כזו שהתוקף יוותר על התקיפה.
- לעשות את התקיפה לקשה מאוד.
- לגרום לאפליקציות שהותקפו בהצלחה להפוך ללא שימושיות על ידי הפיכתם ללא פונקציונליות, גדולות מדי או איטיות מדי.

מה נמצא ב-SandMark

SandMark מממש מגוון של אלגוריתמי ערפול ו-11 אלגוריתמים שונים ל"סימן מים" ביניהם אלגוריתם CT, בנוסף הוא מכיל תשתית לביצוע התקפות מסוג עיוות, חיסור, חיבור והתאמה.

על פיתוח הכלי שקדו 26 איש במשך שנתיים (50,000 שעות) ונכתבו בו 110,000 שורות קוד.

הכלי בנוי מ-8 חלקים:

1. מימוש של מספר אלגוריתמים ל"סימן מים".
2. כלים אנליזה סטטית של הקוד.
3. ניהול האובייקטים של התוכנית.
4. סט של שיטות למדידת תוכנה וסטטיסטיקות סטטיות על הקוד.
5. סט של כלים להתקפות ידניות על הקוד.
6. מספר "מנהלי ערפול" המשתמשים ביוריסטיקות שונות.
7. ממשק משתמש גרפי (GUI).
8. סט כלים לבדיקות ומדידות של האלגוריתמים ונכונותם.

המפתחים ציינו שמימוש אלגוריתמים ל"סימן מים" אינם טריוויאליים כלל.

הכלי תומך ב:

1. אלגוריתמים סטאטיים ל"סימן מים".
2. אלגוריתמים דינאמיים ל"סימן מים".
3. ערפול.
4. אופטימיזציה.

5. הצגת שיטות מדידה להנדסת תוכנה.
6. תצוגה של java bytecode.
7. כלי להשוואה של bytecode.

טכניקות ההערכה של SandMark לאלגוריתמי "סימן מים"

נכון להיום אין שיטה מקובלת למדידת והערכת אלגוריתם להגנה בתוכנה ורוב הספרות בתחום על "סימן מים" לא מנסה/מסוגלת להעריך תיאורטית ו/או מעשית אלגוריתמים אלו.

SandMark מנסה להעריך את האלגוריתמים על סמך הקריטריונים הבאים:

1. יחס נתונים (Data rate) – מה הוא היחס בין גודל "סימן המים" שניתן לשבץ בתוכנית לבין גודל בתוכנית.
2. תקורת השיבוץ (Embedding overhead) – כמה איטית יותר או גדולה יותר התוכנית המכילה את "סימן המים" לעומת התוכנית המקורית.
3. שיעורי החיובי השגוי (false positive rate) – מה ההסתברות לחלץ "סימן מים" תקני בהינתן עבורו קלט אקראי.

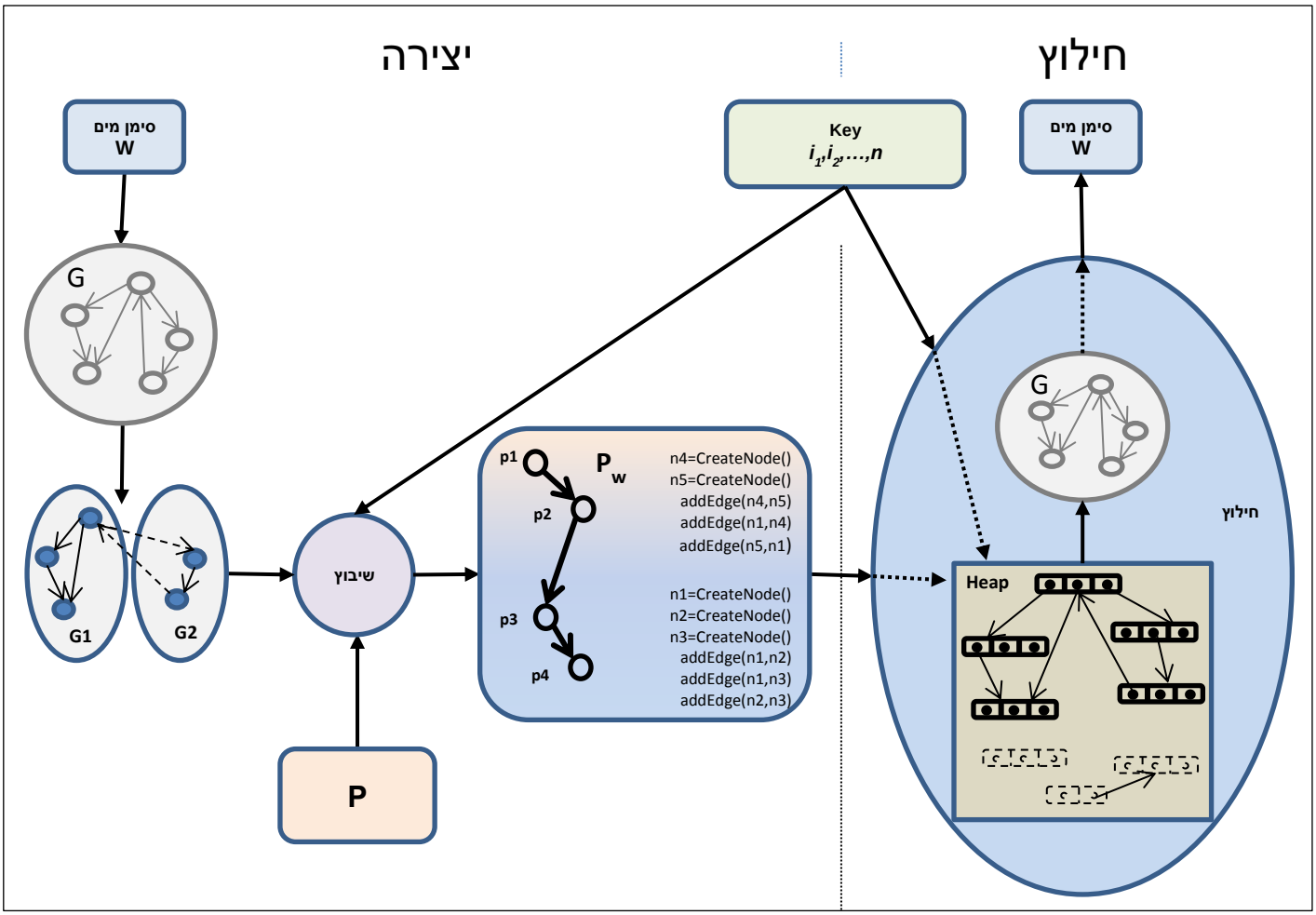
6. אלגוריתם CT (Collberg and Thomborson)

בשנת 2007 פרסמו Collberg and Thomborson מאמר המשך בו הם מרחיבים ומשכללים את האלגוריתם שלהם, הנקרא על שמם CT Algorithm [19]. הרעיון שעומד בבסיס האלגוריתם מסתמך על ההבחנה שניתוח של קוד שבונה מבנים של גרף בתוכנית באמצעות מצביעים הוא בעיה שקשה לנתחה ולכן יקשה על תוקפים (הרחבה בנספח 11.4). האלגוריתם בונה בזמן ריצה מבנה נתונים דינמי (גרף) המייצג את טביעת האצבע בהתאם לקלט (סודי) שנבחר מראש. בזמן הריצה, הקלט גורם לקטעי קוד שונים לרוץ וכך נבנית החתימה בזיכרון הערימה (heap), שאותה יש לחלץ במקרה הצורך. ראוי לציין שהאלגוריתם שתיארו הותאם למימוש בשפת java.

תרומתו של המאמר הייתה בהצגה ראשונה של אלגוריתם מלא לביצוע הטבעת "סימן מים" דינמי בתוכנה, בכך שהציג הצעות לשיפור העמידות שלו ואף ביצע הערכה של עמידות האלגוריתם להתקפות במספר אספקטים.

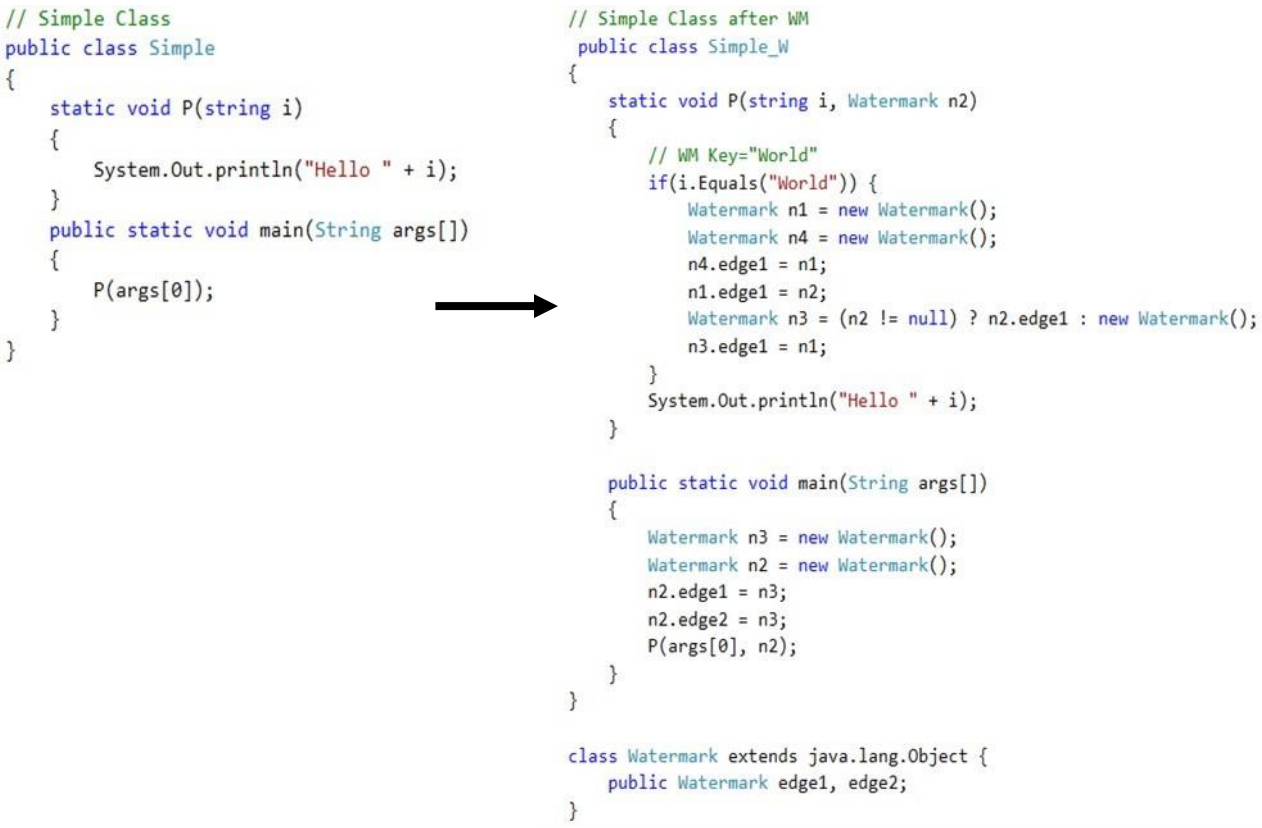
מספר היבטים הובילו להצעה של אלגוריתם זה. הראשון, העובדה שקוד המייצר גרפים קשה לניתוח וזאת עקב אפקט המצביעים (Ghiya and Hendren [12]) ואשר ידועה כבעיה לא כריעה ([13] Ramalingam) (הרחבה בנספח 11.4) יקשה על התוקף לנתח את התוכנית המסומנת או למצוא ולהרוס את סימן המים. ההיבט השני מדבר על כך שתוכניות מונחות עצמים מכילות בדרך כלל מספר גדול של טיפוסים ופעולות הנדרשות לצורך בניית הגרף כך שהגרף הנבנה בדרך כלל ילווה בקוד נוסף המלווה אותו. ההיבט השלישי גורס כי עבור גרף גדול (אשר ניתן יהיה לאתרו בקלות אם שובץ במקום אחד בקוד) ניתן לפצלו למספר רכיבים שרירותיים ולפזרם בכל התוכנית ובכך לשבץ בצורה חשאית בקוד "סימן מים" גדול. ההיבט הרביעי והאחרון אותו הזכירו טוען כי מאחר וסימן מים שכזה הוא מידע ולא קוד קל יותר להקשיח אותו. כך אם לגרף המייצג את "סימן המים" שנבחר יש תכונה ייחודית (למשל גרף מישורי, או גרף עם מרחק מסוים) ניתן להכניס קוד אשר בודק תכונה זו. זאת בניגוד לשיטות סטטיות להטבעת "סימן מים" בהם ההקשחה דורשת בדיקה של מקטע הקוד של התוכנית ואת זה קשה לבצע בצורה שאינה מעוררת חשד.

איור 7 – **שלבים עיקריים באלגוריתם CT**, מציג את השלבים העיקריים באלגוריתם.



איור 7 – שלבים עיקריים באלגוריתם CT

1. סימן המים W משובץ בטופולוגיה של גרף G.
2. הגרף מפוצל למספר תתי גרף…G1, G2,
3. עבור כל תת גרף נבנה קטע קוד המשובץ בתוכנית בנתיבים בהם היא עוברת בהינתן קלט סודי I_0, $I_1,\ldots$.
4. בזמן חילוץ סימן המים מריצים את התוכנית עם הקלט הסודי I_0, $I_1,\ldots$, הגרף המייצג את סימן המים נבנה בערימה (heap memory) , מחלצים את הגרף וחושפים את סימן המים.



```java
// Simple Class
public class Simple
{
    static void P(string i)
    {
        System.Out.println("Hello " + i);
    }
    public static void main(String args[])
    {
        P(args[0]);
    }
}
```

```java
// Simple Class after WM
 public class Simple_W
{
    static void P(string i, Watermark n2)
    {
        // WM Key="World"
        if(i.Equals("World")) {
            Watermark n1 = new Watermark();
            Watermark n4 = new Watermark();
            n4.edge1 = n1;
            n1.edge1 = n2;
            Watermark n3 = (n2 != null) ? n2.edge1 : new Watermark();
            n3.edge1 = n1;
        }
        System.Out.println("Hello " + i);
    }

    public static void main(String args[])
    {
        Watermark n3 = new Watermark();
        Watermark n2 = new Watermark();
        n2.edge1 = n3;
        n2.edge2 = n3;
        P(args[0], n2);
    }
}

class Watermark extends java.lang.Object {
    public Watermark edge1, edge2;
}
```

איור 8 – דוגמא – מחלקה לפני ואחרי אלגוריתם CT בסיסי

איור 8 מציג דוגמא פשוטה לכיצד תיראה התוכנית לאחר הטבעת "סימן המים". התוכנית המקורית Simple שונתה לתוכנית Simple_W כך שכאשר היא תורץ עם קלט סודי שהוא הארגומנט "World" יבנה בזיכרון הערימה הגרף המייצג את "סימן המים". במימוש טיפוסי Simple_W ו-Watermark היו עוברות ערפול בכדי למנוע מתקפת זיהוי תבניות.

6.1 תיאור האלגוריתם

6.1.1 מימוש בסיסי

האלגוריתם מניח קיום מפתח סודי k המשמש לחילוץ "סימן המים".

k הוא סדרה של קלטים $l_0, l_1, \ldots$ לתוכנית.

<u>שלב 1</u> – כתיבת הערות או סימון (annotation)

לפני שניתן לשבץ את "סימן המים" בתוכנית המשתמש חייב להוסיף נקודות סימון לתוכנית. נקודות אלו הינם קריאות לפרוצדורות (פונקציות) מהצורה הבאה:

```
mark ();
String S = … ;
Mark(S);
long L = …;
mark (L);
```

הקריאה ()mark למעשה אינה מבצעת דבר. קריאות אלו מהוות אך ורק אינדיקציות בקוד עבור מטביע "סימן המים" והן מסמנות נקודות פוטנציאליות בקוד התוכנית בהם יוטבעו חלקים מ"סימן המים". קריאות אלו יכולות לקבל משתנים כגון מחרוזות או מספרים אשר (בצורה ישירה או עקיפה) תלויים בקלט המשתמש. משתנים אלו עוזרים להבחין בנתיבים בקוד בהם התוכנית עוברת בהינתן סדרת קלטים על ידי המשתמש (אשר למעשה תפעיל את בניית הגרף המייצג את "סימן המים"), נתיבים אלו הם חלק תוצאתי של ריצה נורמלית של התוכנית (הלא מסומנת).

סדרת הקלטים יכולה להיות לחיצות מקלדת, תנועות ולחיצות עכבר/מסך מגע, קבצים, מחרוזות, וידאו, אודיו, נתונים ביומטריים, מסרי תקשורת וכו'.

<u>שלב 2</u> – מעקב (tracing)

לאחר שלב הסימון, מריצים את התוכנית המסומנת ל"ריצת מעקב". התוכנית רצה עם סדרת הקלטים הסודית המוזנת על ידי המשתמש. במהלך הריצה, תתבצע קריאה אחת או יותר של ()mark חלק מנקודות "הפגיעה" הללו יהיו למעשה המקומות בקוד אשר בהם מטמיע "סימן המים" יטמיע חלקים מסימן המים.

<u>שלב 3</u> – שיבוץ (embedding)

במהלך שלב זה, המשתמש מזין את "טביעת האצבע", למעשה מחרוזת או מספר שלם (integer) אם הזין מחרוזת, היא מומרת למספר שלם. ממספר זה מייצרים גרף מייצג, כך שטופולוגית הגרף מייצגת את המספר. עתה, קריאות ה- ()mark הרלוונטיות (מאלה שהיו נקודות "הפגיעה") יוחלפו כעת עם הקוד לבניית הגרף.

<u>שלב 4</u> – חילוץ (extraction)

במהלך שלב החילוץ, מריצים את התוכנית שוב עם סדרת הקלט הסודית. אותם מקומות בהם היו קריאות ()mark שבוקרו בשלב המעקב ייקראו שוב. אך עתה, מקומות אלה יכילו קוד לבניית הגרף המייצג את "טביעת האצבע". כאשר הוזן הקלט האחרון בסדרה למעשה הושלמה בניית הגרף בזיכרון הערימה. כעת נותר לבדוק את זיכרון הערימה אשר בו נמצא הגרף ולנסות לאתר אותו, לחלץ אותו ולבצע קידוד הפוך מגרף למספר אשר מדווח למשתמש.

שלב זה הוא השלב הקשה והמורכב ביותר למימוש בעיקר משום שיש צורך לחקור את זיכרון הערימה של תוכנית על ידי תוכנית אחרת ולנסות לחלץ ממנו גרף תקף, אשר ייתכן והוא הגרף המייצג את "טביעת האצבע" שהוטמעה.

6.1.2 כתיבת הערות או סימון (annotation)

אלגוריתם CT משתמש במבנה נתונים דינמי. משמע, שהקוד שמוטמע בתוכנית יראה בצורה הבאה:

```
Watermark n1 = new Watermark ();
Watermark n2 = new Watermark ();
n1.edge = n2;
….
```

ולכן, יש להעדיף סימון ב- ()mark מקומות בקוד אשר בהם ישנה יצירת והקצאת עצמים ואשר מתפעלים מצביעים ושתלויים ישירות בקלט המשתמש.

יש להימנע מסימון ב- ()mark מקומות בקוד אשר הם -נקודות חמות (מקומות בקוד אשר נמצא בהם ריכוז של הוראות, כלומר המקומות בקוד בהם נמצאת התוכנית ברוב הזמן) ומקומות שרצים בצורה אקראית (לא דטרמיניסטיים).

במילים אחרות, קריאות ל- ()mark צריכות להתווסף למקומות אשר בהם הקוד האחראי ליצירת "טביעת האצבע" יהיה נכון ומתאים, לא יפגע בביצועים ויורץ בצורה עקבית בכל ריצה וריצה בהתבסס על קלט המשתמש. לדוגמא, הקוד הבא אינו רצוי מאחר ש – ()Math.random עלולה לייצר ערכים שונים בריצות שונות של התוכנית:

```
If (Math.random < 0.5) {
…
mark ();
}
```

באופן דומה, תזמון של תהליכונים (threads), שעוני עצר (timers), פעילות ברשת, עיבוד מקבילי, עומס מעבד וכו' יכולים להשפיע על הסדר בו חלק מהמקומות יורץ, מקומות אלו כמובן אינם תקפים לסימון ויש להימנע מהם.

6.1.3 מעקב (tracing)

במימוש של שלב זה ניתן לעשות שימוש בספריות כמו Java JDI (Java Debugging Interface) כמו שנעשה בכלי SandMark, כך שניתן להריץ את התכנית כתוכנית משנה באופן ניפוי (debug mode) ולבצע את כל הפעולות הנדרשות בהתאם לאלגוריתם.

במהלך שלב זה אנו מעוניינים בהשגת העקבות (trace) של כל הביקורים בקריאות ()mark בזמן ריצת התכנית שמקבלת מהמשתמש את הקלט הסודי. אנו גם רוצים לדעת את הארגומנט שהועבר לקריאת ה- ()mark ואת מצב המחסנית (stack trace) בזמן הקריאה. בסוף שלב זה, תישאר בידינו רשימה של **נקודות המעקב** (TracePoints) שמייצגות את הקריאות השונות ל-()mark. כל נקודת מעקב מכילה שלשה פריטי מידע:

1. המיקום (בקוד) בו נמצאת קריאת ה- ()mark שנקראה.
2. הערך של הביטוי e שהמשתמש סיפק כקלט לקריאת mark(e) או Ø אם נקראה ()mark.
3. רשימה של הקריאות לפונקציות (במחסנית) דרכם התוכנית הגיעה לקריאת ()mark.

איור 9 מראה דוגמא לתוכנית בשפת java ורשימת נקודות המעקב הנוצרת בזמן ריצתה. הפונקציה actionPerformed נקראת בקיצור aP.

```
import java.awt.event.*;
import javax.swing.*;
public class Button implements ActionListener {
    static void P(int i) {
        L0 :mark(i);
    }
    static void Q(int i) {
        L1 :mark(i);
        if (i < 2) P(i);
    }
    public void actionPerformed(ActionEvent e) {
        L2 :mark();
        Q(3);
    }
     public static void main(String argv[]) {
         Q(1);
         Q(2);
         JFrame jw = new JFrame();
         JButton b = new JButton("w00t!");
         b.addActionListener(new Button());
         jw.getContentPane().add(b);
         jw.pack(); jw.show();
     }
}
```

#	Value	Method	Location	Thread	Stack
Ⓐ	1	Q	L_1	1	⟨main,Q⟩
Ⓑ	1	P	L_0	1	⟨main,Q,P⟩
Ⓒ	2	Q	L_1	1	⟨main,Q⟩
Ⓓ	0	aP	L_2	2	⟨aP⟩
Ⓔ	3	Q	L_1	2	⟨aP,Q⟩

איור 9 – דוגמא – רישום נקודות מעקב בזמן ריצה.

לא כל קריאות ה- ()mark המתבצעות במהלך ריצה ניתנות לשימוש עבור בנית הגרף המייצג את טביעת האצבע לחתימת המים.

נקודת סימון (קריאת ()mark שבוצעה בזמן ריצה) המורכבת מ-<ערך, מיקום> היא **נקודה תקפה** (valid) לשימוש אם היא **נקודה יחידנית**. נקודת סימון היא יחידנית אם:

1. ישנה בדיוק נקודה אחת כזו במיקום נתון, או
2. ישנם מספר נקודות סימון במיקום נתון , אך לכל אחת מהן ערך אחר.

נקודה תקפה לשימוש אך ורק אם היא יחידנית, אחרת, קוד (לבנית הגרף המייצג) שיוטמע בנקודה זו עלול להיקרא מספר פעמים, בעלות גבוהה לזמן הריצה ולבזבוז זיכרון ומקום בערימה וזאת שלא לצורך.

לדוגמא:

<0, L_0>
<1, L_1>
<1, L_1>
<10, L_2>
<11, L_2>
<12, L_2>

נקודת הסימון <0, L_0> היא יחידנית מכיוון שהיא היחידה במיקום 0.

הנקודות <10, L_2>, <11, L_2>, <12, L_2> הם יחידניות מכיוון שערכן שונה.

<1, L_1> אינה יחידנית, מכיוון שישנן שתי נקודות זהות כאלה באותו המיקום ועם אותו הערך. אם היינו מכניסים במיקום זה קוד לבניית גרף המייצג את טביעת האצבע, היינו צריכים להשתמש במשתנה מצב על מנת להבחין בין הקריאה הראשונה לשנייה. הכנסה של משתנה כזה הייתה חושפת את התוכנית לפגיעות בפני התקפה מכיוון שתוקף עלול היה להבחין במשתנה זה כאשר היה משווה את התוכנית לתוכנית שאינה מסומנת ב"סימן המים". במצב שכזה יכול התוקף לבנות פונקציית detect() ופונקציית attack() שמעוותת או מסירה את "סימן המים" מבלי לפגוע בנכונות התכנית.

אם ישנו רק ערך יחיד של קריאת mark() במיקום נתון נאמר שנקודה זו היא נקודה **מבוססת מיקום** (LOCATION-based) אחרת נאמר שהיא נקודה **מבוססת ערך** (VALUE-based).

בקריאת mark() מבוססת מיקום הקוד לבניית סימן המים יבוצע ללא תנאי. לעומתו, קוד לבניית סימן המים בנקודת mark() מבוססת ערך, חייב להיות מוגן על ידי פרדיקט לבדיקת הערך. נקודות כאלה נחשבות לגבוליות מבחינת התקפות שלהם באלגוריתם וזאת מכיוון שהפרדיקט המגן אינו חשאי (מוסתר) שכן הוא מבצע בדיקה (גלויה) על ערך שקיים גם בתוכנית שאינה מסומנת.

בנוסף לתכונת היחידנות, לנקודות mark() תקפות (valid) ישנם שלושה מאפיינים נוספים:

1. הן חייבות להיות **ניתנות לשחזור**. כלומר, הם יתרחשו שוב בריצה נוספת של התוכנית עם אותו הקלט.

2. הן חייבות להיות **יעילות**. כלומר, אסור שיתרחשו מספר פעמים רב בכל קלט שהתוכנית יודעת לקבל.
3. הן חייבות להיות **מסוימות** (specific). כלומר, קריאות ה- ()mark לא יתרחשו בקלטים אחרים (או לפחות לא יתרחשו בהרבה קלטים אחרים) מאשר הקלט הסודי לבניית טביעת האצבע.

אם לא נוצרו נקודות מעקב תקפות, אזי יש צורך לבצע את שלב המעקב שוב עם סט חדש של נקודות מעקב ו/או סדרת קלט סודית אחרת.

על מנת לבצע את השלב הזה באלגוריתם בצורה טובה יש לקבוע **ערכי סף** לתקפות נקודות הסימון עבור כל אחד משלושת המאפיינים - ניתנות לשחזור, יעילות, מסוימות.

תקפות (validity) נקודות הסימון יחד עם מספר ריצות המעקב ומספר האפשרויות להכנסת הקלט, כולם הם החלטות שיש לעשות בצורה נכונה בהקשר לתכנית המסוימת אותה רוצים לסמן.

ערך הסף לתכונת ה"**מסוימות**" הוא אפס לכל תכנית שאינה מקבלת קלט.

תכנית שמקבלת קלטים שונים יכולה להיות בעלת טביעת אצבע ייחודית, למרות זאת יש צורך לבחור סדרת קלט אחת שיוצרת סדרת קריאות ()mark שנוצרות בצורה אמינה. ככל שמגדילים את הייחודיות של טביעת האצבע, כך מגדילים את הסתרתה כנגד התקפות.

ערך הסף לתקפות נקודות הסימון בתכונת **היעילות** תלוי באילוצי הביצועים של התכנית. אם התכנית ללא סימן המים קרובה מאוד למגבלת הביצועים, אזי כל תוספת שתגרע בביצועים בעקבות הוספת סימן המים אינה קבילה.

אם כל הקוד לבניית טביעת האצבע נמצא בנקודות מבוססות מיקום, אזי הן מבוצעות פעם אחת בלבד ולכן ניתן להסיק שזה קביל עבור כל התכניות למעט אלו שרגישות לביצועים. אולם, אם הקוד נמצא בנקודות מבוססות ערך, יתווסף קוד אשר יפגע בביצועים בזמן ריצה וזאת עקב הוספת הפיצול המותנה (בלוק if-else) אשר מונע הרצת קטע קוד זה יותר מפעם אחת. נקודות אלו אינם תקפות למשל בלולאות בתכניות רגישות לביצועים.

ערך הסף לתקפות נקודות הסימון בתכונת **ניתנות לשחזור** תלוי ברגישות לקבלת תוצאות שהנם מסוג תוצאה שלילית שגויה (false-negative) בזמן חילוץ סימן המים. בדרך כלל זו אינה בעיה (ממדידות שבוצעו) אלא אם התוכנית היא תכנית זמן אמת או מכילה תחרות בין תהליכונים (race conditions).

6.1.4 שיבוץ (embedding)

לאחר שנבחרו נקודות הסימון הקבילות לתכנית ניתן להתחיל בשיבוץ טביעת האצבע. הקלטים לשלב זה הם רשימת נקודות הסימון הקבילות, סימן מים w וקבצי jar המכילים את המחלקות אשר בהם יש לשבץ את סימן המים. (**<u>תזכורת</u>: האלגוריתם נבנה ומתואר עבור שפת java**)

שלב השיבוץ מחולק לארבע שלבי משנה:

1. יצירת גרף G אשר הטופולוגיה שלו מבוססת על w.
2. יצירת קוד ביניים C אשר בונה את גרף G.
3. תרגום קוד הביניים לשיטה (method) M בג׳אווה אשר בזמן ריצה תבנה את G.
4. לבסוף, החלפת קריאת ()mark תקפה בקריאת M. במידה ונותרות קריאות ()mark יש להתעלם מהם. עלולות להיות נקודות ()mark מיותרות שכן ייתכן שבזמן ביצוע שלב הסימון נוצרו קריאות ()mark רבות ותקפות להחלפה, אך לצורך הטמעת סימן המים לא נדרשות כול הקריאות. במילים אחרות, ליצירת הגרף המיצג את סימן המים נדרשות פחות קריאות בקוד מאשר כמות קריאות ה-()mark התקפות שהתקבלו.

התוצאה הוא קובץ jar חדש, אשר כשיורץ עם סדרת הקלט הסודי יריץ את M, כתוצאה מכך יבנה גרף G בזיכרון הערימה.

בהמשך העבודה, נרחיב את שיטת השיבוץ כך שגרף G יפוצל למספר חלקים (תתי גרף) ויוכנס במספר נקודות סימון תקפות שונות.

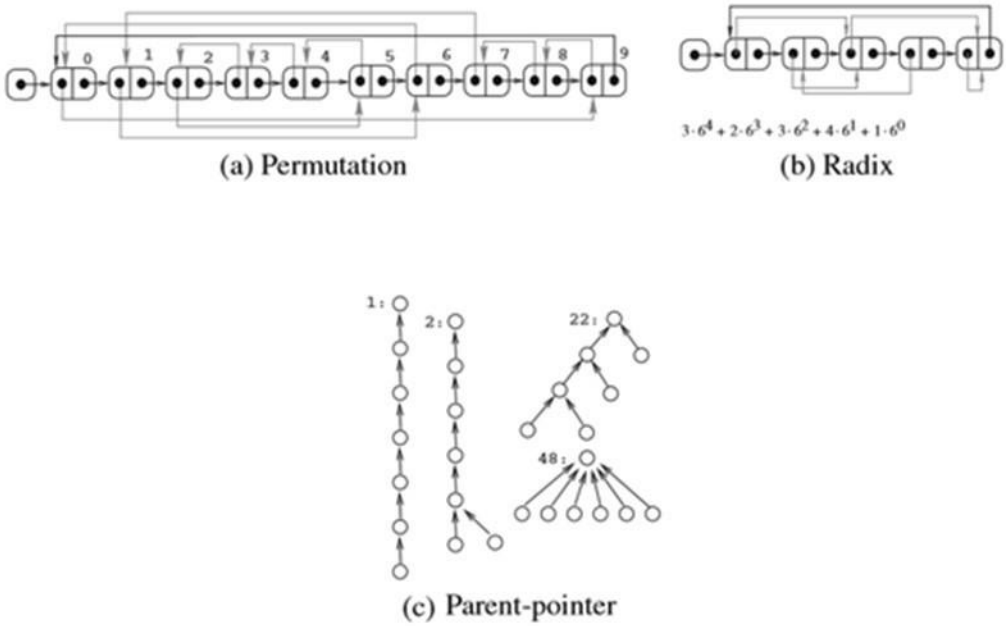
6.1.5 בניית הגרף

מחלקה אידיאלית עבור גרף המייצג את טביעת האצבע תהיה:

1. בעלת שורש אשר ממנו ניתן להגיע לכל הצמתים, וזאת בכדי למנוע מצב בו חלקים מהגרף אינם נגישים לתוכנית ולמעשה ינוקו על ידי ה-garbage collector.
2. בעלת תכולת נתונים גבוהה. כך שניתן יהיה לייצג חתימה גדולה בגרף קטן.
3. בעלת דמיון למבני נתונים אחרים (לא בולטת) כמו רשימות ועצים.
4. בעלת מנגנון לתיקון שגיאות כך ששינויים קלים בגרף המתרחשים כתוצאה מהתקפה או תקלה בזמן ריצה, לא ימנעו מהגרף להיווצר ולהיחלץ.
5. בעלת מבנה פנימי המאפשר יחסית בקלות את הקשחתו.

6. בעלת פונקציות חישוביות ריאליות לביצוע "ranking" ו-"unranking" ([32] Myrvold and Ruskey [2001]) של כל הגרפים במחלקה המאפשרות הפיכת מספר שלם לגרף בזמן השיבוץ, וגרף למספר שלם בזמן החילוץ.
7. בעלת אלגוריתם חישובי ריאלי לאיזומורפיזם של גרף, לשימוש בזמן זיהויו.

אין לצפות למצוא מחלקה אחת ליצירת גרף שעונה על כל הקריטריונים הללו בו זמנית ובצורה מיטבית. במקום זאת ניתן לפתח ספריה של אלגוריתמים לבניית גרפים עם סט מאפיינים שונים. בהתאם לדרישות הפרטניות של המשתמש (תכולת נתונים גבוהה, התאוששות גבוהה מהתקפות, חשאיות גבוהה וכו') ייבחר הגרף המתאים (או צירוף של מספר גרפים). הכלי SandMark למשל, מכיל מימוש של שניים (radix/permutation) משלשת הגרפים המתוארים באיור 10.



איור 10 – גרפי קידוד לסימן המים.

את הגרפים ניתן לייצר על סמך קידודים שונים כלהלן:

1. קידוד תמורה (Permutation encoding)

מספר שלם לסימן מים W , בתחום [0…n-1] ניתן להציג על ידי תמורה של המספרים <0,..., n-1>.

ניתן להשתמש בכל סוג של מיפוי תמורה על מספרים שלמים, למשל פונקציית "unrank1" (Myrvold and Ruskey 2001 [32]) המפורטת בנספח 11.3, תמפה את המספר 180398 לפרמוטציה π = <9, 6, 5, 2, 3, 4, 0, 1, 7, 8>.

כמובן שיש להשתמש בפונקציה זו בזמן יצירת הגרף ובפונקציה ההופכית לה "rank1" לאחר חילוץ הגרף (נספח 11.3).

נשתמש ברשימה מקושרת חד כיוונית מעגלית על מנת לייצג תמורה זו. מבנה זה נקרא "גרף התמורה" וניתן לראותו באיור 10(a). לכל אלמנט i ברשימה יש שני מצביעים. מצביע הנתונים שלו מצביע על π(i) שאליו i ממופה בתמורה π. בנוסף יש לו מצביע רשימה אל האלמנט (I + 1) mod n.

בזמן חיפוש הגרף יש צורך להבחין בין מצביעי הרשימה למצביעי הנתונים. המבנה המעגלי מאפשר לנו את זה על ידי חיפוש המעגל הפשוט הארוך ביותר בגרף.

תכולת הנתונים הדינמית בגרף זה היא המספר (lg n!) של ביטים המיוצגים על ידי רשימה בת n אלמנטים, לחלק למספר הבתים (an + b) הדרושים לייצוג רשימה זו בזיכרון המחשב.

על מנת להטמיע מספר שלם לא שלילי w המייצג את טביעת האצבע משתמשים ב - *n = min {k : k! > w}* אלמנטים ברשימה.

שימוש בקירוב של סטרלינג (קירוב מתמטי ל-n!) עבורו נסמן *n = m/lg m + O(m/lg lg m)* כאשר $m = \lceil \lg(w + 1) \rceil$ הוא מספר הביטים ב- w, כך שתכולת הנתונים הדינמית *r(m) = (lg m)/a + O(lg m / lg lg m)* עולה לאט כפונקציה של m.

ניתן להעריך את תכולת הנתונים הדינאמית על ידי הערכה של a ו-b. שניהם שלמים קטנים שערכם המדויק תלוי בפרטי המימוש. במחשב בעל מרחב זיכרון של 32 סיביות, נצפה ש-a = 16 בתים, מכיוון שהקצאות זיכרון דינמיות בדרך כלל משתמשות בגודל מסגרת שהיא חזקה של 2 ולכל תא ברשימה ישנם שני מצביעים ותוספת תקורה עבור מיקום האחסון. נצפה שגם מקדם התקורה של הרשימה b יהיה קטן, ייתכן ש-16 בתים. ערכו המדויק של b אינו רלוונטי כאשר מחשבים תכולת נתונים עבור טביעות אצבע גדולות, והוא גם לא ממש חשוב אפילו כאשר m הוא קטן. על בסיס הערכות אלה, ניתן לחשב שתכולת הנתונים הדינאמית עבור גרף תמורות הוא lg(n!) / (an + b) = 32.5/224 = **0.15** ביטים חבויים לבתים שהם תקורה, כאשר m = 32.5 ביטים משובצים בגרף תמורה בעל n = 13 אלמנטים ברשימה. נתון זה אורך טביעת האצבע הוא lg m עבור טביעות אצבע גדולות יותר. בהמשך העבודה, ייסקרו עולה לאט בצורה פרופורציונלית ל- אנליזות על מדידות במימושים שנעשו ואשר מאשרות הערכות אלה (פרק 6.3).

2. קידוד בסיס (Radix encoding)

איור 10(b) מתאר גרף ברשימה מקושרת מעגלית באורך n. שדה מצביע הנתונים מקודד מספר בבסיס n באורך הנתיב מהצומת ובחזרה לעצמו. המספר המקודד נקבע על סמך הפונקציה הבאה:

$$Index = a_1 \times k^0 + a_2 \times k^1 + \ldots + a_i \times k^{i-1} + \ldots + a_{k-1} \times k^{k-2}$$

צומת ברשימה המקושרת מכיל שני מצביעים, ימני ושמאלי. הערך של a_i נקבע על ידי המצביע השמאלי לפי החוקים הבאים:

- מצביע ריק (null) מקודד את הערך 0
- מצביע לעצמו מקודד את הערך 1
- מצביע לצומת הבא מקודד את הערך 2 וכו'.
- מצביע לצומת הראשונה הקודמת לו ברשימה, יקודד את הערך k-1
- מצביע לצומת השנייה הקודמת לו ברשימה, יקודד את הערך k-2 וכו'.

איור 14(b) מקודד בגרף בבסיס 6 את המספר (אורך הגרף n=5):

$$3 \times 6^4 + 2 \times 6^3 + 3 \times 6^2 + 4 \times 6^1 + 1 \times 6^0 =$$
$$= 3888 + 432 + 108 + 24 + 1 =$$
$$4453$$

גרפים אלו דומים במבנם לגרפי תמורה, אבל הם בעלי תכולת נתונים גבוהה יותר ורגישים יותר לתיקון שגיאות וזאת מכיוון שישנם פחות אילוצים על המצביעים שלהם.

גרף בסיס באורך n יכול לייצג כל מספר שלם בטווח שבין $0 \ldots (n+1)^n - 1$. הרשימה דורשת $an + b$ מילים (words), כך שתכולת הנתונים הדינאמית שלו כפונקציה של n היא $n \lg(n+1)/(an+b) \approx (\lg n)/a$. סביר שהערכים של a ו-b יהיו דומים לאלו של גרף תמורה, ואכן אנליזות שנעשו בניסויים מאשרות הנחה זו (פרק 6.3).

אם $a = b = 16$ בתים, גרף בסיס באורך $n = 9$ יחביא $m = n \lg (n+1) = 29.9$ ביטים של מידע ב- $an + b = 160$ בתים של מידע בתכולת נתונים דינמית של **0.19** ביטים חבויים לבתים שהם תקורה.

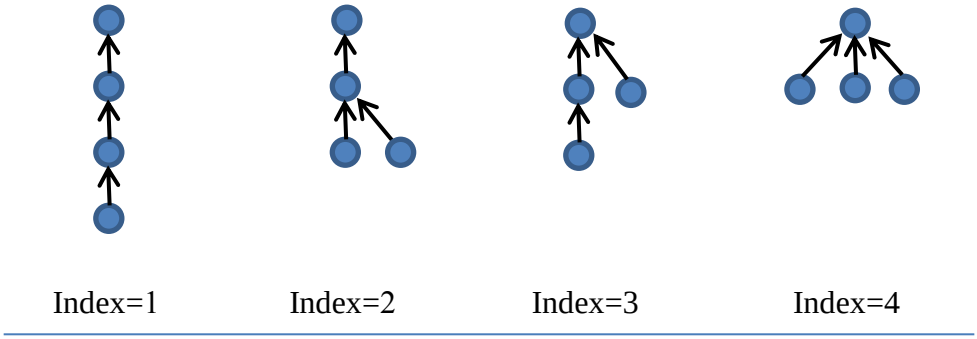
ניתן להבחין כי גרפי בסיס יעילים יותר מגרפי תמורה, וזאת מכיוון שגרפי בסיס הם במבנה פחות קשיח ולכן נושאים יותר מידע עבור כל מספר קבוע של רשימת אלמנטים. כל גרף תמורה הוא גם גרף בסיס אך לא כל גרפי הבסיס הם גרפי תמורה. ככל הנראה תכולת הנתונים הסטטית בגרף בסיס, בכל

מימוש, תהיה מעט טובה יותר מאשר בגרף תמורה במימוש דומה, וזאת מכיוון שמימוש דומה יבנה את אותם מבנים, אך בגרף התמורה יטמיעו פחות מידע חבוי פר אלמנט ברשימה.

3. Parent-Pointer Trees (PPT)

בעצים אלה לכל צומת מצביע להורה (בלבד). עצים עם מצביעים להורה ניתנים למניה בהתאם לטכניקה שמתוארת ב- [33] (Knuth 1997).

איור 10(c) מדגים שימוש בעץ שכזה. האיור הבא מתאר מניה של PPT עם 4 צמתים:



איור 11 – גרף PPT בעל 4 צמתים

בונים את הדרגה או את האינדקס w של PPT על ידי סידור העצים. המניה עבור PPT בעל n צמתים היא על ידי הסדר הנקבע לפי "תת העץ הגדול ביותר ראשון". עץ ללא הסתעפויות (רשימה באורך n-1) יקבל את המספר הסידורי 1. המספרים הסידוריים מ-2 ועד n-1 יקבלו העצים האחרים אשר אין להם הסתעפות בשורש, כלומר כאשר יש תת עץ יחיד בגודל n-1 המחובר לצומת השורש של העץ. המספרים הסידוריים a_{n-1} עד $a_{n-2} + a_{n-1}$ יקבלו עצים אשר להם בדיוק 2 תתי עצים המחוברים לשורש העץ ואשר לאחד מתתי העצים יש בדיוק n-2 צמתים. המספרים הסידוריים הבאים a_2 a_{n-3} a_{n-3} = יינתנו לעצים אשר להם בדיוק 2 תתי עצים המחוברים לשורש ואשר אחד מתתי העצים מכיל בדיוק n-3 צמתים. a_n הוא צומת השורש של עץ/תת עץ בעל n צמתים.

המספר a_n של PPT עם n צמתים הוא אסימפטוטי $a_n = c(1/\alpha)^{n-1}/n^{3/2} + O((1/\alpha)^n/n^{5/2})$, כאשר $c \approx 0.44$ ו- $1/\alpha \approx 2.956$ ולכן ניתן לקודד מספר שלם שרירותי w של 1024 ביטים בגרף סימן מים על ידי $1024 / \lg 2.956 \approx 655$ מצביעים. זה ידרוש 2620 בתים בארכיטקטורה של 32 ביט, במידה ומצביעים אלה הוספו לעצמים שהוקצו על ידי התוכנית המקורית שלא סומנה (ולא התווספו באובייקטים חדשים שנוצרו לשם כך). תכולת הנתונים הדינמית עבור PPT יכולה להיות $\lg(2.956)/4 \approx 0.4$ ביטים חבויים לבית תקורה עבור טביעות אצבע גדולות כמו $m = 1024$. תכולת

הנתונים הדינאמית תהיה נמוכה משמעותית אם העצמים לבניית הגרף יבנו בצורה נפרדת והמצביעים לא יתווספו לעצמים קיימים. תכולת הנתונים לא תשתנה משמעותית עם m, למרות שעבור ערכי m קטנים היא תהיה נמוכה מ- 0.4. למרות זאת, ניתן לחשוב כי PPT עדיפים (מבחינת תכולת הנתונים) על גרפי תמורה או גרפי בסיס עבור m קטן או בינוני, אבל לא עבור m גדול. לפי הערכות שנעשו, גרף בסיס עבור n = 128, יכול לייצג m = 897 ביטים חבויים בתכולת נתונים של 0.43.

ראוי לציין, שמבנה נתונים זה (PPT) טרם מומש בכלים שפותחו בנושא ולכן אין נתונים ניסויים שיכולים לאשש את החישובים לעיל. בנוסף אציין כי בפרויקט המסכם המתואר כאן בפרק 8. תוצאות ומסקנות ממימוש האלגוריתם בפרויקט המסכם, מומשו לבחירת המשתמש קידוד תמורה וקידוד בסיס.

6.1.6 חילוץ (extraction)

בכדי לחלץ את טביעת האצבע, התוכנית המסומנת רצה כתת תוכנית (sub process) במצב ניפוי שגיאות (debugging mode), ותוך שימוש בספריית JDI debugging framework*.

המשתמש מזין את סדרת הקלט $I_0, I_1,...$ בדיוק אותה הסדרה שהוזנה בשלב המעקב. פעולה זו תגרום לקריאות של Watermark.Create() להתבצע ולגרף טביעת האצבע להיווצר בזיכרון הערימה של התוכנית. כאשר הוזן הקלט האחרון בסדרה, רכיב מחלץ צריך לאתר את הגרף בזיכרון, לפענח אותו ולהציג את ערך טביעת האצבע שחולץ ופוענח למשתמש.

*__הערה__: הבחירה של כותבי המאמר לבצע את פעולת החילוץ כך נבעה מעצם העובדה שהמימוש אותו הם מציגים במאמר נעשה בשפת ג׳אווה. בפרויקט המסכם (המעשי) שלי, שהנו המשך לעבודה זו, בחרתי להציג מימוש לאלגוריתם CT בשפת C# ובטכנולוגית Microsoft .Net ולכן השימוש שאעשה (בעיקר בשלב החילוץ) יתבצע בשיטה שונה (אין צורך במצב ניפוי שגיאות) ובספריות קוד שונות (כמו Microsoft.Diagnostic.Runtime).

כמובן שבזיכרון הערימה עלול להיות מספר עצום של עצמים ועלול להיות בלתי אפשרי לבדוק את כולם. בכדי להפחית את מרחב החיפוש אנו נסתמך על ההבחנה שכאשר הוזנה סדרת הקלט הסודית, צומת השורש של הגרף יהיה אחד העצמים האחרונים שהוספו לזיכרון הערימה. כך שאסטרטגיה טובה תהיה לבדוק את זיכרון הערימה בסדר הפוך לזמן הקצאת העצמים. כמובן גם שתוקף, שמסוגל לנחש את סדרת הקלט הסודית (או אפילו חלק ממנה) ומריץ את התוכנית המסומנת יכול לנקוט באותה האסטרטגיה בכדי למצוא צומת המועמד להיות שורש הגרף. מכיוון שאנו פועלים תחת ההנחה שמערכת לסימון תכניות בסימן מים חייבת לשמור בצורה סודית על סוג סימן המים (טביעת

האצבע) המוסתרת וכמובן על סדרת הקלט, ולכן תחת הנחה זו, התוקף יתקשה מאד לדלות מידע מניתוח הריצות של התוכנית.

נכון לזמן כתיבת המאמר, לא הייתה קיימת תמיכה ב-JDI עבור יכולת המאפשרת לתוכנית לבדוק את (או לרוץ על) זיכרון הערימה של תוכנית אחרת בזמן שהיא רצה**. ולכן קיים קושי עצום לבנות מחלץ סימן מים, ולבנות אותו יעיל. ולכן הפתרון המוצע אינו יעיל מצד אחד אך מצד שני הוא יקשה מאוד גם על התוקף לבצע תקיפה.

הדרך האלגנטית והיעילה ביותר היא לשנות את הבנאי (Constructor) של מחלקת הבסיס של עצמי ג׳אווה java.lang.Object ולהכניס בה מונה, בצורה הבאה:

```
package java.lang;
public class Object {
    public static long objCount = 0;
    public long allocTime;
    public Object() {
        allocTime = objCount++;
    }
}
```

איור 12 – התאמת הבנאי של מחלקת הבסיס ל-CT

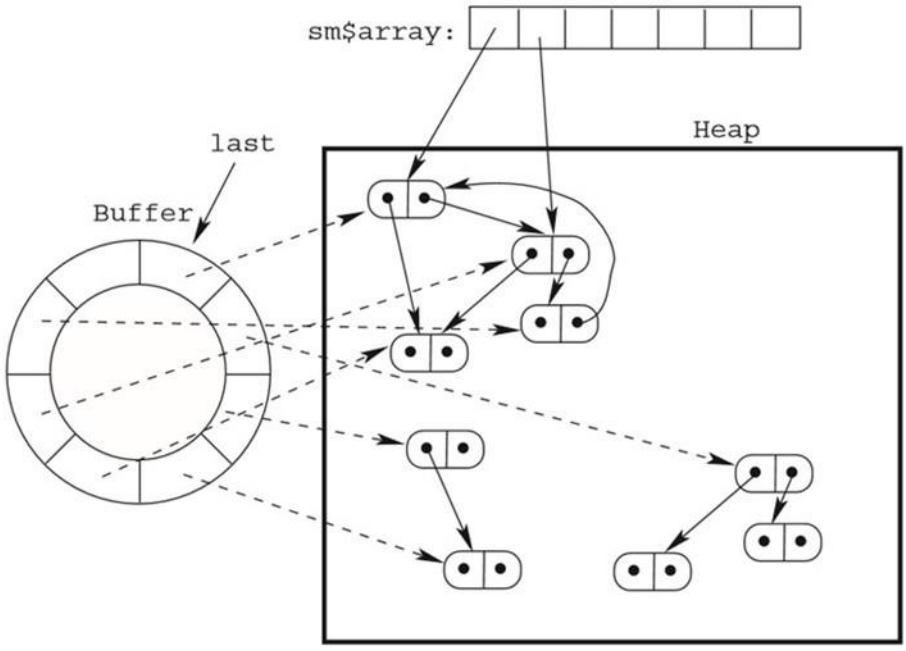
<u>הערה</u>: מימוש אלגוריתם CT בשפת C# ובטכנולוגית Microsoft .Net כן מאפשר לבנות מחלץ יעיל שיכול לרוץ על זיכרון הערימה של תוכנית אחרת.

מכיוון שבנייה של כל עצם בשפת ג׳אווה מחייבת קריאה למחלקת הבסיס ולבנאי זה, המשמעות היא שהכנסנו את סדר ההקצאה של העצמים בתוכנית בזיכרון הערימה וזאת בעלות זמן של תוספת השמה אחת ופעולת חיבור אחת ובתוספת זיכרון של משתנה אחד מסוג long (8 בתים) סטטי לכל העצמים ואחד עבור כל עצם.

אך גישה זו אינה מעשית, מכיוון שהיא דורשת לשנות את ספריות הריצה הבסיסיות של שפת ג׳אווה, יותר מזה סביר שחלק מהמהדרים הקיימים של השפה יבצעו אופטימיזציה ויבטלו תוספת זו (אופטימיזציה של זמן ריצה וזיכרון).

ולכן מוצעת גישה אחרת, יותר קשה למימוש אך יותר יבילה (portable), על ידי שימוש ב-JDI מוסיפים נקודת עצירה לכל בנאי בתוכנית. בכל פעם שמתרחשת הקצאה של עצם מוסיפים מצביע לחוצץ (buffer) מעגלי אשר מצביע לעצם זה. בצורה כזו ניתן לגשת אל 1000 (או כל מספר אחר) העצמים האחרונים שהוקצו בזיכרון הערימה. ניתן לראות זאת ב**איור 13 – אלגוריתם נאיבי לחילוץ סימן המים ב- CT**, החיסרון העיקרי הוא בהאטה משמעותית בביצועי התכנית בזמן החילוץ עקב הצורך לטפל בנקודות העצירה.

בכדי לחלץ את הגרף המייצג את סימן המים, יש לעבור על העצמים בסדר הפוך לסדר יצירתם ולחפש תת גרף נגיש עד שימצא אחד כזה. חילוץ טביעת האצבע מהגרף ניתן לביצוע על ידי האלגוריתם הנאיבי שמוצג ב**איור 14 – אלגוריתם נאיבי לחילוץ סימן מים המיוצג על ידי גרף מזיכרון הערימה**.



איור 13 – אלגוריתם נאיבי לחילוץ סימן המים ב-CT

ב**איור 13 – אלגוריתם נאיבי לחילוץ סימן המים ב-CT** , ניתן לראות את מצב הזיכרון בזמן החילוץ. חוצץ מעגלי מכיל מצביעים לכל העצמים האחרונים שנוצרו בזיכרון הערימה. המחלץ יבדוק את החוצץ בסדר הפוך לסדר הקצאתם ויחפש את שורש הגרף שיכול לשמש כגרף המייצג סימן מים.

```
for each graph codec C do
   for each node n on the buffer, in reverse allocation order do
      G = heap graph with root n, edges [e1,e2,...,ek];
      for each pair (ei,ej) of edges in [e1,e2,...,ek] do
         G' = subgraph of G with edges labeled (ei,ej);
         if fingerprint decoding w=C(G') succeeds then
            yield w
```

איור 14 – אלגוריתם נאיבי לחילוץ סימן מים המיוצג על ידי גרף מזיכרון הערימה

האלגוריתם שבאיור 14 מראה כיצד לבצע בדיקת התאמה של גרפים מזיכרון הערימה. ההנחה היא שהגרפים הם עם דרגת יציאה של 2 (כלומר מכל צומת יוצאות 2 קשתות). האלגוריתם מחלץ גרפים באמצעות פונקציה C, עבור כל צומת n בכל גרף שהתקבל מ-C יש למצוא את G שהוא הגרף עבורו n הוא השורש. עתה, יש לחפש הפוך לסדר הזמנים של הקצאת האובייקטים בזיכרון (ההנחה היא שסימן המים הסתיים להבנות בעצמים האחרונים שהוקצו בזיכרון הערימה), עבור כל גרף שכזה ועבור כל תתי הגרפים שלו (צמתים שמהם יוצאות 2 קשתות), יש לחפש התאמה לסימן המים, ובמידה ונמצא כזה להחזירו.

6.2 שיפורים מוצעים לאלגוריתם

6.2.1 שיפור העמידות

על מנת לתכנן ולהעריך אלגוריתם לסימן מים בתוכנה חיוני להגדיר מודל מדויק של התקפה ריאלית. ההנחה היא שהתוכנית המסומנת גדולה מדי עבור בדיקה ידנית של התוקף, במילים אחרות זה לא מעשי עבור התוקף לקרוא את התוכנית (קובץ בינארי לאחר הידור) על מנת לאתר ולהרוס את טביעת האצבע. במקום זאת, יש להגן על התוכנית כנגד התקפות מחלקה (class attacks) – בנייה של שיטות אוטומטיות שנועדו להרוס את טביעת האצבע. לדוגמא, אם המרת קוד מסוימת (transformation) יכולה להרוס את טביעת האצבע, אזי יהיה לתוקף קל מאד לבנות התקפה שכזו שתהרוס את כל טביעות האצבע בכל תוכנית.

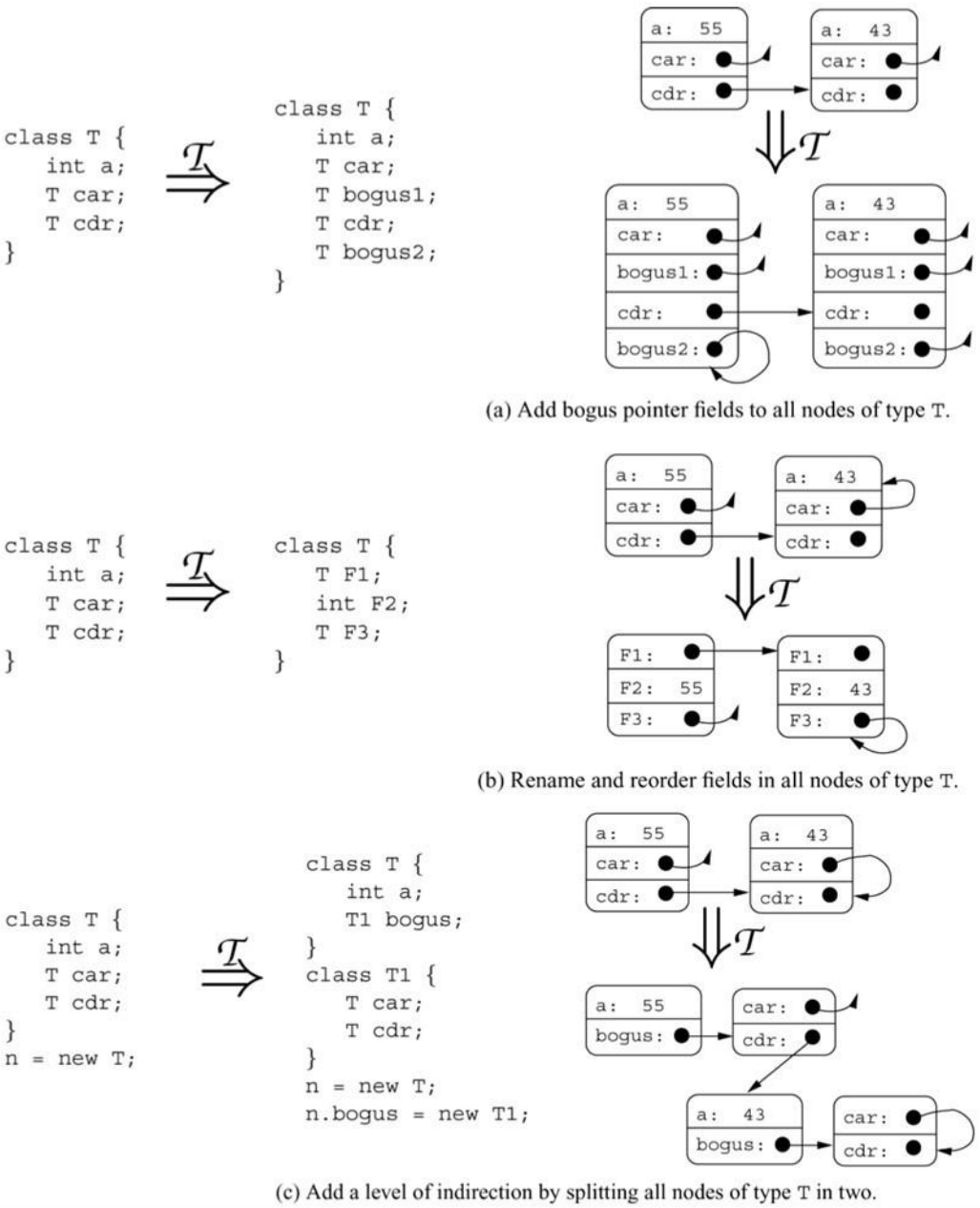
אבחנה מעניינת של גישת אלגוריתם CT היא שהרבה המרות תרגום, אופטימיזציות, וערפול לא משפיעות על המבנים שמוקצים בזיכרון הערימה. אך קיימות טכניקות אחרות שעלולות לערפל נתונים שנבנים דינאמית, בעיקר עבור שפות עם קוד המחייבות הגדרה מפורשת של טיפוס הנתונים (Strongly typed code) כמו java (ו-C#).

ניתן למנות ארבעה סוגים של המרות ערפול שמסוכנות לסימן מים דינאמי בתוכנה, בכדי לנטרל את המחלץ (שלב חילוץ הגרף המייצג את טביעת האצבע), תוקף יכול:

1. להוסיף מצביעים מיותרים לצמתים של המבנים המקושרים (איור 15 a) על מנת להקשות על המחלץ לזהות את צמתי הגרף האמתיים בין המצביעים מזויפים, מכיוון שכך מבנה הנתונים יהיה מורכב יותר להבנה וניתוח.
2. לשנות את השמות ו\או את הסדר של מופעים של משתנים (איור 15 b).
3. להוסיף רמות נוספות של מעקפים, לדוגמא על ידי פיצול הצמתים למספר חלקים מקושרים (איור 15 c).
4. להוסיף צמתים נוספים מזויפים המצביעים לגרף, המונעים מהמחלץ למצוא את שורש הגרף.

ניתן כמובן גם לשלב בין השיטות הללו.

למעט התקפה של סידור מחדש (סעיף 2), להתקפות אלה ישנה משמעות כבדה מאוד בדרישות הזיכרון של תוכנית תקיפה שכזו. לדוגמא, פיצול צומת עולה 12 בתים לכל צומת מוקצה (מצביע אחד של 4 בתים בתוספת של כ-8 בתים של תקורה שמקורה בג׳אווה עצמה). יותר מכך, מאחר שאנו מניחים שהתוקף לא יידע באיזה מבנה נתונים דינמי מוטמעת ומוחבאת טביעת האצבע הוא יבצע את ההמרה (טרנספורמציה) בצורה אחידה על כל התוכנית וזאת בכדי להיות בטוח שטביעת האצבע הושמדה. במילים אחרות, תכניות בעלות קצב הקצאות זיכרון דינאמי גבוה הן בסבירות טובה לעמידות כנגד התקפות אלו וזאת מאחר שתוכנית שכזו לאחר התקפה תצרוך כמות זיכרון גבוהה יותר משמעותית מאשר התוכנית המסומנת לפני שהותקפה.



(a) Add bogus pointer fields to all nodes of type T.

(b) Rename and reorder fields in all nodes of type T.

(c) Add a level of indirection by splitting all nodes of type T in two.

איור 15 – התקפות ערפול כנגד גרפים המייצגים סימן מים

6.2.1.1 הריסת טביעת האצבע על ידי סידור מחדש

דרך פשוטה להגן מהתקפה של שינוי השמות ו\או את הסדר של מופעים של משתנים (איור 15 b) היא להתחשב בסדר של כל משתנה בזיכרון בזמן החילוץ, אך זה עלול להוביל לעליה בשיעור תוצאות חיוביות שגויות (False Positive) מאחר שגרפים של המשתמש תחת טרנספורמציות סידור מחדש

מסוימות עלולים להופיע באותו מבנה כמו של גרף טביעת האצבע. גישה טובה יותר היא לבחור מחלקה של גרפים אשר עבורה הסדר של הקשתות היוצאות אינו משפיע על ערך טביעת האצבע.

6.2.1.2 הריסת טביעת האצבע על ידי פיצול צמתים

פיצול צמתים מהווה מתקפה יעילה כנגד גרפים המייצגים סימן מים, אך למתקפות אלה קיימת השפעה מזיקה מאוד על ביצועי התוכנית (לאחר המתקפה).

6.2.1.3 הריסת טביעת האצבע על ידי הוספת שדות מזויפים

יכולות ההשתקפות (reflection) של ג׳אווה ושפות אחרות נותנות לנו דרכים פשוטות להקשחת הגרף כנגד סוגים רבים של התקפות, כולל התקפות של הוספת שדות מזויפים לצמתי הגרף.

נניח שברשותנו המחלקה הבאה המייצגת צומת בגרף:

```
class Node {
    public int a;
    public Node car, cdr;
}
```

אזי, המחלקות בג׳אווה (reflection) מאפשרות לנו לבדוק את השלמות של עצמים מסוג זה בזמן ריצה בצורה הבאה:

```
Field [] F = Node.class.getFields();
if (F.length != 3) die();
if (Field[1].getType() != Node.class) die();
```

למרבה הצער, קוד שכזה אינו חבוי בתוכנית שלא תוכננה לשימוש ב-reflection.

ניתן להשתמש ב-reflection על מנת להגן כנגד התקפות של סידור מחדש ושינוי שמות. הרעיון הוא לגשת למצביעים של טביעת האצבע באמצעות reflection. לדוגמא, במקום לבצע את ההשמה הבאה:

O.car = V;

car תיוצג על ידי המצביע הראשון שנמצא בצומת O בצורה הבאה:

```
Field [] F = Node.class.getFields();
int n=0;
for (int i=0; i<F.length; i++)
    if (F[i].getType().isAssignableFrom(Node.class)) {
        F[i].set(O, V);
        break;
    }
```

שימוש ב-reflection לא נעשה בכל התוכניות ולכן לא תמיד מעשי (מסיבה זו שיטה זו לא מומשה ב-SandMark).

6.2.1.4 ביצוע מניפולציות על הגרפים (Collberg and Thomborson (2007) [19])

המניע העיקרי בשימוש בגרפים דינמיים המייצגים סימן מים הוא שהקוד שבונה את טביעת האצבע יהיה קשה לניתוח. הסיבה היא בקושי המובנה שבניתוח מצביעים (נספח 11.4). אולם, קיימים אלגוריתמים לניתוח מצביעים שעובדים היטב כאשר מבנה הנתונים אינו מעגלי (כמו רשימות מקושרות ועצים) וכאשר הקוד נועד אך ורק לבניית מבנה הנתונים. אלגוריתמים אלו יבצעו בהצלחה אנליזה לקוד שבונה רשימות לינאריות, אבל הם עלולים להיכשל אם הרשימה תופרד למספר רכיבים ולאחר מכן תורכב מחדש (Ghiya and Hendren [12]).

ניתן להגדיל את החוסן להתקפות המבוססות על אנליזות של השוואת תבניות על ידי ניצול חולשה זו שקיימת באלגוריתמים אלו. לדוגמא, ניתן לא רק למזג חלקי גרף אלא גם, במהלך יצירת, הגרף לפצל אותו למספר רכיבים שלאחר מכן יורכבו מחדש.

6.2.2 הגדלת גודל טביעת האצבע

הפקודות הדרושות לבניית הגרף המייצג את טביעת האצבע (השמות של מצביעים והקצאות זיכרון דינאמיות על ידי 'new') לכשעצמם הן חשאיות, אך מספר רב של פעולות כאלה, המבוצעות במקום סמוך או בזמן סמוך בקוד עלול לעורר חשד. יותר מכך, טביעת האצבע שלנו הופכת להיות מוכוונת קלט כאשר היא נבנית בזמן ריצה במספר נקודות כתלות בקלט מאשר בנקודה אחת יחידה. מסיבות אלו נפצל את הגרף G למספר רכיבים $G_0, G_1, \ldots$ אשר הקוד שלהם פרוס לאורך נתיב הריצה המיוחד (תלוי הקלט). ישנם מספר נושאים אשר צריך להתחשב בהם:

1. תתי הגרף צריכים להיות בערך בגודל שווה. (מסיבות של חשאיות, ייתכן שעדיף שתתי הגרף יהיו בגדלים רנדומליים, אך המימוש הנוכחי של האלגוריתם מפצל את הגרף לתת גרף דומים בגודלם).
2. פיצול הגרף G צריך להיעשות בצורה כזו שלכל תת גרף יש שורש, צומת מיוחדת שממנה כל שאר הצמתים בגרף נגישים. זה מאפשר להחזיק רק מצביעים לצמתי השורשים על מנת למנוע ממנגנון איסוף הזבל (garbage collection) לנקות את תתי הגרפים.
3. יש לנסות לפצל את הגרף G בדרך כזו שמספר הקשתות בין תתי הגרף הוא המינימלי. הסיבה לכך היא שככל שיש יותר קשתות בין תתי הגרפים נדרש לייצר יותר קוד על מנת לחברם חזרה לגרף G.

6.2.2.1 פיצול מספר טביעת האצבע (Collberg and Thomborson (2007) [19])

גישה חלופית להגדלת החתימה היא לפצל את המספר המייצג את טביעת האצבע (במקום את הגרף המייצג את המספר). המספר n יפוצל למספר מספרים (קטנים יותר) $n_0, n_1,\ldots, n_k$ באמצעות משפט השאריות הסיני (Muratani, H. 2001 [36]). המספרים יקודדו לגרפים $G_0, G_1, \ldots, G_K$ אשר משובצים בנתיב הריצה הנבחר של התוכנית. הבעיה היא שבזמן החילוץ יש צורך למצוא את כל הגרפים הללו ולכן זה ימנע מאתנו לבצע חיפוש אך ורק ב-x העצמים האחרונים שהוקצו בתוכנית (פרק 6.1.6). במקום זאת ניתן לייצר k ריצות עבור k סדרות של קלטים ייחודיים, ולשבץ כל תת גרף בנתיב ריצה ייחודי של התוכנית עבור אחד מ-k נתיבי הריצה. אולם גישה זו תכביד מאוד על המשתמש הן בזמן השיבוץ והן בזמן החילוץ.

6.2.3 שיפור

בנושא זה, המטרה העיקרית של האלגוריתם היא להגן על טביעת האצבע מהרס באמצעים אוטומטים אך גם מהתקפות ידניות כשניתן. יותר מכך, רצוי שטביעת האצבע תהיה כמה שיותר חשאית כך שהתוקף לא יוכל לאתר את נוכחותה ברמת דיוק טובה.

המטרה של חשאיות מקושרת לעיתים עם המטרה של חוסן. ניתן לחשוב על תוקף שפיתח כלי אוטומטי שמזהה ברמת בטחון של 90% כמות קטנה (נניח 10%) של קוד הקשור בטביעת האצבע והמכיל קטעים לבנייתה. תוקף שכזה, יוכל לחפש ידנית אזורים דומים בתוכנית ולאתר בצורה נכונה חלקים או אפילו את כל הקוד הקשור ביצירת טביעת האצבע ולשנות אותו או אפילו למחוק אותו בהצלחה. מכאן החשיבות הרבה של החשאיות. נציג שלוש שיטות לשיפור החשאיות של הגרפים המייצגים את טביעת האצבע.

6.2.3.1 הימנעות ממשתנים גלובליים

עד כה הנחנו שצומתי השורשים של תתי הגרפים נשמרים כמשתנים סטטיים, שורשי תתי העצים נשמרו במשתנים גלובליים כמו מערכים וטבלאות גיבוב. זוהי כמובן גישה שאינה חשאית מאחר שתוכניות שנכתבות בשפות מונחות עצמים מודרניות מכילות מעט משתנים גלובליים. במקום זאת, נעביר את צמתי שורשי העצים כמשתנים פורמאליים ברשימת הפרמטרים (ארגומנטים) של פונקציות.

6.2.3.2 הגנה מפני התקפות מסוג התאמה סטאטיות (Static Collusive)

ניתן לבצע ערפול על התוכנית בה הוטמע סימן המים ובכך לסבך מאוד התקפות מסוג התאמת תבניות. יתרון אחד חשוב מאוד שקיים ביישום של סימן מים באמצעות גרף דינמי היא שהמרות ערפול טיפוסיות כמו סידור מחדש של הוראות, פיצול/מיזוג פונקציות, פיצול/מיזוג מחלקות וכו', לא ישפיעו על סימן המים שהוטמע, וזאת בניגוד לרוב השיטות של הטבעת סימן מים בתוכנה בהם ערפול הורס את סימן המים.

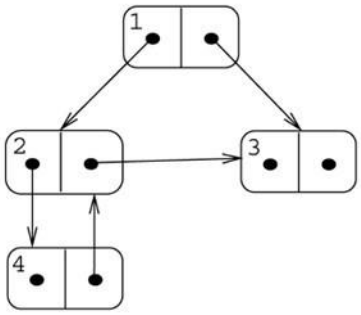
6.2.3.3 החבאת מחלקת סימן המים

בפרק 6 הומחשה יצירה של מחלקה בשם Watermark (**איור 8 – דוגמא – מחלקה לפני ואחרי אלגוריתם CT בסיסי**) אשר נוצרה לצורך יצירת צמתים בגרף סימן המים. ברור כי זו אינה דרך חשאיות במיוחד והיא עלולה למשוך תשומת לב שאינה רצויה.

במקום זאת רצוי לחפש בתוכנית מחלקה (קיימת) שדומה למחלקה Watermark. המחלקה האידיאלית, תכיל כבר שדה אחד או יותר מהסוג המתאים ליצירת צמתים. אם ישנן כמה מחלקות כאלה (מועמדות), נעדיף את אלו שיש להם יותר מופעים (עצמים) סטטיים. אם לא קימת אף לא מחלקה אחת שיש לה מספיק מצביעים, נמצא את המחלקה המתאימה ביותר ונוסיף לה שדות אלו.

במקרים בהם אין מחלקות משתמש מתאימות, והפתרון של הוספת מחלקות חדשות לטובת סימן המים לא יהיה חשאי, ניתן יהיה לנצל מחלקות מספריות ג'אווה.

לדוגמא, בנית גרף תוך שימוש במחלקה LinkedList של ג'אווה (המאפשרת לבנות רשימות מקושרות של רשימות מקושרות), שימוש במחלקה Event (אשר לה קיימים שני עצמים בעלי גישה ציבורית – target ו-arg) ומערכים באורך 2 המחזיקים אובייקטים כללים (Object). ניתן כמובן לשלב בין כל הסוגים הללו:




```
import java.util.LinkedList;
import java.awt.Event;
...
LinkedList n4 = new LinkedList();
n4.add(null);
LinkedList n2 = new LinkedList();
n2.add(n4);
n4.add(n2);
Event n3 = new Event(null,0,null);
n2.add(n3);
Object[] n1 = {n2,n3};
```

איור 16 – שימוש במחלקות LinkedList ו-Event לבנית גרף

במקרה הכללי, על מנת שמחלקה C (של ג׳אווה או כל שפה אחרת) תוכל לשמש כמחלקה עבור גרף דינמי לסימן מים, היא צריכה להחזיק לפחות שני שדות f_i מסוג מצביע t_i כך ש-C היא תת מחלקה של t_i ,ו- f_i הם בעלי הרשאה ציבורית (public) או שיש אליהם גישה באמצעות פונקציות getter/setter.

6.2.3.4 הגנה מפני התקפות מסוג התאמה דינמיות (Dynamic Collusive) (Collberg and Thomborson (2007) [19])

התקפות חזקות ועוצמתיות יותר מההתקפות הסטטיות יכולות להתרחש אם התוקף מורשה או יכול (יש לו את החומרה המתאימה למשל) להריץ את התוכנית בה הוטבע סימן המים, זה בעיקר תקף להתקפות מסוג התקפת התאמה.

לדוגמא ניתן לחשוב על התרחיש הבא. בוב רוכש שתי תכניות p1 ו –p2 מאליס, כאשר הוא יודע ששתיהן עברו שיבוץ של סימן מים באמצעות אלגוריתם CT. שתי התוכניות עברו ערפול באמצעות אלגוריתם חזק, כך שהתקפה סטטית של השוואת תבניות אינה רלוונטית. בוב יכול לנסות להתקיף בצורה דינמית את התוכניות. הוא יריץ את שתי התוכניות במקביל ויזין את אותו הקלט לשתיהן, ויסנכרן ביניהם על ידי עצירה בכל פעם שהתוכניות מבצעות שגרת מערכת (system call) למערכת ההפעלה. בשלב הבא הוא מכריח את שתי התוכניות לבצע ״איסוף זבל״ (garbage collection) ומשווה את כל המבנים המקושרים שנמצאים בזיכרון הערימה. בוב יניח בצורה טבעית כי שתי התוכניות יריצו את אותה סדרה של קריאות מערכת ובאותו הסדר. כאשר עוצרים בקריאת מערכת מסוימת שתי התוכניות צריכות להיות באותו המצב. כך שכשבוב משווה את שתי הערימות, כל מבנה שקשור ל-CT בערימה אחת שאינו תואם למבנה אחר בערימה השנייה בסבירות שיהיה שייך לטביעת האצבע עצמה. בוב יכול עתה לבצע מעקב אחורה בקוד ולגלות מתי מבנה זה נוצר. אם מתקפה שכזו מצליחה אזי טביעת האצבע אינה חשאית כלל.

תרחיש זה מייצג התקפה חמורה מאוד שיהיה קשה מאוד להתגונן נגדה. דרך אפשרית אחת להתמודד מולה היא שאליס תכניס לתוכניות שלה קריאות מערכת מיותרות בצורה רנדומלית, זה ימנע מבוב את היכולת לבצע סנכרון של התוכניות במהלך ריצתם. אך פתרון זה הוא בעייתי ליישום בתוכניות ג׳אווה טיפוסיות שכן לקריאות רבות יש השפעות לא רצויות או כאלה שקשה לבטלם כמו למשל פתיחת חלונות או כתיבה לקובץ. בוב יכול לבחור כל סט של קריאות מערכת לביצוע הניתוח הדינמי שלו, כך שסביר שהרשימה שלו תכלול גרפיקה וקריאות קלט פלט אבל לא תכיל קריאות שיש להם השפעות כאלה שהוא חושב שאליס יודעת כיצד לבטלם. כך שכנראה פתרון זה אינו אידיאלי עבור אליס.

אולם, ייתכן שאפשרי עבור אליס למנוע מבוב להפיק מידע מנקודות הסנכרון שלו. הרעיון הוא להבטיח שבזמן ריצת התוכנית ברוב נתיבי הריצה שלה יבנו גרפי דמה בסגנון CT. אם יבנו גרפי דמה כאלה ״במספרים גבוהים מספיק״ בזיכרון הערימה, בכל פעם שבוב יעצור את התוכניות בקריאות מערכת יהיה לו קשה להבחין בין הגרפים האמתיים ששיבצה אליס בתכנית לבין אלו שאינם אמתיים. למרבה הצער, להוסיף גרפי דמה שכאלה לקוד בצורה אוטומטית לתוכנית שרירותית אינו דבר טריוויאלי. יש צורך להבטיח שכל גרף דמה שכזה ישרוד את מנגנון ״איסוף הזבל״ שיפעיל בוב, וייאשר בזיכרון הערימה לאחר מכן אך לא יותר מדי זמן (או בכמות גדולה מדי) בכדי לא למלא את זיכרון הערימה.

א. הימנעות מקשרים חלשים (Collberg and Thomborson (2007) [19])

אחד החלקים החשובים באלגוריתם הוא לקשור את קוד סימן המים בצורה הדוקה לתוכנית. הסיבה לכך היא שחלקים של קוד הקשורים בצורה חלשה הם חריגים בתכניות אמתיות וניתן לאתרם בצורה קלה באמצעות אלגוריתמים לחיפוש גרפים בזיכרון הערימה. בכדי למנוע מתקפה שכזו Venkatesan et al.[15] הציעו לקשור את קוד טביעת האצבע לתכנית על ידי הוספת מספר קשתות זרימה דמה (באלגוריתם שלהם המבוסס על גרף זרימה) והממומשות על ידי Opaque predicates (פרק 4.1).

המימוש באלגוריתם CT לא מנסה לקשור את גרף טביעת האצבע לתכנית, ולכן ייתכן שהאלגוריתם חשוף להתקפות אלה.

ניתן להשתמש בטכניקה שהציע Venkatesan גם באלגוריתם CT אך ישנן שיטות קלות יותר וחשאיות יותר משיטה זו, למשל מכיוון שמבנה גרף סימן המים מוכר בכל התוכנית ניתן להשתמש בו כמקור לערכים של Opaque predicates. לדוגמא, המספר השלם 3 בתוכנית ניתן להחלפה באמצעות חישוב שמשתמש בגרף סימן המים כקלט לחישוב הערך 3, למשל על ידי פונקציה של אורך המסלול מהשורש לעלה מסוים.

6.3 הערכה של האלגוריתם (שבוצעה על ידי Collberg ו-Thomborson)

אין שיטה מקובלת ומוסכמת למדוד את החוזק של אלגוריתם להטבעת סימן מים בתוכנה. כתוצאה מכך, רוב הפרסומים בתחום מכילים מעט או לא מכילים כלל הערכות אמפיריות (מניסויים) או תיאורטיות. בעוד שמדידת יחס הנתונים של ההטמעה של סימן המים הוא יחסית ברור ופשוט, מדידת החשאיות וכושר ההתאוששות והעמידות היא קשה הרבה יותר, מאחר ומדידה שכזו דורשת מודל שבו יוצג איך התוקף עלול להעריך ולתקוף את התוכנית שהוטמע בה סימן המים.

חלק זה כולל שיטות הערכה בסיסיות שטרם הבשילו עבור חשאיות, יחס נתונים, וכושר עמידה והתאוששות. יש עדיין צורך לאשרר שיטות אלה אך הן עדיין בגדר שיפור ניכר לעומת ניסיונות העבר ותהוונה בסיס מוצק לעבודות עתידיות.

6.3.1 חשאיות

אפשר להגדיר חשאיות בהרבה מאוד דרכים. ב**איור 17** – **הגדרה** – **חשאיות מקומית** מוגדר בצורה פורמלית הקשר שבין חשאיות וכושר עמידות. בצורה לא פורמלית, אלגוריתם הוא בעל דרגה גבוהה של **חשאיות מקומית** אם, בהינתן גישה לאלגוריתם של סימן מים A ותוכנית P_w שידוע כי הוטמע בה סימן מים באמצעות אלגוריתם A, תוקף לא יכול למצוא את המיקום של w בתוך P_w.

ניתן לחשוב על **חשאיות מקומית סטטית** אם התוקף לא יכול לזהות את הקוד הסטטי שמיצר את סימן המים הדינמי. **חשאיות מקומית דינמית** יכולה לרמוז שהתוקף אינו יכול להבחין בין פריטי המידע של התוכנית הקשורים בטביעת האצבע לבין אלו שלא.

למען הבהירות והפשטות, את הביטוי "המיקום של w" בהגדרות הללו יש צורך להבין כ"המיקום של הקוד שבונה את w" במקרה של סימן מים דינמי בתוכנה.

חשאיות מתקשרת עם יחס נתונים כמו גם עם כושר התאוששות ועמידות, לדוגמא תבנית של סימן מים A יכולה לאפשר שיבוץ חשאי של סימן מים של 4 ביטים אבל לא לסימן מים של 40 ביטים בתוכנית נתונה P. כלומר קיימת תלות בין גודל טביעת האצבע לבין החשאיות. ניתן לראות זאת בהגדרות הפורמליות שתובאנה בהמשך.

<u>הגדרה</u> : חשאיות מקומית

A – אלגוריתם לסימן מים בתוכנה.
L – פונקציית איתור של התוקף.
m – אורך, בביטים, של סימן המים w.
w – סימן מים שנבחר אקראית מהתפלגות אחידה מ $0...2^m-1$.

U – קבוצת התוכנית הלא מסומנות (כבנצ׳מארק).

P – תוכנית שנבחרה אקראית מ-U.

λ – מספר ממשי מהתחום [0...1] , קבוע.

פונקציית האיתור של התוקף $L: \rightarrow \{0,1\}^{|X|}$ מיועדת לסמן את ההוראות והנתונים הקבועים בתוכנית מסומנת X. תהי 'L פונקציית איתור ללא שגיאות, כך ש- L'(x)[i] = 1 אם ההוראה ה-i של X בונה, משנה או מטמיעה סימן מים, ו -L'(x)[i] = 0 אחרת. נאמר ש-**A הוא אלגוריתם חשאי λ מקומי עבור L** אם שיעור השגיאות מסוג "תוצאה חיובית שגויה" (False positive) ו״תוצאה שלילית שגויה״ (False negative) הוא לפחות λ על הוראה j שנבחרה בצורה אקראית אחידה מקלט אקראי X = A(P, w) :

$$\max\left\{\begin{array}{l} Prob\left(\hat{L}(X)[j]=0 \wedge L(X)[j]=1\right), \\ Prob\left(\hat{L}(X)[j]=1 \wedge L(X)[j]=0\right) \end{array}\right\} \geq \lambda$$

איור 17 – הגדרה – חשאיות מקומית

ההגדרה העיקרית לחשאיות כאן מנוסחת פורמלית ב**איור 18 – הגדרה – חשאיות סטנוגרפית**. בצורה לא פורמלית, נאמר שתבנית של סימן מים היא בעלת **חשאיות סטנוגרפית** מדרגה גבוהה, אם כאשר בהינתן גישה לאלגוריתם של סימן מים A ותוכנית שעברה הטמעה של סימן מים P_W, תוקף לא יכול להחליט אם ב- P_W הוטמע סימן מים באמצעות A או לא.

<u>הגדרה</u> : חשאיות סטנוגרפית

תהי S פונקציית גילוי של תוקף, הממפה תכניות ל-0-1 (כאשר 1 מציין נוכחות של סימן מים בתוכנית) ותהי σ מספר ממשי קבוע מהתחום [0...1]. אם A, U, P, m, w כמו שהוגדרו לעיל, נאמר ש – **A הוא אלגוריתם חשאי סטנוגרפי** σ **עבור S** אם שיעור השגיאות מסוג "תוצאה חיובית שגויה״ (False positive) ו״תוצאה שלילית שגויה״ (False negative) של S הוא לפחות σ :

$$\max\{Prob(S(P)=1),\ Prob(S(A(P,w))=0)\} \geq \sigma$$

איור 18 – הגדרה – חשאיות סטנוגרפית

לדוגמא, אם σ נבחרה להיות 0.5, אזי A (אלגוריתם לסימן מים בתוכנה) הוא חשאי סטנוגרפי עבור S (פונקציית הגילוי של התוקף), אם ההסתברות המקסימליות של S לטעויות "תוצאה חיובית שגויה" (False positive) על תוכנית P (לא מסומנת) ו"תוצאה שלילית שגויה" (False Negative) על תוכנית מסומנת (על ידי A, עם w) גדולה מ-0.5.

כל אלגוריתם לסימן מים נתון ייתן תוצאה שונה בהסתמך על אחת מההגדרות הללו כתלות בתוקף שיבחר. לדוגמא, ניתן לדמיין במקרה של תרחיש גרוע ביותר בו התוקף מנסה לתקוף את תכונת **החשאיות המקומית** ומבצע דה-קומפילציה לתכנית, קורא ולומד את כל התכנית בכדי לאתר את הקוד הקשור לטביעת האצבע או על תרחיש "קל" יותר בו התוקף מפעיל התקפה סטטיסטית סטטית ויוריסטיקות פשוטות המנסות לאתר האם בתוכנית מוטמע קוד לסימן מים או לאו (**חשאיות סטנוגרפית**). תוצאות הניסויים שמובאות בהמשך מניחות את התרחיש השני, הקל.

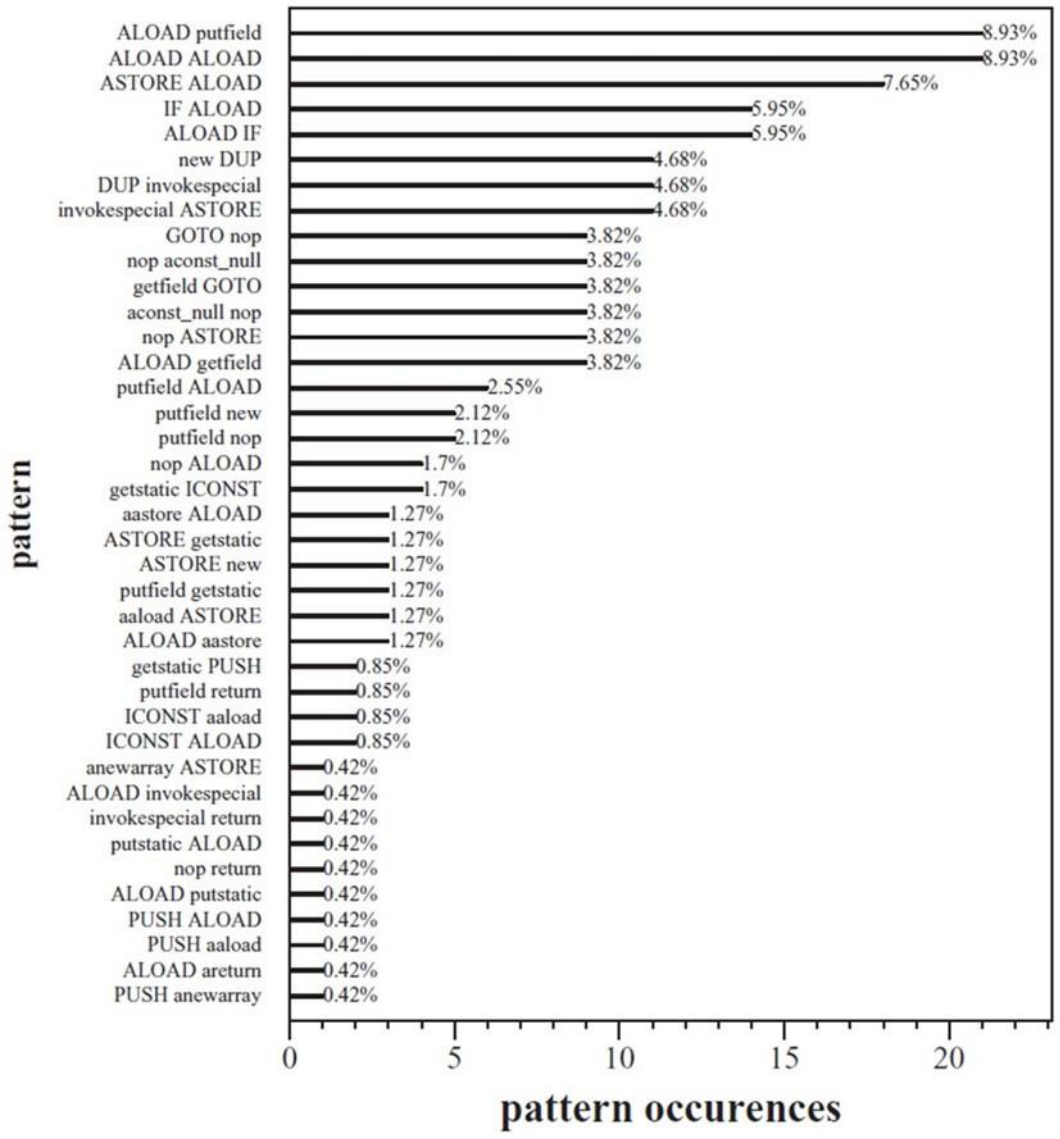
מבנה הניסויים

הניסויים נערכו על 622 תכניות Java (קבצי jar) שנאספו באינטרנט. גודל התוכניות נא בין 6 ל-40858 מתודות, מספר ההוראות בהם נע בין 21 ל-508,806, כאשר הממוצע עמד על 13,600 והחציון על 3,676 הוראות. התוכניות נאספו ממגוון מקורות, נכתבו על ידי מספר רב של תכניתנים שונים, חלקם היו אפליקציות וחלקם יישומי applets (יישומים קטנים בד"כ הרצים בדפדפן). ההנחה שמדגם שכזה הוא אקראי ומייצג של תכניות java.

התוכניות עברו סריקה של ההוראות בקוד הבינארי שלהם (bytecode instructions [22]) בחלונות סריקה בגודל n (n = 1,2,3,4) , כלומר רצף של 1,2,3 או 4 פקודות. ונבדקו ותועדו התדירויות של מופעי ההוראות השונים בתוכנית, תוך שימוש במחלקות אחידות בכדי להפחית אנומליות בין התוכניות שמקורם בכתיבה שונה, שימוש במהדרים שונים וכו'.

לאחר מכן, יצרו מחלקות סימן מים – $W_{4,3}$, $W_{16,3}$, $W_{32,3}$, $W_{64,3}$ מסוג קידוד בסיס (Radix Graph) בגדלים 4, 16, 32, 64 ביטים, והמחולקים ל-3 רכיבים (תתי גרף).

איור 19 – דיאגרמת שכיחויות עבור סימן מים 32 סיביות ב-CT, מתאר את שכיחויות ההוראות בקוד הבינארי עבור $W_{32,3}$ בגודל חלון n=2, ניתן לראות כי ההוראות aload (טעינת רפרנס למחסנית ממשתנה לוקאלי) ו-astore (שמירת רפרנס במשתנה לוקאלי) היו השכיחות ביותר בבניית גרף זה.



איור 19 – דיאגרמת שכיחויות עבור סימן מים 32 סיביות ב-CT

בניתוח החשאיות הסטנוגרפית שיובא להלן מתעלמים משכיחויות המופעים. במקום זאת, מניחים כי התוקף מבצע את חישוביו על סמך גודל חלון הסריקה בקבצי ה-jar בגודל של n <= 4. בצורה פרטנית, נחשוף בפני התוקף (או נניח שהוא למד את זה בדרך כלשהיא) סטים של $W_{m,n,3}$ של מופעי קוד בקוד שנבנה לבניית הגרף כאשר $W_{m,3}$ עבור m = {4, 16, 32, 64} עבור חלונות סריקה בגודל n <=4. התוקף יחשב את הדמיון בין הסטים הללו והמופעים שקיבל בחלונות הסריקה עבור תכנית X כלשהיא בה הוטמע סימן המים. אם כמעט כל הסטים הללו הופיעו ב-X, התוקף יסיק שסביר ש-X היא תוכנית שהוטמע בה סימן מים. בכדי לחשב את החשאיות כנגד תוקף שכזה, נבנו מספר גדול של ערכי נתונים $P_{m,n,i}$ של מספר המופעים של $W_{m,3}$ שהופיעו בסריקה על תכניות הג׳אווה שעברו הטמעה של סימן

מים. סה״כ נבנו 9952 = 4 x 4 x 622 ערכי נתונים כאלה, כל אחד מהם הוא איחוד של תוצאות של מספר המופעים שהתקבלו כאשר בוצעה סריקה על אחת מ-622 התוכניות P_i וכאשר ניתנו מראש גודל חלון הסריקה n = 1, 2, 3, 4 וגודל טביעת האצבע m = 4, 16, 32, 64 ביטים.

לדוגמא: $P_{16,2,5}$ הוא איחוד של מספרי המופעים של חלונות סריקה ב- $W_{16,3}$ (סימן מים W, הוא מספר בגודל של 16 ביטים, ומיוצג בגרף בסיס ללא מעגלים שפוצל ל-3 רכיבים) והמופיעים בתוכנית החמישית (מתוך 622) כאשר גודל חלון הסריקה שהיה בשימוש הוא 2.

בסימולציה זו לא נלקחה בחשבון האפשרות כי בדרך כלל במהלך ו/או לאחר השיבוץ ניתן לבצע ערפול לקוד ובכך להוריד את רמת הדיוק של פונקציית הגילוי של התוקף.

במילים פשוטות יותר, Collberg ו -Thomborson ניסו לחפש התאמות (מסוגים שונים) להמצאות סימן המים (בגדלים שונים) בקוד של תוכניות המדגם לאחר שעברו הטמעה של סימן מים באמצעות האלגוריתם שלהם.

אנליזה של החשאיות הסטנוגרפית מניסויים

החשאיות הסטנוגרפית מחושבת באופן יחסי לפונקציית הגילוי של התוקף B. פונקציה זו תיתן ערך מקסימלי כאשר הקוד X מכיל את כל חלונות הסריקה ביחס לסט $W_{m,n,3}$. ההגדרה הפורמלית של B היא:

$$B_{m,n,\delta}(X) = \begin{cases} 1, & \text{if } \frac{|\text{window types that occur in } W_{m,n,3} \text{ but not in } X|}{|\text{window types that occur in } W_{m,n,3}|} < \delta \\ 0, & \text{otherwise.} \end{cases}$$

הפרמטרים שמקבלת B הם m – גודל טביעת האצבע (4,16,32,64 - ביט), n – גודל חלון הסריקה (n<=4), ופרמטר רגישות δ ($0 < \delta < 1$).

<u>דוגמא</u> - אם נבחר את $\delta = 0.4$ ומספר הסוגים של חלונות סריקה על $W_{m,n,3}$ הוא 10 (על m,n – נתונים) ובתוכנית X שנסרקה באותו גודל חלון סריקה מספר הסוגים של החלונות שנמצאו ונמצאו גם

ב-$W_{m,n,3}$ הוא 7 אזי 0.3 = 10/(10-7) ומכיוון ש-0.4 > 0.3 הפונקציה B תחזיר 1, כלומר התוכנית עברה הטמעה של סימן מים. אך אם מספר הסוגים של החלונות שנמצאו ונמצאו גם ב-$W_{m,n,3}$ הוא 5 אזי 0.5 = 10/(10-5) ומכיוון ש-0.4 < 0.5 הפונקציה B תחזיר 0, כלומר התוכנית לא עברה הטמעה של סימן מים. מכאן ניתן להבין את תפקידו של פרמטר הרגישות.

נצפה שהתוקף יבחר n כך שימקסם את הדיוק של פונקציית הגילוי. נניח שהתוקף יבחר את m בהתאם ובהסתמך על ידע קודם. התוקף יבחר δ מתאימה בהתבסס על מידת הטולרנטיות שלו לטעויות מסוג "תוצאה חיובית שגויה" (False positive) ו"תוצאה שלילית שגויה (False negative) , אם δ קרובה ל-1, פונקציית הגילוי תסבול מטעויות מסוג "תוצאה חיובית שגויה", כלומר היא תדווח על תכניות שבהם לא הוטמע סימן מים כאילו כן הוטמע בהם סימן שכזה. אם δ קרובה ל-0, פונקציית הגילוי תסבול מטעויות מסוג "תוצאה שלילית שגויה", כלומר היא לא תדווח על תכניות שבהם כן הוטמע סימן מים.

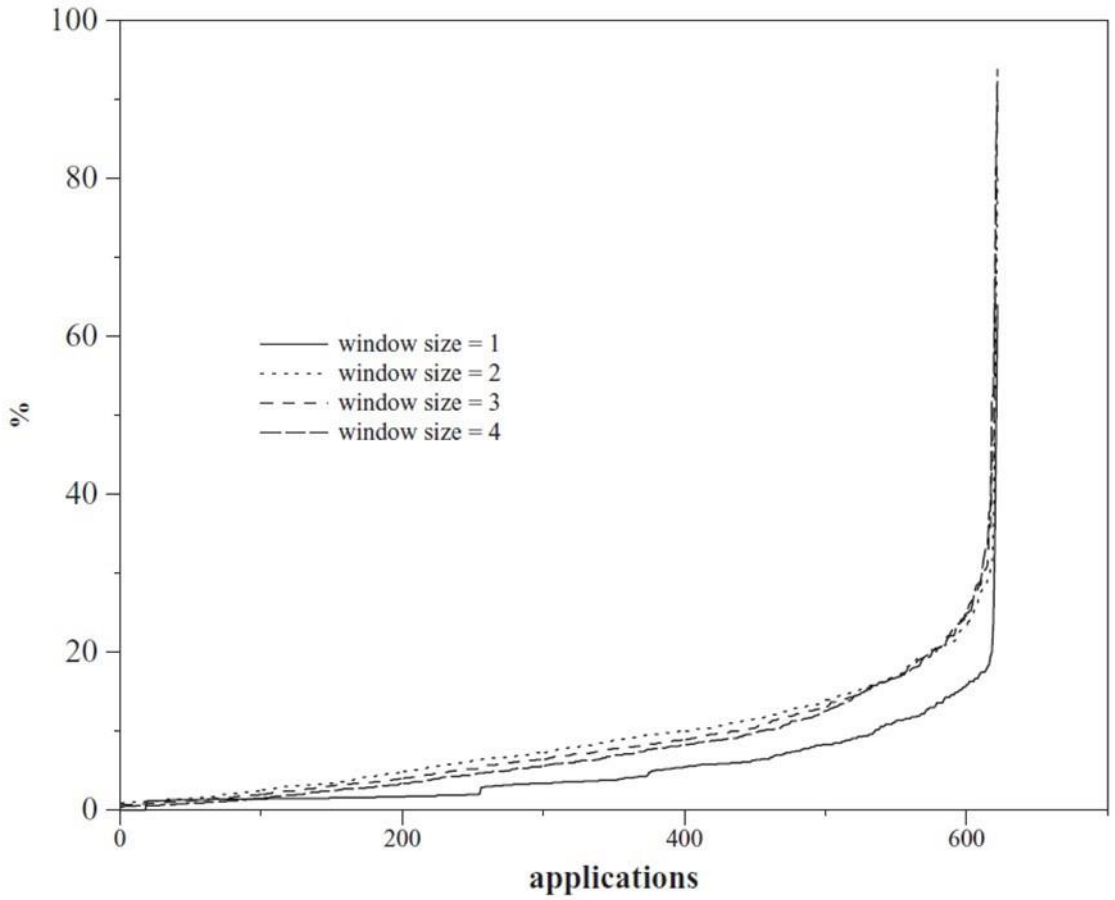
נניח כי התוקף לא ירצה לקבל שגיאות מסוג "תוצאה שלילית שגויה", זאת מכיוון שזה יהיה לו יקר מדי להיתפס בהפצת תכניות המכילות סימן מים של אחרים. מנגד, טעויות מסוג "תוצאה חיובית שגויה", סביר שלא יהיו יקרות לתוקף. בתגובה לשגיאות מסוג "תוצאה שלילית שגויה", התוקף ימשיך לבצע מתקפות על X עד אשר יצור 'X אשר עבורה B(X') = 0. נניח שאין באפשרות התוקף להעריך את שיעורי הטעויות מסוג "תוצאה חיובית שגויה" בצורה מדויקת שכן לשם כך עליו לדעת בצורה טובה מספר רב של פרמטרים שקשה מאוד להעריכם כמו האיחוד U_1 של הקוד שסביר שיהיה הקלט לסימן מים לשיבוץ.

בניסויים שנערכו נקבע ערך רגישות נמוך $\delta = 0.1$, וזאת בכדי לקבל שיעור נמוך של שגיאות מסוג "תוצאה שלילית שגויה".

יש לשים לב כי B תדווח על X ככזו שמוטמעת בה סימן מים ($B_m, \delta (X) = 1$) אם כל סוגי החלונות ב-$W_{m,3}$ הופיעו גם ב-X. ו-B תדווח על X ככזו שלא מוטמעת בה סימן מים ($B_m, \delta (X) = 0$) אם אף אחד מסוגי החלונות ב-$W_{m,3}$ לא הופיע גם ב-X.

התרשים ב**איור 20** – **דיאגרמת הערכה של חשאיות סטנוגרפית**, מראה ברגישות של $\delta = 0.1$ (ציר y), עבור סימן מים בגודל 32 ביט כי 395 תכניות מתוך 622 שנדגמו נמצאו כחשאיות סטנוגרפית.

המסקנה היא שאלגוריתם CT להטבעת סימן מים בתוכנה בצורתו זו, מתאים לשימוש (עבור תוקף כמתואר לעיל) בכמחצית מתוכניות הכתובת בג׳אווה שפורסמו ברשת בתקופת עריכת הניסוי.



Steganographic stealth evaluation. For each of the 622 applications in our benchmark universe, the graph shows the fraction of *n*-grams which occur in the embedded 32-bit CT fingerprint code, but which do not occur in the application itself.

איור 20 – דיאגרמת הערכה של חשאיות סטנוגרפית

6.3.2 יחס נתונים

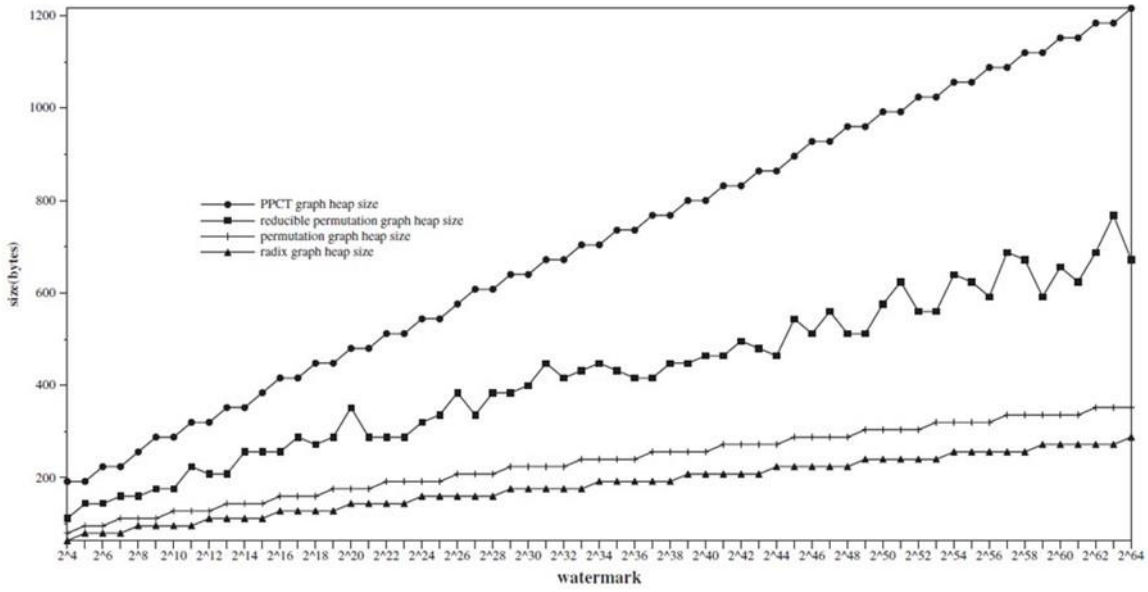
איור 21 – גודל הגרף בזמן הריצה כתלות בגודל טביעת האצבע, לפי סוג גרף, מתאר גודל בזמן ריצה של גרפים במבני נתונים שונים כתלות בגודל סימן המים.

נעשה שימוש ברגרסיה לינארית על מנת להעריך את הפרמטרים a, b במשוואה S(m) = am + b בכל אחת משיטות הקידוד. כאן m הוא מספר הביטים במספר שלם w המייצג את טביעת האצבע, כלומר $m = \lceil \lg(w+1) \rceil$ כאשר w הוא כל מספר שלם לא שלילי.

האנליזות שבוצעו עבור שאר המבנים מראות שהניתוח התיאורטי שבוצע (פרק 6.1) היה מדויק. התוצאות מראות פונקציה עולה מתונה, הניסויים הראו שהעיוותים בלינאריות הם קטנים וכמעט ולא ניכרים בתרשימים.

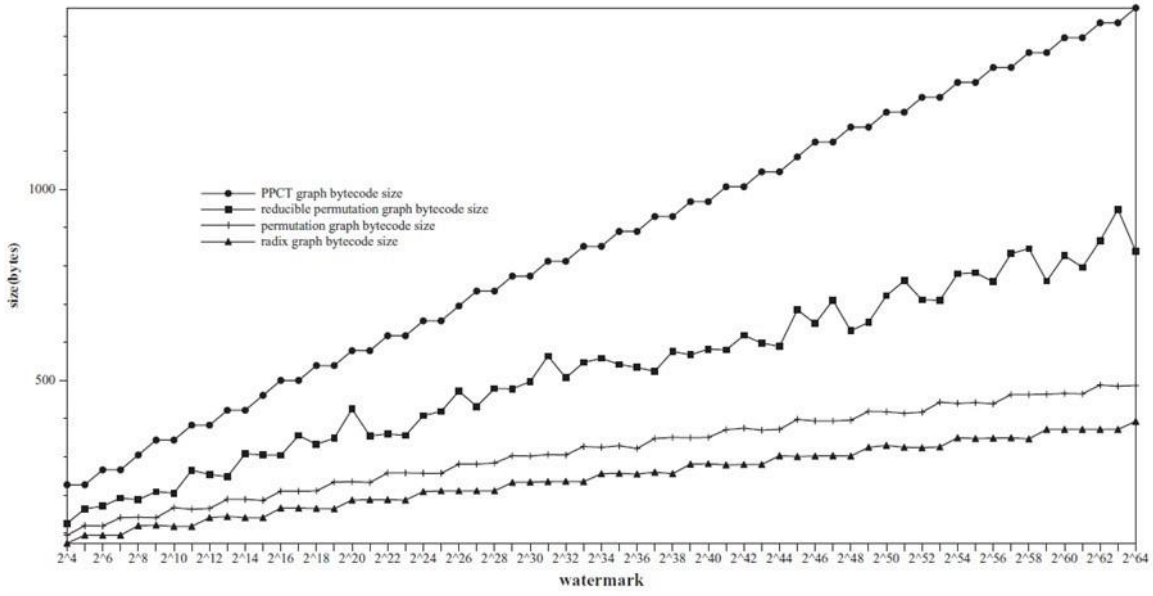
קווי הרגרסיה שהתקבלו עבור קידוד תמורה (Permutation Graphs) וקידוד בסיס (Radix Graphs) הם דומים, אך לקידוד בסיס היה יחס נתונים מעט גבוה יותר. $S_{PG}(m) = 4.4m + 84$ ו- $S_{RG}(m) = 3.4m + 67$, $R^2 = 0.991$ לשניהם.

באף אחד מהניסויים לא נדרש יותר מ-1.3 KB של הקצאת זיכרון דינמית על מנת לשבץ סימן מים בגודל 64 ביט.



איור 21 – גודל הגרף בזמן הריצה כתלות בגודל טביעת האצבע, לפי סוג גרף

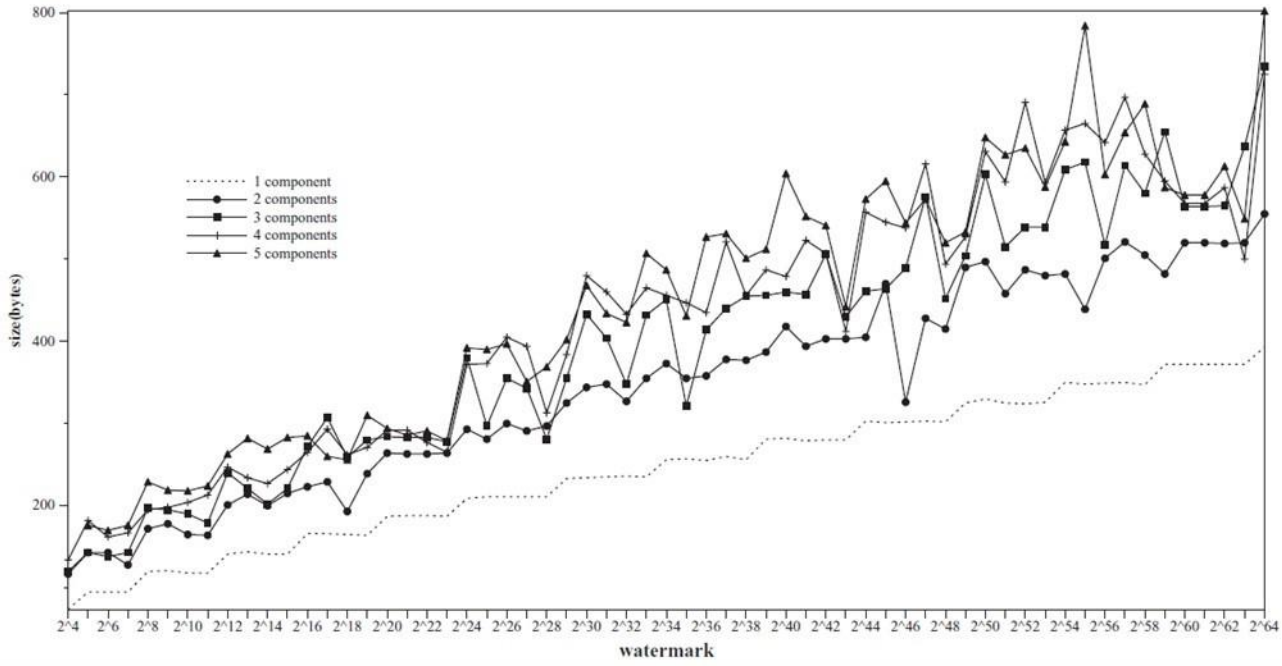
איור 22 – גודל הגרף (בקוד) כתלות בגודל טביעת האצבע, לפי סוג גרף, מראה את גודל התוכנית (bytecode) עבור כל אחד משיטות הקידוד. כמצופה יחס הנתונים הסטטי שהתקבל פרופורציונאלי ליחס הנתונים הדינאמי., 6.3 בתים של קוד פר ביט בטביעת האצבע עבור קידוד תמורה, ו- 4.9 בתים עבור קידוד בסיס (21 בתים עבור מבנה נתונים מסוג PPTC, ו- 12 בתים עבור RPG). קווי הרגרסיה הכילו תקורה נוספת של בין 77 ל-150 בתי קוד.



איור 22 – גודל הגרף (בקוד) כתלות בגודל טביעת האצבע, לפי סוג גרף

איור 23 – גודל גרף בסיס (בקוד) כתלות במספר הרכיבים אליהם פוצל, עבור סימן מים בגדלים שונים, מראה את גודל הקוד כפונקציה של k, מספר הרכיבים אליהם פוצל הגרף. מהניסויים והניתוח הסטטיסטי עולה כי בכלליות הגודל עולה בממוצע ב- 22(k-1) % , לדוגמא, 4.9 בתי קוד פר ביט סימן מים עבור קידוד בסיס הופכים להיות 6 בתי קוד לביט כאשר הגרף מפוצל ל-2 רכיבים. ניתן להסביר עליה זאת בקלות, ככל שהגרף מפוצל ליותר רכיבים כך נדרש לייצר יותר קוד על מנת למזג רכיבים אלה יחד. סביר שהטבעת סימן מים של גרף המפוצל להרבה רכיבים תהיה יותר חשאית מאשר גרף שאינו מפוצל (יחידה אחת), אולם אם k הוא מספר גדול מאד כמות הקוד שתיווצר עלולה להיות גדולה מספיק כך שתוקף עלול להצליח לזהות תבניות החוזרות על עצמם.

כל התצפיות שנעשו אישרו שהתקורה של כמות הקוד הסטטי והדינמי בהטבעת טביעת האצבע הוא קטן, ובמידה מתקבלת על הדעת עבור רוב התכניות.

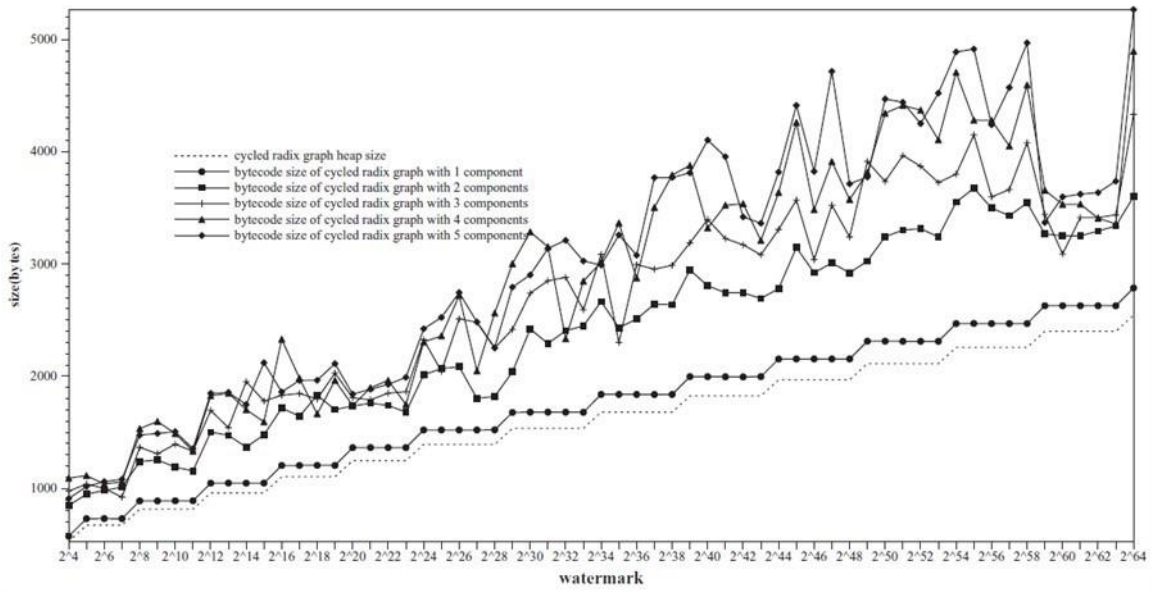


איור 23 – גודל גרף בסיס (בקוד) כתלות במספר הרכיבים אליהם פוצל, עבור סימן מים בגדלים שונים

6.3.3 כושר התאוששות ועמידות

התקפות טרנספורמציות על הקוד והנתונים כמו מיזוג מתודות, פיצול מחלקות, פיצול מערכים, שינוי חתימות של מתודות, הפיכת סקלרים לעצמים וכו'. אינן יכולות למנוע חילוץ של סימן מים שהוטמע על ידי אלגוריתם CT. בנוסף, שימוש בגרפים מעגליים יעיל כנגד התקפות כאלה (למעט הטרנספורמציה של פיצול צמתים, פרק 6.2.1). אולם, כפי שמתואר ב**איור 24 – גודל התכנית וגודל הערימה בזמן ריצה עבור גרפי בסיס מעגליים, עבור סימן מים בגדלים שונים**, שיפור עמידות שכזה מגיע עם עלות משמעותית – יחס הנתונים הדינמי יורד בערך בחצי.

באמצעות SpecJVM (Java Virtual Machine Benchmark) בוצעה בדיקה של העמידות להתקפה של פיצול צמתים. התוצאות מראות שעבור תכניות רבות תוקף יכול לבצע בקלות פיצול אחד או שניים של צמתים בלי לדאוג לירידה בביצועים. התוצאות מראות גם שהשונות בין פיצולים שונים היא גבוהה. למשל ביצוע שני פיצולי צמתים יגרום לפקודה _228_jack להיות איטית ב-16% ול-_227_mtrt להיות איטית יותר ב-286%. לכן, ניתן להסיק כי לתכניות בהם הביצועים אינם קריטיים, במקרים רבים יהיה כדאי להשתמש בגרפים מעגליים, וזאת בתנאי שיחס הנתונים הנמוך שיתקבל יהיה נסבל.



איור 24 – גודל התכנית וגודל הערימה בזמן ריצה עבור גרפי בסיס מעגליים, עבור סימן מים בגדלים שונים

לגבי אופטימייזרים של מהדרים (Compilers), באופן כללי ניתן לומר כי טרנספורמציות של מהדר טיפוסי הם חסרות משמעות להתקפות כנגד סימן מים שהוטמעה בעזרת אלגוריתם CT, למעט אם המהדר מגלה (באמצעות אנליזה סטטית) כי הקוד שהכנסנו הוא קוד יתיר (redundant) ומסיר אותו.

7. דיון ומסקנות

מההיבט הפרקטי, השאלה החשובה ביותר היא מהו מודל האיום הסביר. במאמרים שנסקרו זוהו מספר סוגים של מודלי איום:

1. התקפות עיוות – המשתמשות בשיטות שונות לשינוי סימן המים בתוכנית.
2. התקפה סטטיסטית על תכונת החשאיות המקומית של טביעת האצבע, על ידי ניסיון לאתר את טביעת האצבע באמצעות זיהוי אנומליות בפיזור של הוראות או חישובים.
3. התקפת התאמה – על תכונת החשאיות המקומית של טביעת האצבע, בניסיון לאתרה באמצעות השוואות של מספר עותקים שונים של תוכנית בה הוטמע סימן מים.
4. התקפות חיסור – ניסיון להסיר סימן מים שאינו חשאי.
5. התקפות חיבור - על ידי הכנסת סימן מים מזויף לתוכנית בה הוטמע סימן מים.

אף אחת משיטות ההטמעה שהוצגו אינה חסינה מפני כל סוגי המתקפות הללו.
Easter Egg Fingerprints וסימן מים בגרף דינמי מראות עמידות גבוהה בפני התקפות עיוות, אבל מטבען הם מוטמעות בתוכנית שלמה ולא במודולים עצמאיים. כתוצאה מכך, חיתוך של מודול בעל ערך מהתוכנית לצורך שימוש לא חוקי יכול להוות התקפה מוצלחת כנגד שיטות אלו.
מנגד, שיטות סטטיות מאפשרות להעתיק בקלות את טביעת האצבע למודולים השונים, אך למרבה הצער שיטות אלה אינם עמידות להתקפות עיוות.
הצלחה של התקפה סטטיסטית תהיה תלויה באופי טביעת האצבע ובטבעה של התכנית. טביעות אצבע בגרף דינמי הינן חשאיות בתוכניות מונחות עצמים טיפוסיות אשר נוטות ליצור מבנים מורכבים וגדולים בזיכרון הערימה. הן תהינה מאוד לא חשאיות, וכתוצאה מכך חשופות להתקפות סטטיסטיות בתוכניות שמטבען הם בעיקר מספריות (מבצעות חישובים אריתמטיים), והאנליזה שהוצגה כאן מראה שהמימוש הבסיסי של אלגוריתם CT אינו חשאי בערך במחצית מתוכניות הג׳אווה שהופצו באינטרנט (בתקופת הבדיקה).
מעניין לשים לב שהבעיות שיש להתמודד איתם בהטבעת סימן מים בתוכנה לעיתים תכופות שונות מאלו שמתמודדים איתם בהטבעת סימן מים במדיה. הסיבה לכך היא תכונת הזרימה של התוכנה, אשר מאפשרת לנו לשנות את הטקסט של התכנית מבלי לשנות את התנהגותה. לדוגמא, קשה יחסית להגן מפני התקפת התאמה בסימן מים המוטמע בתמונה, מאחר שמטבעם כל העותקים של תמונה בה הוטמע סימן מים חייבים להראות זהים. הטבעת סימן מים בתוכנה לא צריכה להתמודד עם בעיה שכזו. ניתן להגן בקלות מפני התקפת התאמה על ידי החלת טרנספורמציות ערפול על כל עותק של התוכנית שמופץ, כך שלא סביר שהשוואה של מספר עותקים של תכנית שהוטמע בה סימן מים תחשוף את מיקום טביעת האצבע, מאחר והטקסט של כל תוכנית יראה שונה לחלוטין. מסיבות דומות, התקפות עיוות מהוות איום נמוך יותר עבור הטבעת סימן מים במדיה מאשר בהטבעת סימן מים בתכנה. התקפת עיוות על מדיה היא מוגבלת שכן השינויים שהיא יכולה לבצע מוגבלים וצריכים להיות בלתי נראים, בעוד שמתקפת ערפול על תכנית מוגבלת רק בצורך לשמור על הסמנטיקה של התכנית.

8. תוצאות ומסקנות ממימוש האלגוריתם בפרויקט המסכם

הפרויקט המסכם שבצעתי, מימש את אלגוריתם CT מתחילתו ועד סופו והדגים אותו על תכנית אמיתית. התשתית שיצרתי בפרויקט יכולה לשמש כאמצעי להזנת סימן מים בתוכניות הכתובות ב- C# Net.. הרעיון שבבסיס הפרויקט היה לבנות יישום כללי, מעין כלי תשתיתי, כך שבקלות ובצורה אוטומטית ניתן יהיה לבנות ולהטמיע, ולאחר מכן לחלץ את סימן המים ל/מכל תכנית מארחת (כמובן שרק עבור תכניות שנבנו בטכנולוגיה שנבחרה למימוש כאן, Microsoft .Net).

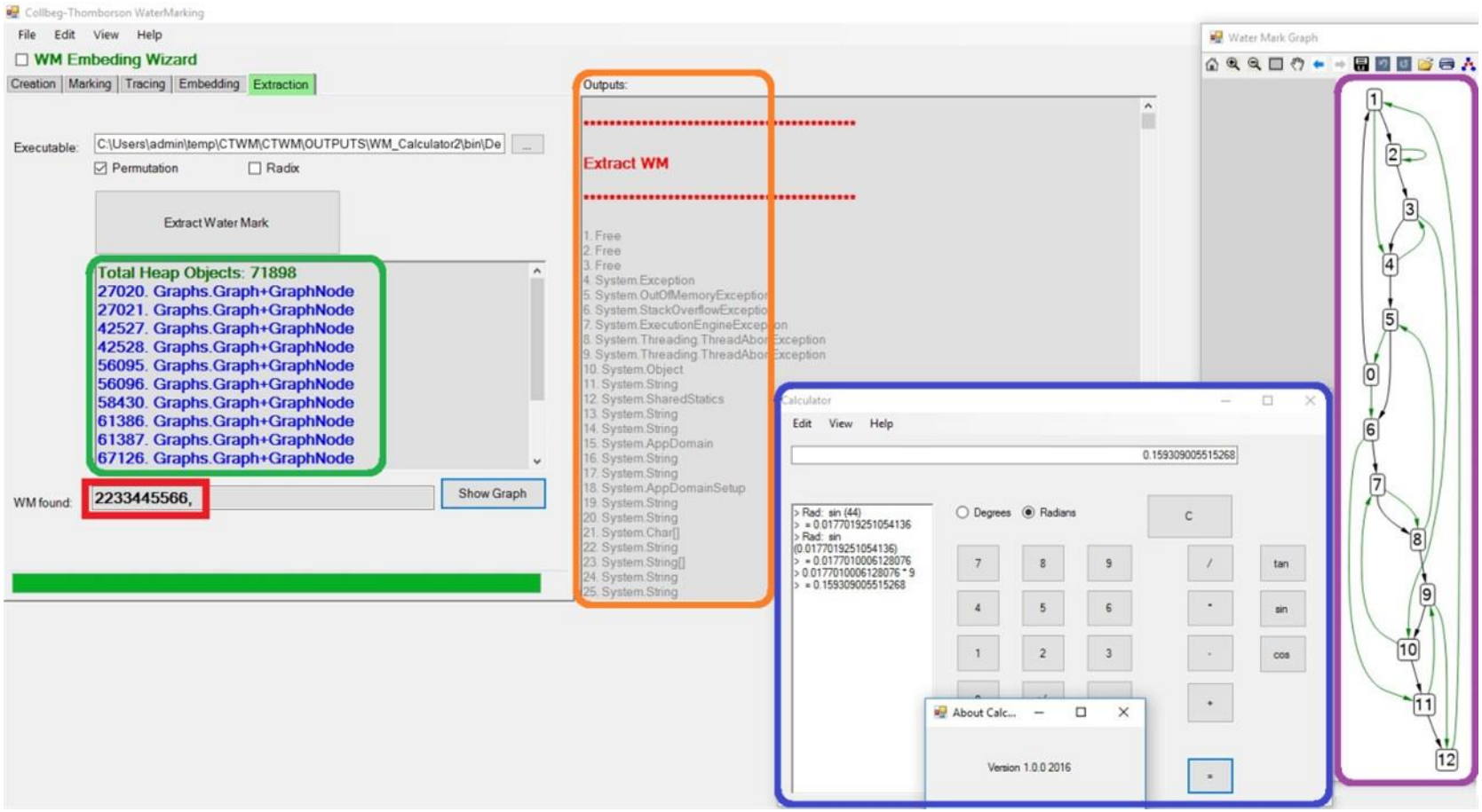
המימוש עובד היטב ומוצא תוצאות, בכל המקרים שנבדקו. את הבדיקות בצעתי על תוכנית לדוגמא שכתבתי והממשת מחשבון פשוט. האלגוריתם והמימוש אינם נטולי בעיות, הבעיות העיקריות שהתגלו בזמן מימוש :

- סימן המים אינו חבוי, ניתן לראות את הקריאות ליצירתו באמצעות ILDASM (Intermediate Language disassembler), ניתן להתגבר על כך על ידי ביצוע ערפול (obfuscation). כמו כן ניתן להכניס דמויים (המימוש אינו טריוויאלי), כלומר קריאות שאינן אמתיות ליצירת הגרף, או כאלו שיגרמו לגרף להיווצר בצורה שאינה תקינה כאשר מוזן קלט שאינו תקין.
- בבחירת המקומות בקוד בהם יש לבצע פעולות הקשורות ליצירת הגרף, יש לבחור מקומות בהם התכנית עברה פעם אחת ויחידה במהלך הזנת הקלט הסודי, אחרת יש צורך בסימון מיוחד המציין את מספר הביקורים באותה שורה בתכנית בזמן ריצה, דבר המסבך מאוד את הטמעת סימן המים ובנוסף פעולה חריגה זו עלולה לעורר חשד אצל תוקף שעלול לזהות כי נקודה זו קשורה ליצירת סימן המים.
- סדר הזנת הקלט אינו משנה במימוש שבצעתי, לצורך העניין אם הקלט הסודי הוא "3 + 8 * 5 =" אזי גם הקלט "3 + 8 * 5 =" יעבוד.
- אם הקלט הסודי שהוזן הוא תת קבוצה של הקלט שנבדק אזי גם ייתכן שייבנה הגרף המתאים בזיכרון הערימה וייתכן שיחולץ סימן המים.
- נצפו מקרים בהם חולץ יותר מגרף אחד שמתאים לקידוד ולכן, במקרים כאלו הוצגו כל סימני המים שנמצאו. תופעה זו מתרחשת לרוב כאשר סימן המים הוא מספר קטן.
- זמני הריצה בכל שלבי השיבוץ הם זניחים, ועולים מעט ככל שגודל התוכנית עולה.
- זמן הריצה בזמן החילוץ הוא משמעותי ותלוי בכמות העצמים שהוקצו בערימה בזמן ביצוע החילוץ ובעצמת המחשב המבצע. עבור תוכנית המחשבון, במחשב עם מעבד אינטל I7 וזיכרון של 8 gb ram משך החילוץ בסריקה של סה"כ 56,518 עצמים היה כ-30 :2 דקות (cpu 13%).

- גודל התוכנית בה מוטמע סימן המים גדל. בתוכנית שנבדקה, עבור סימן מים מסוג פרמוטציה, הייתה עליה של כ- 2% אחוזים בהידור בתצורת debug (מ 26,112 בתים ל -26,624 בתים), ובכ-4% בהידור בתצורת release (מ 24,576 בתים ל-25,600 בתים).
- ככל שסימן המים המוטמע גדול יותר התוכנית המוטמעת גדולה יותר. לדוגמא, הטמעת סימן המים "2233445566778899111" מסוג פרמוטציה בתצורת release הגדילה את תוכנית המחשבון, ביחס לדוגמא בסעיף 5 בכ- 2% (מ-25,600 בתים ל-26,112 בתים).

הטמעת סימן מים באמצעות קידוד בסיס יוצרת לרוב תוכנית מעט קטנה יותר מאשר פרמוטציה (עד כ-2% בתוכנית המחשבון).

איור 25 – חילוץ מוצלח של סימן המים מהתוכנית Calculator לאחר הזנת הקלט הסודי, (בעמוד הבא) מראה את ממשק המשתמש של התוכנית שמומשה בפרויקט המסכם, לאחר שביצעה חילוץ מוצלח של סימן המים בתוכנית המחשבון.



איור 25 – חילוץ מוצלח של סימן המים מהתוכנית Calculator לאחר הזנת הקלט הסודי

ניתן לראות את רשימת כל העצמים שחולצו מזיכרון הערימה (מוקפים במסגרת **כתומה**) של התוכנית Calculator (מוקפת במסגרת **כחולה**), את אלו מביניהם שמרכיבים את הגרף המייצג את סימן המים (מוקפים במסגרת **ירוקה**), את הגרף שחולץ (מוקף במסגרת **סגולה**) ואת סימן המים עצמו (מוקף במסגרת **אדומה**.

9. סיכום

הטבעת סימן מים בתוכנה משבצת ערך מזהה בתוכנית. טביעת האצבע האידיאלית ניתנת לחילוץ בקלות באמצעות מפתח אך ללא מפתח קשה מאוד לאתרה. הטבעה שכזו כרוכה בתוספת קטנה לקוד והיא צריכה להיות עמידה כנגד מגוון רחב של התמרות תוכנה והתקפות.

רוב השיטות להטבעת סימן מים בתכנה הם סטטיות: הם מוחלות על ומתגלות בקבצים הבינאריים שניתנים להרצה. בעבודה זו תוארו מספר שיטות ואלגוריתמים מסוגים שונים להטבעת סימן מים בתכנה, אך עיקרה הופנה לתיאור אלגוריתם CT, אשר בו מבוצעת הטבעת סימן מים בתוכנה בצורה דינאמית, כלומר סימן המים מקודד בצורה דינאמית בגרף מייצג אשר נבנה בזיכרון הערימה של התכנית במהלך ריצתה. בניית הגרף נגזרת מרצף של קלטים ייחודיים אשר בפועל משמשים כמפתח. סימן מים דינאמי קשה יותר לגילוי בהשוואה לסימן מים סטאטי ובנוסף הוא גם עמיד יותר כנגד התמרות כמו ערפול ואופטימיזציה.

אלגוריתם CT הוטמע בכלי SandMark בעיקר לצורך מחקר, ביחד עם מספר רב של כלים ועבור תכניות הכתובות בשפת ג'אווה. הכלי מספק מספר אפשרויות לאופן ביצוע האלגוריתם כולל בחירה של סוג הגרף באמצעותו יבוצע קידוד סימן המים.

מחקרים ותצפיות שבוצעו על ידי מפתחי האלגוריתם הראו שישנן פשרות בין סוג הגרף שנבחר לקידוד וגודל סימן המים והעריכו את השפעתם על גודל הקוד ודרישות הזיכרון. כמו כן נמדדה ההשפעה של האלגוריתם על זמן הריצה של תכניות ונמצא כי האלגוריתם פוגע בזמני הריצה ומגדיל אותם אך העלייה אינה משמעותית. בעבודה הוצג מודל שפותח על ידי מפתחי האלגוריתם למדידת החשאיות של סימן מים בתכנה ונמצא כי שהקוד המיוצר על ידי אלגוריתם CT הוא חשאי (עבור סימן המים) לאחוז ניכר מתכניות הג'אווה. האלגוריתם נמצא עמיד כנגד ערפול למעט כנגד פיצול צמתים, אך ניתן להתגבר על כך.

החולשה הגדולה ביותר של האלגוריתם בצורת המימוש שהוצגה קשורה לכך שחשאיותו של סימן המים מוגבלת. תוקפים מתוחכמים יכולים לגלות את סימן המים על ידי ניתוח מספר רב של תכניות שבהם הוטמע סימן מים.

10. רשימת מקורות

[1] A. Monden, H. Iida, et al. A watermarking method for computer programs.
In Proceedings of the 1998 Symposium on Cryptography and Information Security,
SCIS'98. Institute of Electronics, Information and Communication Engineers, January 1998.

[2] A. Monden, H. Iida, K. I. Matsumoto, K. Inoue, and K. Torii. Watermarking java programs.
In International Symposium on Future Software Technology '99, pages 119–124, October 1999.

[3] A. Monden, H. Iida, K. I. Matsumoto, K. Torii, and K. Inoue.
A practical method for watermarking java programs.
In COMPSAC '00: 24th International Computer Software and Applications
Conference, pages 191–197, Washington, DC, USA, 2000.

[4] J. Stern, G. Hachez, F. Koeune, and J. J. Quisquater.
Robust object watermarking: Application to code.
In Information Hiding Workshop'99, pages 368–378, 1999.

[5] R. Davidson and N. Myhrvold.
Method and system for generating and auditing a signature for a computer program,
June 1996. Microsoft Corporation,
US Patent 5559884.

[6] M. Shirali-Shahreza and S. Shirali-Shahreza.
Software watermarking by equation reordering.
In Proceedings of the 3rd International Conference on Information and Communication
Technologies: From Theory to Applications, 2008. ICTTA 2008, pages 1–4, 2008.

[7] D. Gong, F. Liu, B. Lu, P. Wang, and L. Ding.
Hiding information in java class file.
In Proceedings of the 2008 International Symposium on Computer Science and Computational
Technology - Volume 02, pages 160–164.

[8] G. Qu and M. Potkonjak.
Analysis of watermarking techniques for graph coloring problem.
In Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design,
pages 190–193, San Jose, California, United States, 1998.

[9] G. Myles and C. Collberg.
Software watermarking through register allocation: Implementation, analysis, and attacks.
In Proceedings of the International Conference on Information Security and Cryptology,
volume 2971/2004 of Lecture Notes in Computer Science.
Springer Berlin / Heidelberg, 2003

[10] W. Zhu and C. Thomborson.
Algorithms to watermark software through register allocation.
In Digital Rights Management. Technologies, Issues,
Challenges and Systems, volume 3919 of Lecture notes in computer science, pages 180–191,
Berlin, Allemange, 2006. Springer.

[11] Z. Jiang, R. Zhong, and B. Zheng.
A software watermarking method based on Public-Key cryptography and graph coloring.
In Proceedings of the 3rd International Conference on Genetic and Evolutionary Computing, 2009. WGEC '09 , pages 433–437.

[12] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap directed pointers in c.
In Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 1–15, St. Petersburg Beach, Florida, 1996.

[13] G. Ramalingam. The undecidability of aliasing. ACM Trans. Program. Lang. Syst., 16(5): 1467–1471, 1994.

[14] C. Collberg, S. Kobourov, E. Carter, and C. Thomborson.
Error-Correcting graphs for software watermarking.
In Proceedings of the 29th Workshop on Graph Theoretic Concepts in Computer Science, pages 156–167, 2003.

[15] R. Venkatesan, V. Vazirani, and S. Sinha.
A graph theoretic approach to software watermarking.
In Proceedings of the 4th International Workshop on Information Hiding, 2001.

[16] G. Arboit.
A method for watermarking java programs via opaque predicates.
In Proceedings of the Fifth International Conference on Electronic Commerce Research (ICECR-5), 2002.

[17] G. Myles and C. Collberg.
Software watermarking via opaque predicates: Implementation, analysis, and attacks. In ICECR-7, 2004.

[18] M. Dalla Preda,M. Madou, K. De Bosschere, and R. Giacobazzi.
Opaque predicates detection by abstract interpretation. In Michael Johnson and Varmo Vene, editors, AMAST, volume 4019 of Lecture Notes in Computer Science, pages 81–95. Springer, 2006.

[19] C. S. Collberg, C. Thomborson and G. M. Townsend, 2007. Dynamic graph-based software fingerprinting. ACM Trans. Program. Lang. Syst. 29, 6, Article 35 (October 2007),

[20] J. Hamilton and S. Danicic, A Survey of Static Software Watermarking. In proceedings of 2011 World Congress on Internet security (WorldCIS), , 100-107 (IEEE).

[21] C. S. Collberg, G. Myles and A. Huntwork. Sandmark—A Tool for Software Protection Research, IEEE Security & Privacy 2003, vol.1,Issue No.04 - July-August

[22] www.Wikipedia.org, https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings, https://en.wikipedia.org/wiki/Watermark

[23] Globalstudy.bsa.org/2016/ (BSA, The software alliance)
Seizing Opportunity Through License Compliance BSA GLOBAL SOFTWARE SURVEY May 2016

[24] www.eeggs.com

[25] J. Stern, G. Hachez, F. Koeune, and J.-J. Quisquater, "Robust object watermarking: Application to code," in Information HidingWorkshop '99, 1999, pp. 368–378. [Online]. Available: http://citeseer.ist.psu.edu/stern00robust.html

[26] D. Curran, M. O. Cinneide, N. Hurley, and G. Silvestre, "Dependency in software watermarking", In Proceedings of the First International Conference on Information and Communication Technologies: from Theory to Applications, 2004, pp. 311–324.

[27] P. Cousot and R. Cousot, "An abstract interpretation-based framework for software watermarking," in Principles of Programming Languages 2003, POPL'03, 2003, pp. 311–324.

[28] J. Nagra and C. Thomborson, "Threading software watermarks," in IH'04, 2004.

[29] W. Bender, D. Gruhl, N. Morimoyo and A. Lu. 1996. Techniques for data hiding. IBM Syst. J. 35, 3&4, 313–336.

[30] M. Madou, B. Anckaert, B .D. Sutter and K.D. Bosschere. 2005. Hybridstatic-dynamicattacks against software protection mechanisms. In DRM '05: In Proceedings of the 5th ACM Workshop on Digital Rights Management. ACM, New York, 75–82.
[31] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao and Y. Zhang. 2000. Experience with software watermarking. In Proceedings of ACSAC'00, 16th Annual Computer Security Applications Conference. 308–316. citeseer.nj.nec.com/323325.html.

[32] W. Myrvold and F. Ruskey. 2001. Ranking and unranking permutations in linear time. Inf.Proc. Lett. 79, 6 (Sept.), 281–284.

[33] D. E. Knuth. 1997. Fundamental Algorithms, Third ed. The Art of Computer Programming, vol. 1. Addison-Wesley, Reading, MA.

[34] I. P. Goulden and D. M. Jackson. 1983. Combinatorial Enumeration. Wiley, New York.

[35] S. Kundu and J. Misra. 1997. A linear tree partitioning algorithm. SIAM J. Comput. 6, 1 (Mar.), 151–154.

[36] H. Muratani. 2001. A collusion-secure fingerprinting code reduced by Chinese remaindering and its random-error resilience. In Information Hiding: 4th International Workshop (IHW 2001). (Pittsburgh, PA), 303–315.

[37] S. Chow, Y. GU, H. Johnson and V. Zakharov. 2001. An approach to the obfuscation of control flow of sequential computer programs. In Information Security: Fourth International Conference (ISC 2001), Davida and Frankl, Eds. Lecture Notes in Computer Science, vol. 2200. Springer Verlag, 144–155.

[38] C. Wang. 2000. A security architecture for survivability mechanisms. Ph.D. dissertation, University of Virginia, School of Engineering and Applied Science. www.cs.virginia.edu/survive/pub/wangthesis.pdf.

[39] K. Fukushima and K. Sakurai. A software fingerprinting scheme for java using classfiles obfuscation, 2004.

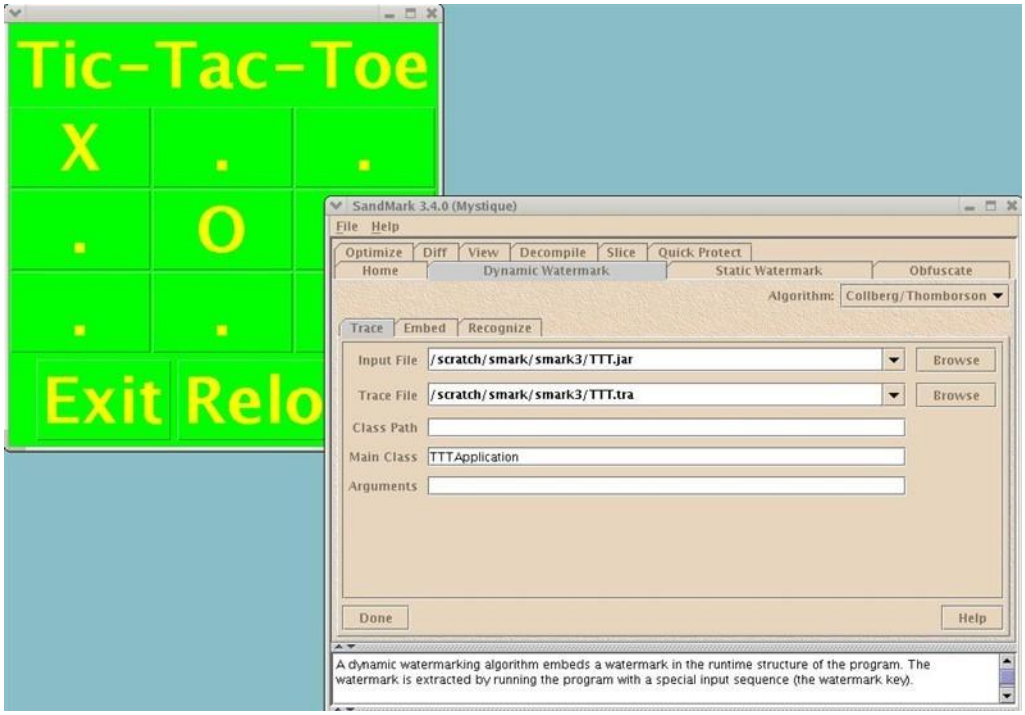
[40] K. Fukushima, T. Tabata, T. Tanaka and K. Sakurai. A software fingerprinting scheme for java using class structure transformation.Transactions of Information Processing Society of Japan, 46(8): 2042–2052, August 2005.

[41] W. Landi and B. G.Ryder. 1992. A safe approximate algorithm for pointer-induced aliasing. SIGP-L4N Not. 27, 7 (July), 235-248

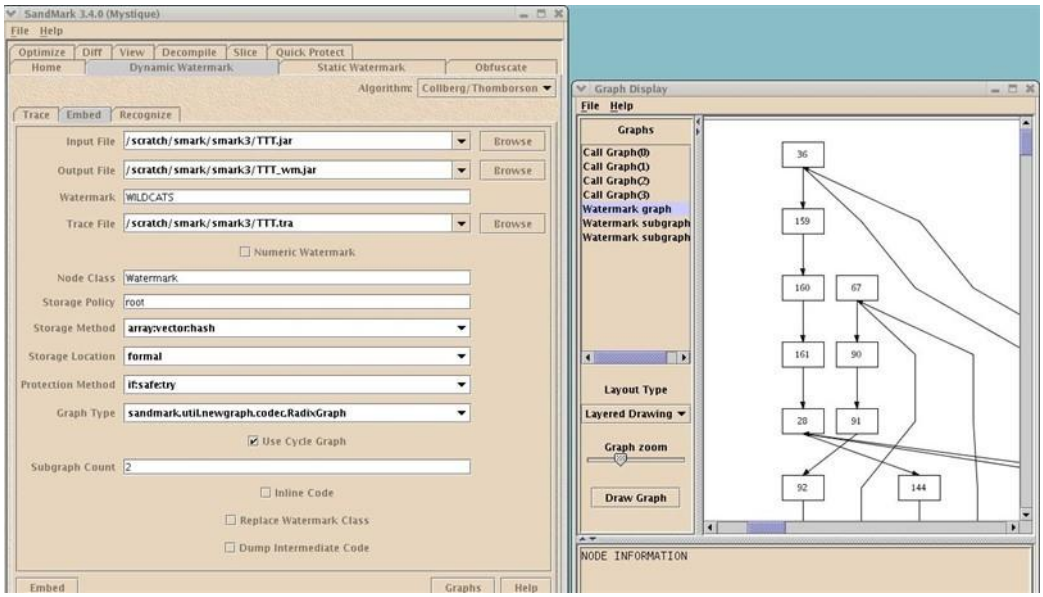
[42] Venkatesan T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 115–125, New York, NY, USA, 2003.

11. נספחים

SandMark 11.1 – מעקב (trace).



SandMark 11.2 – הטבעת הגרף המייצג.



11.3 Rank1 / Unrank1 [32]

פרק 6.1.5 מתאר קידוד תמורה (permutation encoding). לצורך ביצוע קידוד זה נדרשות פונקציות לביצוע מיפוי תמורה על מספרים שלמים, כלומר פונקציות להפיכת מספר שלם לסדרת מספרים (ולהיפך) ואשר ניתן לייצגם בגרף. להלן תיאור הפונקציות שהוצעו לצורך כך על ידי Collberg and Thomborson (2007) [19].

```
procedure unrank1(n, r,π)
        if n>0 then
                swap(π[n− 1],π[r mod n]);
                unrank1(n− 1, _r/n,π);
        fi;
end {of unrank1};
```

```
function rank1(n,π,π−1) : integer;
        if n = 1 then RETURN(0) fi;
        s := π[n− 1];
        swap(π[n− 1],π[π−1[n− 1]]);
        swap(π−1[s],π−1[n− 1]);
        RETURN(s + n · rank1(n − 1,π,π−1));
end {of rank1};
```

11.4 בעיית ההכרעה של המצביעים [13]

קומפיילרים משתמשים באנליזה סטטית. על מנת לבצע ניתוח של משתנים דינמיים נדרש מידע על "כינויים" או "שמות נוספים" (alias), כלומר האם שני ביטויים מסוג L-VALUED (משתנים, בצד שמאל של סימן = בתוכנית) עלולים/יכולים או חייבים להכיל את אותו ערך בנקודה מסוימת בתוכנית.
בצורה לא פורמלית, נאמר ששני ביטויים מסוג L-VALUE הם כינויים אחד לשני (alias) בנקודה ספציפית בזמן ריצת תוכנית אם שניהם מתייחסים (מצביעים) לאותו מיקום.
בבעיית ה-*may-alias* (ייתכן כינוי) מעוניינים לזהות כינויים שעלולים להתרחש בזמן חלק מהריצות של התוכנית. בעוד שבבעיית ה-*must-alias* (בהכרח כינוי) מעוניינים בזיהוי הכינויים בכל הריצות של התוכנית. מובן שהבעיה השנייה רלוונטית לרוב בעיות ניתוח זרימת המידע.
ניתוח התוכנית מתבצע תחת ההנחה השמרנית שכל הנתיבים בקוד ניתנים להרצה, מכיוון שההחלטה האם נתיב שרירותי בתוכנית ניתן להרצה אינה כריעה.

הנחה זו מאפשרת לפתור בעיות אנליזה של תוכניות, אך למרבה הצער הנחה זו אינה מספיקה בשביל להכריע את שתי הבעיות שתוארו למעלה.

<u>הגדרות</u>

- נאמר ששמות a ו-b הם *may-alias* אחד לשני בנקודה t מסוימת בתכנית אם קיים נתיב p מתחילת התוכנית לנקודה המסוימת t כך ש-a ו- b מתייחסים לאותו מיקום לאחר הרצת התוכנית לאורך נתיב p.
- נאמר ששמות a ו-b הם *must-alias* אחד לשני בנקודה t מסוימת בתכנית אם לכל נתיב p ששייך ל-P מתחילת התוכנית לנקודה המסוימת t, השמות a ו- b מתייחסים לאותו מיקום לאחר הרצת התוכנית לאורך הנתיב.

[41] Landi (1992) הראה שהם בלתי ניתנות להכרעה גם עבור שפות המאפשרות רקורסיה.

<u>בעיית ההכרעה של ה-aliasing</u>

1. may-alias

בהינתן נקודה בתוכנית ו-2 שמות, יש להחליט האם יחס ה-*may-alias* קיים ביניהם באותה נקודה בתוכנית. בצורה כללית יותר, אנו מעוניינים לייצר את הסט של כל השמות שהם *may-alias* אחד לשני בריצת תוכנית זו עד לנקודה.

<u>משפט</u>: בעיה זו אינה כריעה עבור שפות עם משפטי תנאי (if) לולאות (loops), זיכרון דינאמי ומבני נתונים רקורסיביים.

ההוכחה ניתנת במאמר.

2. must-alias

בהינתן נקודה בתוכנית ו-2 שמות, יש להחליט האם יחס ה-*must-alias* קיים ביניהם באותה נקודה בתוכנית בכל נתיב הרצה קיים. בצורה כללית יותר, אנו מעוניינים לייצר את הסט של כל השמות שהם *must-alias* אחד לשני בכל נתיבי הריצות של התוכנית לאותה הנקודה.

<u>משפט</u>: בעיה זו אינה כריעה עבור שפות עם משפטי תנאי (if) לולאות (loops), זיכרון דינאמי ומבני נתונים רקורסיביים.

ההוכחה ניתנת במאמר.

The Open University of Israel
Department of Mathematics and Computer Science

Principles, Techniques and Algorithms in Software Watermarking

Final Paper submitted as partial fulfillment of the requirements
towards an M.Sc. degree in Computer Science
The Open University of Israel
Computer Science Division

By
Yohay Librider

Prepared under the supervision of Prof. Ehud Gudes

May 2017

Table of contents

Abstract **1**

1. Purpose of work **2**

2. Introduction **3**

2.1 The problem 3

2.2 Goals and main uses 6

2.3 Pattern in software watermark embedding 6

2.4 Softwate watermark evaluation measurements 7

3. Attacks **9**

3.1 Additive attacks 11

3.2 Subtractive attacks 12

3.3 Distortive attacks 12

3.4 Collusive attacks 13

3.5 Protocol attacks 13

4. Main approaches and algorithms **15**

4.1 The static approach 15

4.2 The dynamic approach 20

5. Existing tools **22**

5.1 SandMark **22**

6. Collberg and Thomborson (CT) algorithm **25**

6.1 Algorithm description 28

6.1.1 Basic implementation 28

6.1.2 Annotation 29

6.1.3 Tracing 30

6.1.4 Embedding 34
6.1.5 Building the graph 34
6.1.6 Extraction 39
6.2 Algorithm improvements suggestions 42
6.1.1 Improving resilence 42
6.1.2 Increasing fingerprint size 46
6.3 Algorithm evaluation done by Collberg and Thomborson 51
6.1.1 Stealth 51
6.1.2 Data rate 57
6.1.3 Resilience 60
7. Discussion and conclusion 62
8. Results and conclusions from my project 63
9. Summary 66
10. References 67
11. Appendices 71
11.1 SandMark – Trace 71
11.2 SandMark – Graph embedding 71
11.3 Rank1 / Unrank1 [32] 72
11.4 The undecidability of aliasing [13] 72

List of figures

Figure 1 – Example of Embedding sowtare watermark process (1) 5
Figure 2 – Embedding sowtare watermark (2) 10
Figure 3 – Additive Attacks 11
Figure 4 – Subtractive Attacks 12
Figure 5 – Distortive Attacks 13
Figure 6 – SHKQ – Algorithm example. 18
Figure 7 – CT Algorithm parts 26
Figure 8 – Example of basic CT algorithm – A class before and after CT algorithm. 27
Figure 9 – Example of run time trace points registration 31
Figure 10 – Software watermarking graph encodings 35
Figure 11 – A 4-nodes PPT graph 38
Figure 12 – Modifying the constructor for Java's root class to support CT algorithm 40
Figure 13 – A naive algorithm for extracting the software watermark 41
Figure 14 – A naive algorithm for extracting a sw graph encoding in the heap memory 42
Figure 15 – Obfuscation attacks on software watermarks graphs 44
Figure 16 – Using the LinkedList and Event classes for building the graph 49
Figure 17 – Definition – Local stealth 52
Figure 18 – Definition – Stenographic stealth 52
Figure 19 – Digram frequencies generated for a 32-bit CT fingerprint. 54
Figure 20 – Steganographic stealth evaluation 57
Figure 21 – Sizes of the graph as a function of the size of the fingerprint. 58
Figure 22 – Sizes of the graph building bytecode as a function of the size of the fingerprint 59
Figure 23 – Size of the bytecode for building a Radix Graph, as a function of the number of components into which the graph is split. 60
Figure 24 – Bytecode size, and runtime heap size, for cycled radix graphs. 61
Figure 25 – A succsessive watermark extract from the calculator application 65

Abstract

In today's world of Internet, there is a growing importance in dealing with software theft. The ease with which unauthorized programs are downloaded, allows more and more people to use them (consciously or unconsciously). The estimated value of these software stands at $ 52 billion in 2016 [23]. A survey of the BSA from 2016 [23] also shows a close link between unauthorized software and malware and cyber-attacks.

One of the measures proposed to reduce this phenomenon is through ownership authentication mechanisms. This field has been studied extensively and incorporates many methods and algorithms. One of the suggested methods is the embedding of software watermark. Embedding a "watermark" in the software for identity verification is a kind of "fingerprint" hidden in the code, and in effect serves as a license to use the software. This license can be retrieved when required and claimed for the authenticity (or lack of authenticity) of the software.

This work will present an overview of various "watermark" technologies, review their advantages and disadvantages and their resistance to various attacks, it is also review a number of commercial and research tools used, the work use various articles, but mainly articles [19], [20], [21].

The main part of the work will be directed to Collberg and Thomborson (2007) [19], which presents a complete algorithm for embedding a dynamic software "watermark" in a program that is realized by using a Dynamic Graph Watermarking (DGW). The work will present the algorithm, will discuss techniques to improve the concealment of the fingerprint, resilience, and resistance to various attacks, and will present an analysis of the efficiency of the algorithm in these aspects.

In my final project, the CT algorithm was fully implemented. The implementation has built a general application, a kind of infrastructure tool, so that it is easy and automatic to build and implement, and then extract the watermark to / from any hosted program. In fact, it created a useful tool for software creators in the Microsoft .NET environment that enables them to automatically embed a watermark in their program and without any manual intervention in the code they created based on a confidential input. The tool also allows you to extract the watermark from the programs in which it is embedded, also automatically.