אבטחת מידע – תיאוריה בראי המציאות

אלי דיין[1]

[1] <eliyahud@post.tau.ac.il>

תקציר

מסמך זה יביא את סיכומי השיעורים מהקורס אבטחת מידע – תיאוריה בראי המציאות, שהועבר על ידי ד"ר ערן טרומר בסמסטר א' בשנה"ל תשע"ג.

תוכן עניינים

פרק 1

מבוא[1]

1.1 אלגוריתמים קריפטוגרפיים

המודל: קלט (גלוי, מפתח) שנכנס לקופסה שחורה שיוצא ממנה פלט (סתר).

יש הגדרות לאבטחה (כמו CPA, CCA1, CCA2 ועוד) ואלגוריתמים שנלמדו רבות (RSA, DES, AES ועוד). בגלל שהאלגוריתמים האלה מוכרים מאוד, כולם מאמינים שהם בלתי ניתנים לתקיפה.

בעולם שבו יש אלגוריתמי הצפנה בלתי ניתנים לתקיפה, והמון הגדרות, למה שתהיינה בעיות אבטחה? האלגוריתמים הללו טובים ויפים ותאורטיים, אולם במציאות הכל צריך לעבור מימוש כלשהו, וייתכן כי במימוש יש בעיות.

לפעמים, השגת נגישות למימוש הפיזי נותנת לנו אפשרות לתקוף את ההצפנה.

1.2 התקפות ארכיטקטוניות

לא תמיד יש לנו יכולת להגיע פיזית למכונת ההצפנה. אולם, לפעמים יש לנו נגישות לתשתית הלוגית (כמו למשל הרצת קוד על מחשב מרוחק).

למשל, במחשוב ענן[2], בו חברה מחזיקה חוות שרתים גדולה, ומקימה מכונות וירטואליות לפי דרישה. ישנם שירותי ענן ציבוריים (כמו Amazon), המאפשרות לכל אחד לפתוח מכונה בתשלום.

נוצרת כאן בעיה חמורה: כל המכונות הווירטואליות רצות על אותה החומרה, כלומר על אותו המחשב הפיזי. האם יש אפשרות שמכונה וירטואלית אחת יכולה לגנוב מידע ממכונה וירטואלית אחרת שנמצאת על אותה החומרה?

כדי למנוע מצב כזה, צריך לוודא שלכל מכונה יהיה מערך זיכרון נפרד, מערכת קבצים נפרדת, כל מכונה תקבל מערך דפים פיזיים של זיכרון נפרד, ולהקפיד לאפס דפים כאשר הם עוברים בין מכונות.

באופן דומה, ניתן לחשוב גם על כל תהליך במערכת ההפעלה כבלתי תלוי ובלתי מודע לתהליכים הנוספים שרצים על אותו המחשב, באותה מערכת ההפעלה. אז לכאורה ההפרדה בין התהליכים היא אבסולוטית, אולם שני תהליכים מתחרים על אותם המשאבים הפיזיים (כמו למשל זיכרון או מטמון).

---

[1] השיעור התקיים בתאריך 23.10.2012.

[2] הטרמינולוגיה האנגלית היא Cloud Computing.

1.2.1 סוגי תקיפות

נדבר כאן במונחים של גנבת מידע. יש לנו שני מחשבים וירטואליים שרצים על אותה המכונה:

ערוץ צד התוקף משתמש בערוץ צד כלשהו כדי לגנוב את המידע. משהו שלא אמורים לחשוב עליו בכלל.

ערוץ סמוי התוקף משתמש בערוץ סמוי לתוך המידע החסוי. כלומר, נוצר קשר בין התוקף למותקף, אבל בצורה סמויה.

ראוי לציין כי יש עוד סוגי תקיפות נוספים ורבים.

1.2.2 התקפות מטמון

במערכות מבוססות מטמון, התוקף יכול להרגיש בהאטה בקצב הגישה לזיכרון כאשר הקרבן ניגש לזיכרון שלו.

זה קורה כאשר המטמון מתמלא בכתובות של התוקף, וכאשר הקרבן מבצע פעולה, הוא דורס את אחת הכתובות של התוקף במטמון. כשהתוקף ירוץ מחדש, הוא ירגיש האטה בגישה לזיכרון, כי צריך להביא את הזיכרון מחדש למטמון.

אמנם בצורה כזו לא ניתן לגשת למידע הסודי, אך ניתן להבין את דפוסי הפעולה של הקרבן. מעבר לכך, ניתן להסיק מידע על מיקומי הכתובות אליהן ניגש הקרבן.

תקיפה מסוג זה מאפשרת לנו למשל להשיג מידע אודות מפתח AES, באמצעות הפעלת משוואות לינאריות על פי ה־lookup tables.

1.2.3 Hyper Attacks

התקפות Hyper הן הרחבה של התקפות מטמון.

במחשבים מודרניים, כדי להשיג מקביליות, המעבדים משתמשים בטכנולוגיית Hyper-threading, המאפשרת הרצת מספר Thread־ים במקביל על אותה ליבת חישוב בו־זמנית. כמו כן, יש מספר ליבות חישוב, עם מטמונים משותפים.

כדי לתקוף, נבדוק את נתוני המטמון בזמן אמת. תוכנת ההצפנה לא מדברת עם העולם החיצון, אלא מבצעת חישובים רבים, ולכן נראה שהמטמון מתנהג באופן שונה.

1.2.4 דוגמה לתקיפת RSA

בחישובי הצפנת RSA, נחשב ביטויים מהצורה c^d. לרוב, עושים את זה בעזרת אלגוריתם Square-Multiply, כפי שמתואר באלגוריתם 1.1.

ואכן, מהמימוש ניתן לשחזר את d: בכל פעם שניגשים ל־v, אנחנו עוברים ביט של d. אם ניגשים גם ל־c, אז הביט הוא 1. אחרת, הביט הוא 0.

חשוב לשים לב שבאלגוריתם RSA, לרוב d הוא המפתח הפרטי, וזה המידע הסודי אותו אנו גונבים.

הערה כל פעולות הכפל שמבוצעות בתוך האלגוריתם הן מעל החבורה $\mathbb{Z}_n$[3], ולכן מדובר על פעולות כפל גדולות יותר, ולא על מילות מכונה. זה מקל את העבודה על התוקפים.

[3] לרוב החישובים הם מעל $\mathbb{Z}_{2048}$.

אלגוריתם 1.1 חישוב חזקת שלמים באלגוריתם Square-Multiply

נחשב את c^d:

1. נשים $v \leftarrow 1$.
2. לכל $i \in \{1, \ldots, \log d\}$ (נעבור על הביטים של d):
   (א) נשים $v \leftarrow v^2$.
   (ב) אם $d_i = 1$:
   i. נשים $v \leftarrow v \cdot c$.
3. נחזיר את v.

1.2.5 דוגמה לתקיפת ערוץ צד

מדידת הרעש שיוצא מהמחשב בזמן החישובים.
כאשר המחשב ב־Idle, הוא מרעיש יותר. כל פעולה שהמחשב מבצע משפיעה על הרעש שיוצא ממנו (למשל, הקבלים מוציאים רעשים שונים).
זוהי התקפת צד פיזיקלית.

1.2.6 משמעויות

כנראה שאפשר לבצע את התקיפות האלה על אותה המכונה בצורה קלה יחסית.
אפשר לחשוב גם על מערכת רבת משתמשים כמערכת שחשופה לכך, מכיוון שכל המשתמשים בסופו של דבר משתמשים באותה תשתית חומרתית.
למשל, כל הדפדפנים מריצים היום JavaScript, שמאפשרת הקצאת זיכרון ומדידת זמנים. באופן כללי, יש הרבה אפשרויות להרצת קוד לא מאובטח, אפילו בצורה שהיא sandboxed (כמו ActiveX, .NET, Google Native Client).
לפעמים, התוקף הוא המשתמש הלגיטימי של המכונה. למשל, DRM[4] מאפשר הגנה על תוכן, בצורה כזו שגם ה־Administrator של המכונה לא יכול לעשות בו כרצונו.

1.2.7 וירטואליזציה

הווירטואליזציה מוערכת מאוד, הרבה בזכות היתרונות האבטחתיים שלה. אולם, היא אינה חסינה בפני התקפות ערוצי צד.
יחד עם זאת, יש בעיה לגשת למידע האמיתי בתוך מערכות וירטואליות. כתובת המידע עוברת טרנספורמציה פעמים רבות בשכבות רבות בדרך (הכתובת שידועה לתהליך, מערכת ההפעלה במכונה הווירטואלית, מערכת ההפעלה האמיתית, וכו').
הסתברותית, קשה מאוד לתאם את כל הפרטים האלה. האמנם? חברת Amazon מאפשרת קישור בין המכונה הווירטואלית לבין המכונה הפיזית, ממגוון סיבות[5].
כמו כן, בתקן EC2[6], לפי כתובת ה־IP של המכונה הווירטואלית ניתן להבין לאילו מכונות וירטואליות נוספות המכונה קרובה, כי קרבה ב־IP היא גם קרבה פיזית.

[4] ראשי תיבות של Digital Rights Management.
[5] כמו רצון שמכונה מסוימת תהיה בשטח ארה"ב, או האיחוד האירופי. אולי גם רצון לפרוש את המכונות על פני מקומות שונים בעולם, כדי לאפשר יתירות.
[6] ראשי תיבות של Elastic Cloud.

איך אפשר לוודא שהגענו לאותה מכונה? לפי כתובות IP, או באמצעות הרצת Trace Route. אפשר גם לשלוח בקשות למכונה המותקפת, ולראות אם גישה לזיכרון במכונה התוקפת מואטת כאשר מגיעה בקשה. כך ניתן לוודא שאכן נמצאים על אותה מכונה פיזית.

1.3 בירוקרטיה

אתר הקורס: http://course.cs.tau.ac.il/istvr12-13/, או http://cpiis.cs.tau.ac.il ומשם לבחור ב־Courses.

1.3.1 סילבוס

ראינו כמה דוגמאות לבעיות באבטחת מידע.
הקורס בא לתאר מקרים בהם יש הבדל בין התאוריה למציאות באבטחת המידע.
בהמשך נראה מה אפשר לעשות כדי להגן על המידע בכל זאת, הן במישור המערכות והן במישור האלגוריתמים הקריפטוגרפיים.

1.3.2 דרישות הקורס

- מילוי שאלון באינטרנט.
- לוודא שכולם ברשימת התפוצה.
- תרגילים:
  - יהיו כ־6 תרגילים.
  - משקלם יהיה 30% מהציון הסופי.
  - להגשה תוך שבועיים.
  - הגשת התרגילים בזוגות.
- מטלות קריאה.
- מבחנים:
  - מועד א': 21.02.2013.
  - מועד ב': 12.04.2013.
- מבנה המבחן:
  - שילוב של שאלות פתוחות ואמריקאיות.

1.3.3 משאבים

- אתר הקורס: http://course.cs.tau.ac.il/istvr12-13/.
- רשימת תפוצה.
- אין ספר שמכסה את תוכן הקורס.
  - מומלץ לעיין בספר הבא: Ross Anderson, *Security Engineering*, 2nd Edition.

פרק 2

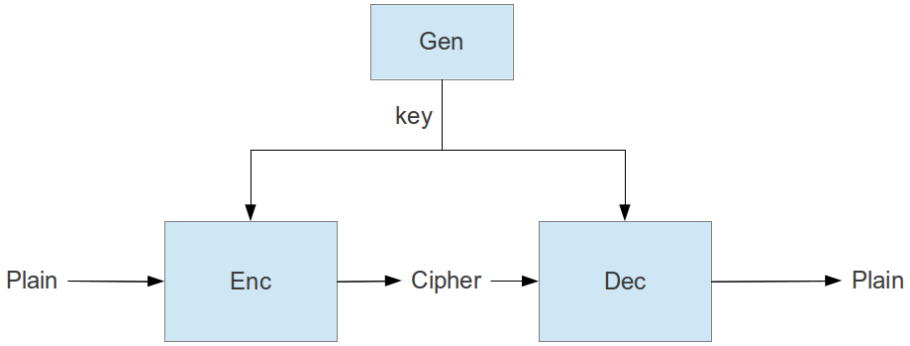
קריפטוגרפיה ותקיפות כשלים[1]

2.1 סכימת הצפנה כללית

איור 2.1 מתאר סכימת הצפנה כללית, בה יש מנגנון הצפנה ומנגנון פענוח. שני המנגנונים צריכים לקבל מפתח, שמוגרל על ידי מנגנון נוסף, המעביר את הפלט שלו לשני המנגנונים האחרים.
אנחנו מודדים את מערכות ההצפנה שלנו בשני מדדים:

- נכונות (Correctness).
  על המנגנון להיות הפיך ונכון.
- אבטחה (Security).
  כל אחד מהחלקים בסכימת ההצפנה צריך להיות חזק ובלתי ניתן לשבירה. אחרת, בעזרתו ניתן לשבור את כל אחד מהחלקים האחרים (או להוציא גלוי מסתר).

[1] השיעור התקיים בתאריך 30.10.2012.



איור 2.1: סכימת הצפנה כללית

2.2 תקיפות למנגנוני הצפנה

ניתן לסווג את הבטיחות של מנגנון הצפנה לפי התקיפות בפניהן הוא עמיד.

2.2.1 תקיפת באמצעות Chosen Plaintext Attack

השיטה עובדת כך: התוקף שולח שני גלויים למצפין, ומקבל את הסתר של אחד מהם.

באופן פורמלי, התוקף מגריל שני גלויים, m_0 ו־m_1, ושולח אותם למצפין. המצפין בוחר $b \in \{0,1\}$, ואז מצפין את m_b בעזרת מפתח k שיוצר על ידי Gen. כלומר, $c_b = \mathbf{Enc}_k(m_b)$. התוקף מקבל בחזרה את c_b, וצריך לומר מהו b. התוקף מחזיר את הניחוש שלו b'. נסמן: $\Pr[b = b'] = 1/2 + \varepsilon$. אם $b' = b$, נאמר שהתוקף הצליח.

נשים לב שההסתברות הבסיסית להצלחה היא $1/2$, ו־ε הוא פונקציה של כוח החישוב של התוקף.

התקיפה הזאת מאוד מאתגרת, אבל זה עדיין מאוד קשה ליריב. לכן, נשנה קצת את המשחק: לפני תחילת התהליך, היריב יכול לשלוח את הגלויים $\tilde{m}_i$ למצפין, ולקבל את $\tilde{c}_i = \mathbf{Enc}_k(\tilde{m}_i)$ בחזרה.

הבעיה מבחינת המצפין היא שהתוקף יכול לשלוח את m_0 ואת m_1, ואז לדעת מראש את הסתרים שהוא צפוי לקבל. פתרון לבעיה זו הוא הוספת אלמנט של אקראיות בפונקציית ההצפנה, כך שעבור גלוי m כלשהו, נקבל שני סתרים $c = \mathbf{Enc}_k(m)$ ו־$c' = \mathbf{Enc}_k(m)$ שונים. הצפנה מסוג זה נקראת *Probabilistic Encryption*.

2.2.2 תקיפה באמצעות Known Plaintext Attack

שיטה זו דומה ל־Chosen Plaintext Attack, אולם התוקף לא יכול לבחור את הגלוי, אך הוא יודע אותו, ומבצע את המחקר על הסתר שמתקבל ממנו.

התיאור המדויק של השיטה תלוי במנגנון ההצפנה הספציפי.

2.2.3 תקיפה באמצעות Chosen Ciphertext Attach

לשיטה זו – Chosen Ciphertext Attack (או בקיצור CCA), שתי וריאציות:

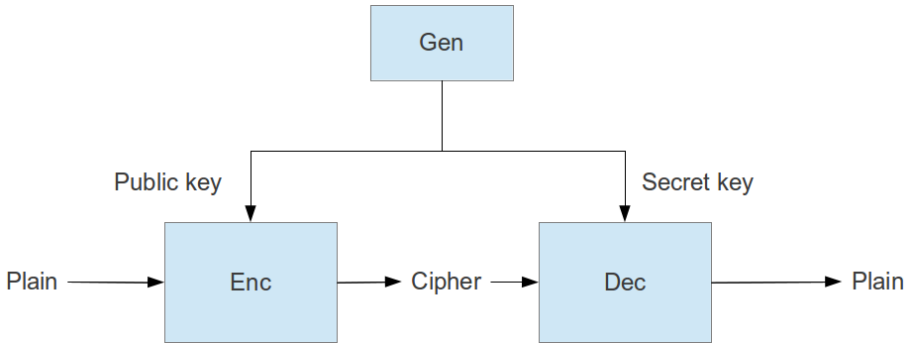
CCA-1 לפני תחילת התהליך, לאחר שהתוקף שולח רצף של גלויים להצפנה, הוא יכול לשלוח רצף של סתרים לפענוח, עם אותו המפתח k.

CCA-2 בנוסף למה שיש ב־CCA-1, בסוף התהליך, התוקף שולח סתר לפענוח. יצוין כי הסתר שהוא שולח חייב להיות שונה מ־c_b.

2.2.4 חוזק התקיפות

קל לראות כי Known Plaintext Attack יותר חלשה מ־Chosen Plaintext Attack, כי אם אנחנו בוחרים את הגלוי, אז אנחנו גם יודעים אותו.

כמו כן, Chosen Ciphertext Attack יותר חזקה מ־Chosen Plaintext Attack, כי אפשר להתעלם מהתוצאות של המפענח.



איור 2.2: סכימת הצפנה א־סימטרית

2.2.5 סיווג אלגוריתמים

אלגוריתמים כמו RSA או AES הם דטרמיניסטיים, ולכן הם לא יכולים לעמוד בפני Chosen Plaintext Attack או Known Plaintext Attack.
ההתמודדות עם זה היא הוספת מאפיינים אקראיים לגלוי (כמו שרשור של תווים אקראיים), בכדי להוציא פלט אקראי בכל פעם.

2.3 אלגוריתמים א־סימטריים

עד עכשיו דיברנו על אלגוריתמים סימטריים, בהם מפתח ההצפנה ומפתח הפענוח זהים. אולם, אלגוריתמים מודרניים רבים, כגון RSA, משתמשים בשני מפתחות שונים:

- מפתח ציבורי/פומבי (Public Key).
  זהו מפתח הידוע לכולם, ומפורסם לכולם.
- מפתח סודי/פרטי (Secret/Private Key).
  זהו מפתח סודי, השמור למפענח בלבד, וההצפנה תצליח רק אם הוא ישמר בסוד.

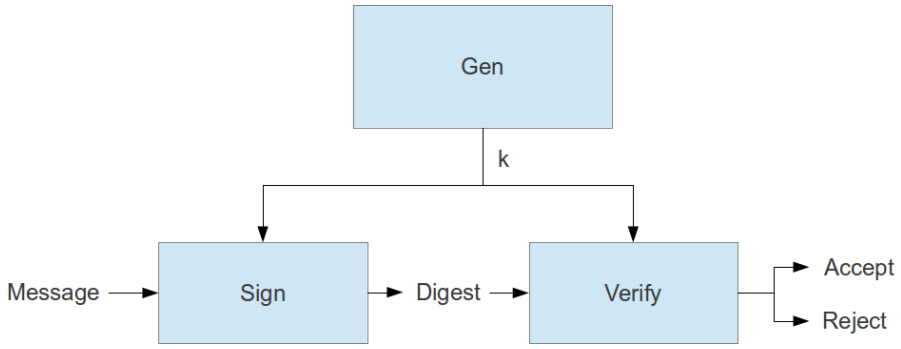
תיאור סכימתי של הצפנה א־סימטרית ניתן למצוא באיור 2.2.
כדאי לשים לב שחלק מהתקיפות המוזכרות לעיל אינן תקפות עבור אלגוריתמי הצפנה א־סימטריים (כמו למשל Chosen Plaintext Attack), מכיוון שמפתח ההצפנה ידוע מראש לכל.

2.4 הצפנה ב־AES

אלגוריתם AES מצפין בלוקים בצורה סימטרית. האלגוריתם מצפין בלוקים של 128 סיביות, בהינתן מפתח של 128 סיביות, ומחזיר סתר של 128 סיביות.

2.5 מטרות האבטחה

לאבטחה יש 3 מטרות:



איור 2.3: תיאור סכימתי של MAC

סודיות המידע צריך להישאר סודי, ואסור שהוא ייחשף לאנשים שלא אמורים להיחשף אליו. בלעז, מטרה זו נקראת Secrecy או Confidentiality.

שלמות האבטחה צריכה להבטיח שהמידע לא שובש, ונשלח מהאדם הנכון. בלעז, מטרה זו נקראת Integrity.

זמינות המידע צריך להיות זמין ונגיש למי שצריך אותו. תכונה זו לא מעניינת אותנו בהקשר של הקורס. בלעז, מטרה זו נקראת Availability.

2.6 שלמות המידע

2.6.1 Message Authentication Code (MAC)

איור 2.3 הוא תיאור סכימתי של MAC (סימטרי). לעתים, יש שיגידו שגם המוודא (Verifier, או Verify באיור) מקבל את ההודעה.

מטרתו של האלגוריתם היא לחתום על מידע, ובכך לאשר שהוא לא שובש, והגיע מהמקור האמיתי שלו.

לרוב, המימוש משתמש בפונקציית Hash על ההודעה. פונקציית Hash קריפטוגרפית אמורה להיות חזקה יותר מפונקציה רגילה, מכיוון שהפלט שלה אמור להיראות אקראי[2].

מינוח התהליך שלנו הוא $\mathbf{Sign}(m) = d$, כאשר m הוא ההודעה (Message) ו־d הוא החתימה (Digest).

2.6.2 חתימות דיגיטליות

החתימות הדיגיטליות מוגדרות באופן דומה ל־MAC, כאשר המנגנון החותם (Sign), והמנגנון המוודא (Verify) מקבלים מפתחות שונים.

המפתחות המשתתפים במקרה הזה הם:

[2] למעשה, זהו נפנוף ידיים, מכיוון שהפונקציה דטרמיניסטית. יש הגדרה תאורטית פורמלית לא מעניינת. נתייחס לזה כאל עובדה.

- מפתח חותם (Signing Key).
  זה המפתח המשמש לחתימה על ההודעה.
- מפתח וידוא (Verification Key).
  זה המפתח המשמש לווידוא החתימה על ההודעות.

לרוב, אלגוריתמים מסוג זה עושים שימוש באלגוריתם MAC כלשהו, ובאלגוריתם הצפנה א־סימטרי בכיוונים השונים שלו.

2.6.3 הגדרות בטיחות לשלמות המידע

הגדרות הבטיחות לשלמות המידע נובעות מהאלגוריתמים לתקיפה על אלגוריתם החתימה.

- תקיפת התנגשות (Collision Attack).
  תקיפה כזו מצליחה למצוא שתי הודעות m_1 ו־m_2, כך ש־$\mathbf{Sign}(m_1) = \mathbf{Sign}(m_2)$.
- תקיפת תמונה־מוקדמת (Preimage Attack).
  תקיפה כזו מצליחה למצוא הודעה m בהינתן חתימה d ש־$\mathbf{Sign}(m) = d$.
- תקיפת תמונה־מוקדמת מסדר שני (2nd Preimage Attack).
  תקיפה כזו מאפשרת בהינתן הודעה m_1 למצוא הודעה m_2, כך ש־$\mathbf{Sign}(m_1) = \mathbf{Sign}(m_2)$.

חשוב להבין כי אין אלגוריתם שלמות מידע קריפטוגרפי שחסין בפני אף אחת מההתקפות הללו, מכיוון שאלגוריתם החתימה הוא למעשה פונקציה ממרחב גדול למרחב קטן, ולכן בהכרח אינו חד־חד־ערכי (חח"ע).

דוגמאות לפונקציות Hash קריפטוגרפיות שעדיין לא נמצאו עבורן אלגוריתמי תקיפה הן SHA-1, SHA-2, SHA-256, KECCAK.

2.6.4 במציאות

אלגוריתם לחתימה דיגיטליות מניב לנו פלטים נוספים, בנוסף על החתימה, ואותם אפשר לתקוף. פלטים אלו אינם קיימים באלגוריתם התאורטי, אבל במימוש המציאותי הם קיימים.

למשל, לכל אלגוריתם יש Memory Access Pattern משלו, שניתן לנסות ולשחזר באמצעותו את הקלט (כפי שראינו בשיעור שעבר עם AES), וערוצי צד נוספים, כמו תקיפות תזמון, צריכת אנרגיה, קרינה אלקטרו־מגנטית (EM) ועוד רבים שבוודאי לא ידועים לנו כעת.

בנוסף על הפלטים הנוספים, יש קלטים נוספים לאלגוריתם החתימה, שאינם קיימים בתיאוריה. למשל, במציאות קשה לחולל מספרים אקראיים בצורה טובה.

באמצעות הקלטים הנוספים שאינם קיימים בתיאוריה, ניתן לתקוף את האלגוריתם בכל מיני מתקפות שונות, כגון Power Attacks או Fault Injection Attacks, EM, כל מיני תקיפות פיזיות (למשל כיפוף סיליקון) ועוד.

למעשה, היינו רוצים לצמצם ככל הניתן את ערוצי הקלט והפלט של האלגוריתם שלנו, ובכך לשפר את הבטיחות שלו.

2.7 כשלים (Faults)

- כשל חולף (Transient fault).
למשל, התחממות יתר של שבב אלקטרוני עשויה לגרור התנהגויות בלתי־צפויות.

- כשל קבוע (Permanent fault).
למשל, Electro migration.

- סוסים טרואנים (Trojan horses).
לרוב, המשתמש אינו מכיר את יצרן החומרה. עשוי להיות מצב שבו יצרן חומרה מכניס בתוך החומרה סוס טרואני, שביום פקודה גורם לחומרה להפסיק לעבוד, או חמור יותר – לעשות דברים רעים.

- שגיאות בחומרה (Errors in circuit).
למשל, שגיאת החלוקה בנקודה צפה (FDIV) במעבדים של חברת Intel.

2.7.1 כשל ב־RSA

תזכורת

ב־RSA, נתונים שני ראשוניים (סודיים) p ו־q. המפתח הפומבי הוא $n = p \cdot q$.
בהינתן הודעת גלוי m, מחשבים את הסתר c באופן הבא:

$$c = \mathbf{Enc}_n(m) = m^3 \pmod n$$

(הנחנו כי 3 הוא המפתח הציבורי).
הפענוח מתבצע באופן הבא: בהינתן סתר c, והמספר d כך ש־$d \cdot 3 \equiv 1 \pmod{(p-1)\cdot(q-1)}$, מחשבים את הגלוי m באופן הבא:

$$m = \mathbf{Dec}_n(c) = c^d \pmod n$$

לעתים, על מנת לחסוך בזמן ריצה, מממשים את הפענוח כפי שמתואר באלגוריתם 1.2, שעשוי להיות פי 2 יותר יעיל.

אם נוכל בדרך כלשהי לשבש את החישוב של c^{d_p}, ולהחזיר במקומו את $\overline{c^{d_p}}$, נקבל $\overline{m_p}$ במקום m_p, ואז $\bar{m}$ במקום m. אם יש לנו דרך לגרום למפענח לפענח פעמיים את אותו הסתר, פעם אחת באופן תקין, ופעם אחת באופן שגוי, נקבל את m ואת $\bar{m}$. בסך הכל, קיבלנו:

$$\begin{aligned} m &\equiv \bar{m} \pmod q \\ m &\not\equiv \bar{m} \pmod p \end{aligned}$$

נוכל להסיק כי $q = \gcd(m - \bar{m}, n)$, כי $q \mid m - \bar{m}$, אבל $p \nmid m - \bar{m}$.

2.7.2 תקיפה למערכת סימטרית

נדבר שוב על תקיפת חומרה. נניח כי קיים רכיב חומרה, שאליו נצרב המפתח בזמן הייצור, ולאחר מכן אין אפשרות לשנות את המפתח הצרוב ברכיב.

אלגוריתם 1.2 פענוח יעיל ל־RSA

בהינתן מספרים ראשוניים p ו־q, נסמן: $n = p \cdot q$. נניח כי המפתח הפרטי (לפענוח) הוא d.

1. נסמן:

$$\begin{aligned} d_p &= d \mod (p-1) \\ d_q &= d \mod (q-1) \end{aligned}$$

2. נסמן: $m_p \leftarrow c^{d_p} \mod p$.
3. נסמן: $m_q \leftarrow c^{d_q} \mod q$.
4. לפי משפט השאריות הסיני, קיים m יחיד שמקיים:

$$\begin{aligned} m &\equiv m_p \pmod p \\ m &\equiv m_q \pmod q \end{aligned}$$

וקיים אלגוריתם יעיל למציאת m.

הרכיב מממש מנגנון הצפנה, שמקבל גלוי m, ומחזיר סתר c לפי המפתח k שצרוב בו. המפתח k אינו ניתן לשינוי, לשליטה או לקריאה.
נניח כמה הנחות לגבי הכשלים שאנחנו יכולים לעשות:

- כשל יכול לאפס ביט, אבל לא לשנות את ערכו ל־1.
- כל פעם מתאפס ביט בודד.
- הכשל נשאר לנצח ברכיב, כלומר לאחר שהכשל קרה, לא ניתן לחזור חזרה למצב התקין.
- האלגוריתם הינו דטרמיניסטי.

תקיפת המערכת הקלט של המערכת יהיה תמיד הגלוי m_0, ונקבל סדרה של סתרים c_i, אחרי איפוסים של ביטים שונים. למעשה, בפועל לכל c_i מתאים גם k_i (כי k משתנה בהתאם לכשלים, שעשויים לשנות את c).
אם t היא הפעם האחרונה שהסתר השתנה, אנחנו יכולים להניח כי $k_t = 00\cdots0$. לכן, אנחנו יודעים כי כי c_t הוא הסתר של הגלוי m_0 עבור המפתח $00\cdots0$.
עכשיו, נוכל לנסות לנחש מהו הביט שהתאפס אחרון, ולראות מתי נקבל את הסתר c_{t-1} עבור הגלוי m_0. נוכל להמשיך כך עד אשר נשחזר את כל $k = k_0$.
הסיבוכיות של האלגוריתם בצורה הנאיבית ביותר (Brute Force) היא $O(2^t)$.

הערה נשים לב לכך שהתקיפה לא עובדת על אלגוריתמים חלשים מדי, כמו אלגוריתמים שמתעלמים מהמפתח.

פרק 3

Tamper resilience and hardware security[1]

סיכום השיעור אינו זמין כעת.

[1] השיעור התקיים בתאריך 06.11.2012.

פרק 4

[1]Process Confinement

4.1 הרצת קוד לא מהימן

לעתים קרובות, קיים הצורך בהרצה של קוד שאינו מהימן. דוגמה לכך היא בשימוש באינטרנט – ActiveX או JavaScript. מעבר לנושא האבטחה, לפעמים הקוד מכיל באגים, ונרצה אפשרות להריץ את הקוד בבטחה. למשל, לא היינו רוצים שדפדפן האינטרנט יקרוס בגלל באג באתר האינטרנט.

קוד מסוכן נוסף הוא תכנות ישנות (*legacy*) שמותקנות על המחשב זמן רב. בגלל ההיכרות של המשתמש עם התכנות האלה, הוא בוטח בהן לחלוטין. אולם אם הקוד מאוד ישן, ייתכן מאוד שכבר נמצאו בו חולשות, ואף אחד עדיין לא עדכן את התכנה וסגר פרצות האבטחה.

לפעמים, נרצה לרצה גם לטמון לקוד זדוני "פלכודת דבש" (*honeypot*). כלומר, אנחנו יודעים שיש קוד זדוני שרץ במחשב כלשהו, והיינו רוצים להריץ את הקוד באופן בטוח, שלא ייגרם נזק, כדי ללמוד את הקוד הזדוני.

המטרה העיקרית שלנו היא להרוג תכנות בעלות התנהגות חמורה.

4.2 Confinement

היינו רוצים להגביל תכנות, כדי שלא יצאו מה־scope של מה שהן אמורות לעשות. למשל, מהדר (Compiler) לא אמור לפתוח socket ולשלוח למישהו בחוץ את הקוד אותו אנו מהדרים.

יש מגוון של הגבלות הניתנות ליישום כדי לכפות את ה־Confinement:

- חומרה – בידוד של החומרה (למשל מחשב stand-alone).

  יתרונות:

  - מאובטח יחסית, מכיוון שאי אפשר לתקשר עם המחשב מרחוק.
  - התוקף חייב (כנראה) גישה (פיזית) ל־red side.

  חסרונות:

[1] שיעור שהתקיים בתאריך 13.11.2012.

- קשה למימוש בפועל (הפתרון יקר, וקשה לוגיסטית).
- לא תמיד מספק את הצרכים, כי האפליקציה לא תמיד יכולה להיות מבודדת מהעולם.

- מכונות וירטואליות.

  הרעיון הוא שמערכת ההפעלה לא מכירה מערכות אחרות. הבעיה היא שהמשאבים הפיזיים עדיין משותפים, ואנחנו כבר מכירים תקיפות אפשריות במצב הזה.

- מערכת ההפעלה מבודדת תהליכים שונים.

  גם כאן, הרעיון הוא שכל תהליך הוא כמו "מחשב וירטואלי". כמו כן, מערכת ההפעלה מגדירה הרשאות לכל תהליך, ואומרת לו מה מותר ומה אסור.

  הבעיה כאן היא עודף האבסטרקציות על המשאבים המשותפים, ונדרשת עבודה רבה כדי להתאים לכל אפליקציה את סט ההרשאות המינימליות הדרוש כדי לקבל ריצה תקינה.

- לפעמים נוח לעשות דברים בתוך תהליך בודד. למשל, JavaScript בדפדפן רצה באותו התהליך ביחד עם הדפדפן, ולכן תאורטית הקוד נגיש לאותו מרחב הכתובות.

  לא היינו רוצים שהקוד שאינו מהימן יוכל לגשת למרחבי הזיכרון של חלקים אחרים בתהליך המהימן.

- Interpreter עבור קוד שאינו Native.

  למשל, אם לא אמורה להיות גישה לרשת, ה־Interpreter לא יאפשר גישה לרשת.

  דוגמאות: JavaScript, JVM, NET CLR..

4.3 מימוש Confinement (בתוכנה)

כדי לממש Confinement, נצטרך לקבל החלטות רבות. ברגע שתכנית תבקש גישה למשאב, נצטרך להבין לפי מדיניות הבטיחות מה עושים.

התהליך הזה קורה בתוך ה־*Reference Monitor*. בכל פעולה חשודה, ה־Reference Monitor יבדוק אם הגישה מותרת, ואם לא יפעל בהתאם (למשל יהרוג את התהליך).

חשוב לשים לב לכך שה־Reference Monitor חייב להיות תהליך שאנו בוטחים בו, וצריך לעבוד בדיקות קפדניות מאוד נגד תקיפות אפשריות.

4.3.1 דוגמה – `chroot`

במערכות הקבצים של מערכות UNIX כבר יש הרשאות מובנות, שמספקות סוג של Confinement. תהליכים לא יכולים לגשת לקבצים ללא ההרשאות המתאימות. בפועל, האנשים שוכחים וטפל ולהגדיר הרשאות נכונות, ולעתים קרובות הקבצים נגישים (לפחות לקריאה) גם על ידי גורמים נוספים.

כדי לייצר את סביבת האבטחה הנכונה, המציאו את הפקודה `chroot`, המשנה את השורש (root) של מערכת הקבצים, ובכך מונעת גישה לקבצים שלא אמורים להיות נגישים. דוגמה לתקיפה:

```
chroot /home/guest
open("/etc/passwd", "r")
```

למעשה, בשורה השנייה פתחנו את הקובץ /home/guest/etc/passwd, ואז תכנות כמו su עושות מה שאנחנו רוצים (כי יש לנו שליטה מלאה על /home/guest/etc/passwd.
יחד עם זה, נוצרות בעיות אחרות עם chroot, כמו היעלמותן של התיקיות /proc, /sys, /dev. לכן, עולה הצורך ביצירתן מחדש ב־/home/guest. אבל אם יוצרים את /proc בתוך /home/users/guest, כבר אפשר לראות פרטים על תהליכים אחרים. אם ניגשים ל־/dev/sda, אפשר לשנות את התוכן של הכונן הקשיח כולו.
אם משתמש כלשהו הצליח להפוך ל־root, ואז קורא מתוך /home/guest/dev/sda, הוא כבר רואה את הכונן הקשיח כולו.

4.3.1.1 מנגנון ה־jail של Free-BSD

```
jail jail-path hostname IP-addr cmd
```

המנגנון מוסיף על פני chroot גם את היכולת להגביל גישות לרשת, והגבלה קשיחה יותר גם למערכת הקבצים.

4.3.2 System Call Interposition

ההנחה שלנו היא שכדי לתקוף ולגרום נזק, צריך לבצע קריאות מערכת (*System Calls*). למשל: unlink, open, write, socket, bind, connect, send ועוד. אם נפקח על קריאות המערכת של האפליקציה מסוימת, נוכל לתפוס תקיפות מתוך האפליקציה.
נוכל לממש בקרה כזו בכמה דרכים: ב־Kernel Space, ב־User Space או באופן היברידי שישלב ביניהם.

4.3.2.1 דוגמאות לניטור ב־User Space

דריסת LD_PRELOAD לרוב תכנות ב־UNIX לא מבצעות את קריאת המערכת ישירות באמצעות int 0x80, אלא הן קוראות לעטיפה שלהן (למשל הפונקציה write).
הרעיון שלנו הוא הוספת ספרייה[2] שלנו למשתנה הסביבה LD_PRELOAD, ושם נדרוס את קריאות המערכת.
הבעיה היא שתוכנה עשויה לבצע בעצמה ישירות את קריאת המערכת, ואז התכנה שלנו לא תדע על כך בכלל. הבעיה נובעת מהבעיה האינהרנטית שקיימת כאן: מי שכותב את ה־confinement יש לו בדיוק את אותו הכוח כמו מי שכפוף ל־confinement, ואז ניתן לעקוף את ה־confinement יחסית בקלות.

שכתוב דינמי של קוד התכנה (*Program Shepherding*) הבעיה היא שאפשרות זו מאוד קשה. אבל יחד עם הקושי, יש בה שימוש למכונות וירטואליות (לא נפרט כאן).

strace נשתמש בקריאת המערכת ptrace, שמקבלת מזהה תהליך (*Process ID*, או pid בקיצור), ומתעוררת רק כאשר התהליך המנוטר מבצע קריאת מערכת.
סיבוכים:

- עוקבים אחרי כל קריאות המערכת של התהליכים, דבר המכביד מאוד על המחשב.
- ה־Monitor צריך לזכור את ה־*Current Working Directory* (או בקיצור CWD) של התהליך המנוטר.

[2] או *Shared Object* בקובץ so..

- ה־Monitor צריך להתפצל (fork) ביחד עם התהליך המנוטר.
- הרבה סיבוכים בלתי־צפויים של מערכת ההפעלה שנובעים מתכונות (*Features*) מתקדמות של מערכת ההפעלה (כמו למשל *File Descriptors*, chroot, *core-dump* ועוד).

  נסביר קצת על File Descriptors (או בקיצור fd): תהליך יכולים לקבל fd מתהליך אחר. ה־Monitor לא יודע אם לתהליך יש הרשאות גישה לקובץ, כי אנחנו לא יודעים את שם הקובץ (או במילים אחרות, אנחנו לא יודעים לאן מתייחס ה־fd). כדי להתגבר על הבעיה הזאת, צריך לתקשר עם ה־Monitor של התהליך האחר (במידה וקיים כזה),
- שימוש ב־ptrace עשוי לגרום לבעיות Integrity של התהליך המנוטר.

  למשל, מה קורה אם מנסים לפתוח קובץ שאסור לפתוח? להרוג? או להמשיך בלי לפתוח?

4.3.2.2 דוגמאות לניטור היברידי

Systrace הינו תכונה מודרנית ב־Linux, שרצה בתוך ה־Kernel Space. Systrace רק מעביר קריאות מערכת ל־Monitor שרץ ב־User Space, ובכך גם חוסך *Context Switch* בכל קריאת מערכת.
יש לזה גם יתרונות נוספים:

- לא צריך לשמור את ה־CWD.
- מאפשר גישה למידע פנימי של מערכת ההפעלה (כמו לאן מצביע ה־fd).
- מקבל *Policy Files*, שיודע גם להתעדכן כאשר קוראים ל־execve. למשל:

```
path allow /tmp/*
path deny /etc/passwd
network deny all
```

  Systrace יודע לייצר Policy File כזה בעצמו מהתבוננות בריצה אופיינית של התהליך, ולאחר מכן לחסום פעולות חריגות. כמובן שקשה מאוד להבין דבר כזה. למשל, דפוסי השימוש בדפדפן עשויים להשתנות מיום ליום (לא כל יום גולשים לכל האתרים, או ניגשים לכל הקבצים של הדפדפן). זו הסיבה לכך ש־Systrace אינו נפוץ.
- Systrace מאפשר לסרב לקריאת מערכת, במקום להרוג את התהליך הזדוני, ובכך לשמור על ה־Integrity של התהליך המנוטר.

4.3.3 דוגמה מודרנית – *Native Client*

חברת Google רוצה שנבלה את כל חיינו בדפדפן, ולכן רוצים לאפשר הרצה של קוד בתוך הדפדפן, אפילו אם הקוד אינו מהימן ואינו חתום. בשביל היעילות, אפילו היינו רוצים להריץ *Native Code* (כלומר קוד מכונה).
שפת Java מספקת איזון הולם יחסית בין קוד מכונה לאבטחה בהרצת קוד שאינו מאובטח. אולם Java עדיין אינה מהירה מספיק, ולא יודעת להשתמש בתכונות של המעבד

כמו SSE וגרפיקה. התשתית ActiveX של חברת Microsoft הינה תחליף ראוי, אולם Microsoft מתחרה של Google, ולכן Google הלכו לכיוון אחר.

לכן, Google רוצים לאפשר להריץ קוד Native x86 בתוך הדפדפן. Google קראו לפתרון זה *Native Client* (או *NaCl* בקיצור). הקוד ירוץ כ־Thread בתוך תהליך הדפדפן, והוא צריך לרוץ ב־sandbox.

אפשר להשתמש ב־Binary Dynamic Translation, אבל כאמור, זה מאוד קשה למימוש. לכן, Google מאפשרים שימוש בגרסה מוגבלת של x86, שלא מאפשרת תקיפות (לטענתם לפחות).

מה אם הקוד ינסה לגשת למרחב זיכרון פנימי של הדפדפן, ולהוציא משם מידע פרטי? כדי למנוע תקיפות מסוג זה, Google הוסיפו מנגנון הדנה חיצוני נוסף: שימוש בסגמנטים של זיכרון של x86. פעם, בארכיטקטורות ישנות, היו סגמנטים, שמאפשרים גישות לזיכרון וירטואלי, או לחלקים ממנו. יכולת זו השתמרה גם בארכיטקטורות מודרניות, כגון x86, אך אינה בשימוש נפוץ. לכן, Google מאפשרים פקודות `mov` לסגמנטים בלבד. זה עולה קצת בביצועים, אבל לא יותר מדי.

תקיפה אפשרית: פקודות x86 הן באורך משתנה. אם נשרשר סוף של פקודה לגיטימית להתחלה של פקודה לגיטימית, אנחנו עשויים לקבל פקודה שאינה לגיטימית. את זה Google פתרו באופן הבא: גודל ה־*Basic Block* חייב להיות קבוע, וקפיצות (*jumps*) בתוכנה חייבות להיות להתחלה של Basic Block.

4.4 Confinement בעזרת מכונות וירטואליות

ההנחות שלנו:

1. תכנות זדוניות עלולות להזיק למערכת ההפעלה ולאפליקציות בתוך המכונה הווירטואלית.
2. תכנות זדוניות לא יכולות לגלוש החוצה מהמכונה הווירטואלית – לא למכונה המארחת ולא למכונה וירטואלית שכנה.

ההנחות דורשות מימוש שיגן גם על עצמו ושלא יכיל באגים.

בעיה: ערוצים סמויים (Covert Channels) – תקשורת שלא אמורה לקרות בין שתי מכונות וירטואליות. למשל, כדי לשלוח ביט $b \in \{0,1\}$, בשעה 1:30 ה־CPU צריך לעבוד הרבה או לנוח.

נחשוב רגע על אנטי־וירוסים. אם האנטי־וירוס רץ על אותו המחשב (מגרש המשחקים) ביחד עם הווירוס שאותו הוא מנסה לתפוס, אז הוא מאוד פגיע. למשל, אפשר לכבות אותו, אפילו אם הוא רץ ב־Kernel Space. אפשר לשים את האנטי־וירוס מחוץ למכונה, למשל בתקשורת. אבל גם אז אי אפשר לדעת מה קורה מרחוק, ואי אפשר לנטר את כל התקשורת (למשל תקשורת מוצפנת מהווה בעיה).

אפשר גם להריץ אנטי־וירוס כחלק מה־VMM[3], ואז האנטי־וירוס רץ בסביבה שונה וחזקה יותר מהווירוס. אז גם האנטי־וירוס נגיש לכל החומרה הווירטואלית של המכונות הווירטואליות. למשל, אם יש וירוס שמסתיר את עצמו ב־`ps` ו־`netstat`. אז ה־VMM ומערכת ההפעלה הפנימית מגלים הבדל ברשימות, וה־VMM יכול להרוג או להקפיא את המכונה הווירטואלית. כלומר, ה־VMM מכיר את כל התהליכים וה־socket־ים שקיימים באמת. אם מערכת ההפעלה הפנימית מריצה `ps`, ומגלים שחסר תהליך, פה נחשוד.

בדיקות נוספות שה־VMM יוכל לעשות:

[3] ראשי תיבות של Virtual Machine Manager.

- בדיקת Integrity – מריצים Hash על הקוד של המכונה הווירטואלית.
- בדיקת Integrity של מערכת ההפעלה האורחת (למשל לצורך זיהוי שינוי ב־System Calls Table).
- הרצת מזהה חתימות של וירוסים במכונה וירטואלית מבחוץ.
- זיהוי כניסה ל־Promiscuous Mode במכונה אורחת.

  Promiscuous Mode הוא מצב שבו כרטיס הרשת מדווח למערכת ההפעלה על כל הפקטות שהוא רואה, ולא מסנן רק עבור הפקטות של המחשב הנוכחי. זוהי פעולה שלרוב מעידה על כל שמישהו מנסה לראות תעבורה של מחשבים אחרים, ולכן היא חשודה.

  נשים לב לכך שהכניסה ל־Promiscuous Mode מתאפשרת רק דרך החומרה, וה־VMM מדמה את החומרה של המכונה הווירטואלית, ולכן קל מאוד לראות כניסה ל־Promiscuous Mode.

יתרון חשוב מאוד של מכונות וירטואליות הוא היכולת לנתח וירוסים בזמן פעולה. במכונה אמיתית, אם אנטי־וירוס מזהה וירוס, הוא ינסה לעצור אותו בכל מחיר. אולם מכונה וירטואלית מאפשרת הקפאה, לצורך ניתוח הווירוס בעודו פועל.

אפשר גם לעשות למכונה וירטואלית *Snapshot* כדי לחזור אחורה בזמן, ואפילו לעבור פעולה־פעולה ושחזור המצב של המכונה – *Rewind and Replay*. באופן מפתיע, המימוש של זה לא כל כך יקר (מבחינת משאבי זיכרון) כמו שזה נשמע.

4.4.1 רעיון לווירוס – Subvirt

אם הווירוס מגיע למכונת הקרבן, הוא יכול להסתתר ב־VMM, ואז הוא בלתי נראה למזהי הווירוסים בתוך המכונות הווירטואליות.

הווירוס Blue Pill יושב על הכונן הקשיח ב־Boot Record, ומערכת ההפעלה כלל לא מודעת לכך שהיא רצה מתוך הווירוס.

4.4.2 זיהוי VMM

האם מערכת הפעלה יכולה לזהות שהיא רצה בתוך VMM?
ברמת האפליקציה:

- מזהי וירוסים יכולים לזהות VMBR.
- וירוס רגיל יכול לזהות VMM.
- תוכנה שמתמפה ישירות לחומרה (למשל מערכת ההפעלה) יכולה לסרב לרוץ מעל VMM.
- מערכות DRM עשויות לסרב לרוץ מעל VMM.

למה שמערכת הפעלה תסרב לרוץ מעל VMM? יש כל מיני סיבות. למשל, יכול להיות שהמשתמש בחר לרוץ מעל וירוס הדומה ל־Blue Pill, כדי להריץ מערכת הפעלה שאינה חוקית (למשל מערכת הפעלה שדורשת רישיון בתשלום), או לפרוץ תוכן DRM.
איך לזהות ריצה מעל VMM?

- בדרך כלל מכונות וירטואליות מדמות חומרה ישנה.
- הביצועים של המכונות הווירטואליות פחות טובים מהביצועים של מכונה רגילה.
  למשל, פגיעות תכופות במטמון (Cache) של המכונה (הפיזית).
- ה־TLB של המכונה הווירטואלית משותף עם ה־TLB של המכונה המארחת. המכונה האורחת יכולה לזהות שה־TLB קטן משמעותית מהמצופה.

בשורה התחתונה, ה־VMM המושלם לא קיים, כלומר נוכל לזהות ריצה מעל כל VMM.

פרק 5

Homomorphic Encryption and Computational Proofs[1]

5.1 מוטיבציה

האם ניתן לשלוח חישוב למקום שאנו לא בוטחים בו? כלומר, האם יש אפשרות שנשלח את החישוב למקום אחר, ונבטיח את הנכונות שלו ואת סודיות המידע בחישוב?

למשל, האם אפשר לבצע חיפוש ב־Google, בלי ש־Google ילמד עלינו? כלומר, נשלח ל־Google שאילתה מוצפנת, ונקבל תשובה מוצפנת, בלי ש־Google ידע מה תוכן השאילתה ומה תוכן התשובה (כי Google לא יודעים את המפתח הסודי)?

דוגמה נוספת לשימוש היא מחשוב ענן: שולחים קלט x מוצפן ותוכנית P, והשרת יחשב את ההצפנה של $P(x)$ (בלי לפענח את ה־x המוצפן).

הגדרה מערכת הצפנה הומומורפית מכילה $(\mathrm{Key}, \mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec}, \mathrm{Eval})$, כך ש־Eval מאפשר חישוב על מידע מוצפן.

תכונת הנכונות של Eval: אם $c = \mathrm{Enc}(\mathrm{PK}, x)$ ו־$c' = \mathrm{Eval}(\mathrm{PK}, c, P)$, אזי $\mathrm{Dec}(\mathrm{SK}, c') = P(x)$.

אנחנו למעשה כבר מכירים מימוש לזה (שכולל הצפנה): Eval יכול לשרשר את c ו־P, ו־Dec יעשה את כל העבודה. כלומר, בהינתן שיטת הצפנה CCA-1, ניתן לבנות מערכת הצפנה הומומורפית.

הבעיה בפתרון הזה היא עניין של יעילות: Eval למעשה לא מבצעת את פעולת החישוב. כדי למנוע מצב כזה, נגדיר את תכונת הקומפקטיות: האורך של c' צריך להיות בלתי־תלוי בגודל של P.

נגדיר גם את תכונת הבטיחות: סמנטית.

עד לא מזמן, לא ידעו איך להשיג הומומורפיות מלאה. ידעו לעשות רק הומומורפיות מוגבלת.

למשל, אם נצפין את m_1 ו־m_2 עם RSA, נקבל c_1 ו־c_2 בהתאמה, כאשר $c_i = m_i^3$ (כמובן עם מודולו). נוכל לחשב באופן הומומורפי את $m_1 \cdot m_2$ באמצעות חישוב של $c_1 \cdot c_2$.

[1] סיכום לשיעור שהתקיים בתאריך 20.11.2012.

אלגוריתם 1.5 הצפנה הומומורפית עם מפתח פרטי

1. נבחר מפתח פרטי – מספר אי־זוגי p.
2. כדי להצפין את הביט $b \in \{0,1\}$:
   (א) נבחר כפולה "גדולה" של p, למשל $p \cdot q$.
   (ב) נבחר מספר זוגי "קטן" $2r$ (רעש).
   (ג) הסתר יהיה: $c = q \cdot p + 2r + b$.
3. כדי לפענח את הסתר c, נסתמך על העובדה הבאה:

$$c \equiv 2r + b \pmod{p}$$

לכן, $(c \mod p) \mod 2$ הוא הביט הגלוי b.

למה זה עובד? נשים לב:

$$c_1 \cdot c_2 = m_1^3 \cdot m_2^3 = (m_1 \cdot m_2)^3$$

למעשה, הבטיחות לא בהכרח נשמרת. למשל, ב־CCA-2: נבקש מהחבר הטלפוני לפענח את $\tilde{c} \cdot x^3$, כאשר $\tilde{c} = c_1 \cdot c_2$. נקבל $\tilde{p} \cdot x$. אבל $\tilde{p} = m_1 \cdot m_2$ היא התשובה לחישוב ההומומורפי, ולכן נפגעה כאן הבטיחות.

זוהי למעשה הוכחה לכך שמערכת הצפנה הומומורפית לא יכולה לקיים CCA-2. במילים אחרות, אם מערכת ההצפנה היא CCA-2, אנחנו בוודאות לא נוכל לקיים הומומורפיות. עם אנחנו בכל זאת רוצים הומומורפיות, נצטרך לוותר על חוזקת מערכת ההצפנה.

באופן דומה, במערכות כמו GM ו־Paillier אפשר לעשות הומומורפיות עם חיבור. החיפוש האמיתי הוא אחר מערכות שיהיו הומומורפיות גם לכפל וגם לחיבור. רק בשנת 2009 נמצאה חבורת השריגים לביצוע הומומורפיות חיבורית וכפלית.

הפתרון מאוד מסובך וקשה למימוש, אבל קיים. נתאר כאן גרסה מפושטת של הפתרון.

5.2 הצפנה הומומורפית עם מפתח פרטי

אלגוריתם 2.5 מתאר הצפנה הומומורפית עם מפתח פרטי.

נשים לב שתיתכן בעייה בפענוח: אם r גדול מדי, יש לנו בעיה, כי בתוצאה עשוייה להיות השפעה על $p \cdot q$.

כדי למנוע בעיות מהסוג הזה, נקבע פרמטר בטיחות n שיוגדר על ידי:

- p יהיה בגודל של n^2 ביטים.
- q יהיה בגודל של n^5 ביטים.
- r יהיה בגודל של n ביטים.

מבחינת נכונות – הפרמטרים מבטיחים אותה.

למה q צריך להיות בגודל של n^5 ביטים? בשביל הבטיחות, כדי לרווח את מרחב הסתרים (וליצור בתוכו סתרים שאינם אפשריים).
איך נחבר ונכפיל ביטים מוצפנים? אם:

$$\begin{aligned} c_1 &= q_1 \cdot p + 2r_1 + b_1 \\ c_2 &= q_2 \cdot p + 2r_2 + b_2 \end{aligned}$$

אזי:

$$c_1 + c_2 = (q_1 + q_2) \cdot p + 2\,(r_1 + r_2) + (b_1 + b_2)$$

בחרנו את הפרמטרים כך ש־$2\,(r_1 + r_2) + (b_1 + b_2)$ יהיה באיזור של p, ולכן הנכונות מתקיימת.
בנוגע לכפל:

$$\begin{aligned} c_1 \cdot c_2 &= (c_2 \cdot q_1 + c_1 \cdot q_2 + q_1 \cdot q_2 \cdot p) \cdot p \\ &+ 2 \cdot (r_1 \cdot r_2 + r_1 \cdot b_2 + r_2 \cdot b_1) + b_1 \cdot b_2 \end{aligned}$$

נבחן את הגודל של הרעש: $r_1 \cdot b_2$ ו־$r_2 \cdot b_1$ הם בגודל של n ביטים לכל היותר, ו־$r_1 \cdot r_2$ הוא בגודל של $2n$ ביטים.
מה הבעיות במימוש הזה?

- הסתר גדל בכל פעולה.
- הרעש גדל בכל פעולה.

זאת הסיבה לכך שהפתרון הוא Somewhat Homomorphic. הפתרון לא מסוגל לחשב כל פונקציה P כפי שהוא. לכל פונקציה P צריך למצוא פרמטר בטיחות n שיתאים לה, ולפעול באמצעותו.

5.3 הצפנה הומומורפית עם מפתח פומבי

אלגוריתם 2.5 מתאר הצפנה הומומורפית עם מפתח פומבי.
למה בהצפנה עושים $\pmod{x}_0$?

- כדי להקטין את הגודל של הסתר c.
- כדי להגדיל את הבטיחות.

איך זה עוזר לבטיחות?

$$\begin{aligned} c &= \sum_{i \in S} x_i + 2r + b \pmod{x_0} \\ &= p \cdot \cancel{\left[\sum_{i \in S} q_i \right]} + 2 \cdot \left[r + \sum_{i \in S} r_i \right] + b - k \cdot x_0 \\ &= p \cdot \cancel{\left[\sum_{i \in S} q_i - k \cdot q_0 \right]} + 2 \cdot \left[r + \sum_{i \in S} r_i - k \cdot r_0 \right] + b \end{aligned}$$

אלגוריתם 2.5 הצפנה הומומורפית עם מפתח פומבי

1. נבחר מפתח סודי – מספר אי־זוגי p בגודל של n^2 ביטים.
2. המפתח הפומבי יהיה:

$$(x_0, \ldots, x_t) = [q_0 \cdot p + 2r_0, \ldots, q_t \cdot p + 2r_t]$$

   את $\{q_i\}$ ו־$\{r_i\}$ נבחר כמו קודם.
   למעשה, אלו הן $t+1$ הצפנות של 0. בשביל הנוחות, נניח כי x_0 הכי גדול.
3. הפענוח יהיה זהה להצפנה הומומורפית עם מפתח פרטי: $(c \mod p) \mod 2$.
4. ההצפנה:
   (א) נבחר תת־קבוצה אקראית $S \subseteq \{1, \ldots, t\}$.
   (ב) הסתר יהיה:

$$\sum_{i \in S} x_i + 2r + b \pmod{x_0}$$

   (ג) Eval יהיה כמו קודם.

היינו רוצים ש־$q_0 \sim \sum_{i \in S} q_i$. מכיוון ש־$x_0$ הוא הגדול ביותר, $q_0 > q_i$, ולכן $k \geq t$, ולכן הרעש

$$r + \sum_{i \in S} r_i - k \cdot r_0$$

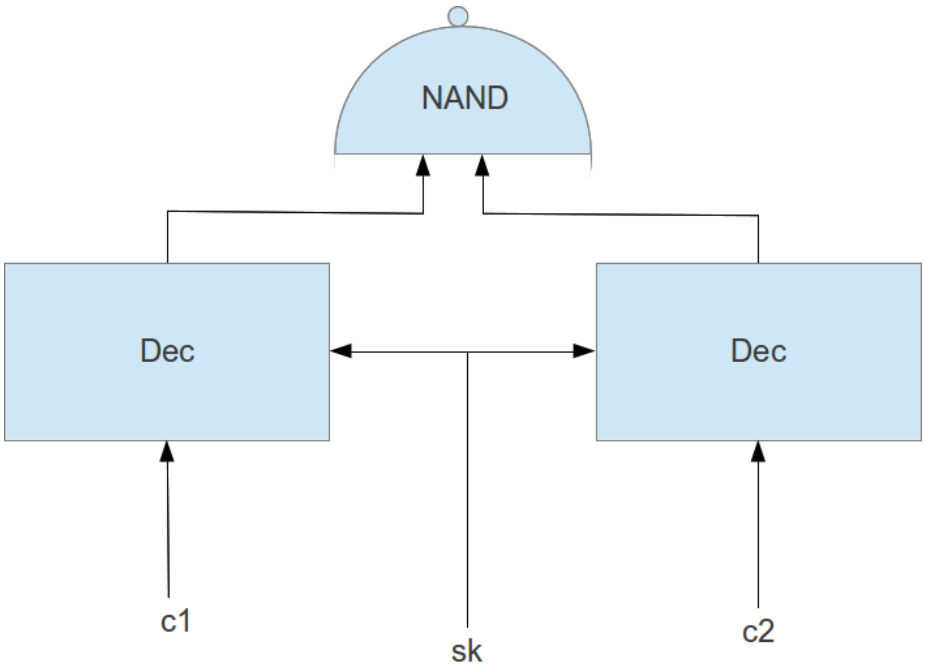
הוא רעש קטן יחסית וזניח.

5.4 בעיית ה־gcd המקורב

זוהי בעיה חישובית, שתגדיר לנו מאפיין בטיחות.
הקלט: שלושה מספרים P, Q ו־R. בוחרים $p \in \{0, \ldots, P\}$, מכפילים אותו בהרבה רעש מהצורה $q_i \cdot p + r_i$ כאשר $q_i \in \{0, \ldots, Q\}$ ו־$r_i \in \{0, \ldots, R\}$, ומבקשים מהיריב לחשב את p בהסתברות טובה.
ההנחה אומרת שעבור בחירת פרמטרים צנועים, ליריב יהיה קשה מאוד לחשב את p.
אפשר להוכיח שהבעיה שקולה להגדרת הבטיחות הסמנטית.

5.5 מעבר מ־Somewhat Homomorphic להצפנה הומומורפית מלאה

נוכיח את המעבר בכך שהפעולה NAND היא פעולה אוניברסלית. איור 5.1 מתאר איך עושים את זה.



איור 5.1: יצירת הצפנה הומומורפית שלמה מ־Somewhat homomorphic

נשים לב כי האיור מתאר מעגל (נקרא לו α), שאותו אפשר לקודד כתוכנית ולשלוח ל־Eval. לאחר הקריאה ל־Eval, נקבל את $\left(\left(\bar{c}_1, \bar{c}_2, \overline{\mathrm{sk}}\right), \alpha\right)$ Eval, כאשר:

$$\begin{aligned} \bar{c}_1 &= \mathrm{Enc}_{\mathrm{pk}}\left(c_1\right) \\ \bar{c}_2 &= \mathrm{Enc}_{\mathrm{pk}}\left(c_2\right) \\ \overline{\mathrm{sk}} &= \mathrm{Enc}_{\mathrm{pk}}\left(\mathrm{sk}\right) \end{aligned}$$

Eval יחזיר לנו $\bar{c}_3$, כאשר:

$$\begin{aligned} \mathrm{Dec}_{\mathrm{sk}}\left(\bar{c}_3\right) &= p_1 \mathrm{NAND}\ p_2 \\ p_1 &= \mathrm{Dec}_{\mathrm{sk}}\left(c_1\right) \\ p_2 &= \mathrm{Dec}_{\mathrm{sk}}\left(c_2\right) \end{aligned}$$

משפט כל מערכת *Somewhat Homomorphic* יכולה להפוך ל־*Fully Homomorphic*. אבל למעשה, צריך לדרוש גם *Bootstrappable*.

פרק 6

[1]Power Analysis Attack

6.1 מצפין AES

מצפין בלוקים של 128 ביטים.
האלגוריתם מורכב מ־4 שלבים:

- BytesSub
- ShiftRows
- MixColumn
- AddRoundKey (AddKey)

6.2 מבוא להנדסת חשמל

כשבונים חומרה, יש שני סוגים עיקריים:

- ASIC (Application Specific).
רכיב שלוקח זמן רב עד לייצור הראשון, ואחריו קל ליצר ביצור המוני. המעגלים החשמליים בו מתאימים לאפליקציה ספציפית (ומכאן שמו), ולא ניתנים לשינוי.
- Microcontroller.
הרכיב הוא כמו מעבד – כלומר הוא מקבל פקודות (Intructions) ומידע (Data), ורץ עליהם.

6.3 Power Analysis Attack

איך נעשה Power Analysis Attack?
לכל קלט (גלוי) ופלט (סתר), נחזיר גם *Power Trace*, שזה בעצם מעקב מדידות הספק המכשיר, כאשר ההספק נמדד ברצף לאורך זמן, בהפרשי זמן מאוד מקוד קטנים.
ההנחות שלנו כשאנחנו עושים את זה:

[1] סיכום חלקי לשיעור שהועבר בתאריך 27.11.2012.

- ההספק משתנה ביחס לזמן.
- ההספק משתנה לפי המידע שמעבדים.
- ההספק תלוי בפקודה שמריצים.

עד כאן הסיכום החלקי של השיעור הזה.

פרק 7

Reverse engineering[1]

הסיכום אינו זמין.

[1] שיעור שהתקיים בתאריך 04.12.2012.

פרק 8

Integrity on Untrusted Platforms[1]

המטרה שלנו היא להבטיח שהחישוב שלנו יתבצע נכון, כאשר אנחנו לא סומכים על הפלטפורמה שמריצה את החישוב שלנו.

יכול להיות שהפלטפורמה גם מדליפה מידע, אז היינו רוצים להצפין עד כמה שניתן.

יש כמה גישות להשגת המטרה הזאת:

- מאוד להיזהר.

  הרעיון הוא לוודא הרבה וגם לאמת את הזהות. כמו כן, האחריות שלנו היא לוודא שכל המרכיבים במערכת שלנו, מהשבב הקטן של החומרה ועד התוכנה הכוללת שרצה על החומרה, כולם מאובטחים: אנחנו מכירים את המפעל והיצרן, אנחנו כתבנו את התוכנה בעצמנו, ועוד.

  הגישה דורשת משאבים רבים, ובפועל אינה פיזיבילית במיוחד.

- וידוא שמשתמשים במודול מאובטח (*Trusted Platform Module – TPM*).

  ראינו דוגמה לדברים כאלה בשיעורים הקודמים. הרעיון הוא להשתמש במודול חומרה מאובטח, שיוודא בשבילנו שהכל מתנהל כשורה.

  הבעיה העיקרית היא שלעתים קרובות קל לעקוף את ההגנה הזאת.

- פרוטוקולים קריפטוגרפיים שמאפשרים חישוב רב־משתתפים, וגם מאפשרים הוכחת נכונות למידע.

  כאן, במקום לנסות לאבטח מחשב בודד, היינו רוצים לוודא קיום השאלה האמיתית: איך אנחנו יודעים שהמחשב חישב את המידע שאותו הוא היה אמור לחשב? כלומר, איך ניתן לוודא את נכונות וסודיות החישוב? ראינו דוגמה לדבר הזה בפרק 5.

  יש פרוטוקולים שמאפשרים ביצוע חישוב על נתונים שמפוזרים בין הרבה גורמים, כך שהמידע של כל אחד מהגורמים נשאר גלוי רק אליו, אולם התוצאה המשותפת ניתנת להבנה על ידי כולם. יש גם פרוטוקולים כאלה שמאפשרים חישובים כאלה, גם במקרה שחלק מהמשתתפים הם "רמאים".

[1] סיכום לשיעור שהתקיים בתאריך 11.12.2012.

8.1 דוגמה לווידוא נכונות עם שלושה משתתפים

יש לנו שלושה משתתפים: Alice, Bob ו־Carol.

נניח כי ל־Alice יש את הקלט x עם הפונקציה F. אז היא מחשבת את $y = F(x)$. Bob רואה את y, ונתונה לו פונקציה G. הוא מחשב את $z = G(y)$, ושולח את z ל־Carol. עכשיו Carol שואלת את עצמה: "איך אני יכולה לדעת בוודאות ש־$z = G(F(x))$?". מה Carol יכולה לעשות?

אם Carol יודעת את x, F ו־G, היא יכולה לחשב מחדש את $z' = G(F(x))$, ולבדוק אם $z' = z$.

הבעיה היא שזה מאוד בזבזני, כי מישהו כבר חישב את z, ולא צריך לחשב אותו מחדש כדי לדעת אם הוא נכון. למשל, טלפון סלולארי לא יחשב מחדש את מה שהענן כבר חישב, כדי לוודא את נכונות החישוב (אם הפלאפון יכול היה לחשב את החישוב, הוא לא היה שולח אותו לענן מלכתחילה).

בעיה נוספת היא ש־Carol צריכה לדעת את x, F ו־G. לרוב זה לא המצב, ואם זה המצב אז זה לא המקרה המעניין קריפטוגרפית.

נרצה לעשות משהו שדומה להגדרה של $\mathcal{NP}$. ניזכר בהגדרה לשפה L ב־$\mathcal{NP}$:

$$\alpha \in L \Leftrightarrow \exists x. M(\alpha, x) = 1$$

במקרה שלנו, מכונת הטיורינג M מחשבת את $G(F(x))$. היינו רוצים למצוא Hash α כך ש־$z = G_\alpha(F(x))$.

לרוב, החישוב יהיה קצת יותר מורכב: במקום Alice ו־Bob לפני Carol, יכולים להיות הרבה יותר אנשים בדרך, עם קשרי גומלין מורכבים יותר בשרשרת החישוב. הרבה פעמים, אנחנו לא יודעים את הפונקציות של החישוב (F ו־G), ואז אנחנו צריכים לוודא מול כולם שהם חישבו את הפונקציות הנכונות, או לדמות המון חישובים של כלל המשתתפים במערכת.

היינו רוצים משהו הרבה יותר קומפקטי להוכחת הנכונות. בדוגמה הקודמת, Bob יחשב גם π_z, שיהיה מחרוזת הוכחה לכך ש־$z = G(F(x))$, ואז Carol יכולה לוודא את z באמצעות π_z.

הדרישות שלנו מ־π_z:

- קטן (פולי־לוגריתמי בחישוב של z).
- אפשרות לווידוא יעיל על ידי Carol.

יש פתרונות שהם תלויי בעיה (F ו־G). למשל: ניתן לוודא פירוק של מספר על ידי הכפלת כלל הראשוניים בתוצאה (שזוהי אכן פעולה זולה יותר מפירוק המספר הגדול). אולם היינו רוצים למצוא פתרון גנרי, שלא יהיה תלוי בסוג החישוב.

איך אפשר לעשות את זה? השיטה הנאיבית היא ש־Carol תדע את x, F ו־G, ומחרוזת ההוכחה תהיה המחרוזת הריקה. Carol תוכל לחשב מחדש את $z' = G(F(x))$, וכך לוודא את נכונות הפלט. אולם זה לא ריאלי מהסיבות שתיארנו קודם.

נשים לב שבמקרה שלנו, Bob לא יודע ש־$y = F(x)$. הוא יכול לחשב מחדש את y, אבל הוא היה רוצה לקבל הוכחה על נכונות החישוב ש־Alice ביצעה.

יש פתרון למצב הזה: *Incrementally-Verifiable Computation*. Alice תחשב את $y = F(x)$ ואת π_y, ואז Bob יחשב את $z = G(y)$ ואת π_z. כל משתתף מכין מחרוזת הוכחה למי שישתמש בתוצאה שלו. מחרוזת ההוכחה צריכה לקיים את הדרישות שדרשנו במקרה הקודם.

נחדד את הדרישה לפולי־לוגריתמיות של מחרוזת ההוכחה: אם החישוב של $y = F(x)$ לוקח t צעדים, אז אנחנו דורשים:

$$|\pi_y| \leq \log^c t$$

למה אנחנו דורשים פולי־לוגריתמיות ולא לוגריתמיות? כי זה מה שאנחנו יודעים לבצע היום, אבל לפחות הקבוע c הוא קטן.

נדרוש גם שהזמן הדרוש ליצירת π_y יהיה פולינומי ב־t, והזמן שלוקח לוודא את y לפי π_y יהיה פולי־לוגריתמי ב־t (למעשה פולינומי בגודל של π_y).

למעשה, Bob לא יכול להסתפק בהוכחה ש־$z = G(F(x))$. הוא גם צריך להוכיח שהוא קיבל הוכחה ש־$y = F(x)$. כלומר, π_z יהיה הוכחה לכך ש־$z = G(y)$, ו־$y = F(x)$.

8.2 חישוב מבוזר כללי

בחישוב מבוזר, כל אחד מהמשתתפים מחליף הודעות עם המשתתפים האחרים ומבצע חישובים בעצמו.

מעבר לכך, הקלטים לחישובים עשויים להשתנות. למשל, קלט מהמשתמש, אקראיות ועוד. נשאלת השאלה: אם החישוב הוא לא שרירותי, איך אנחנו מגדירים את הנכונות שלו? ואיך מוכיחים אותה? נרצה להכליל את המושג של הנכונות גם לחישוב שאינו שרירותי.

מתכנן מערכת מגדיר נכונות באמצעות Compliance Predicate $C(\texttt{in}, \texttt{code}, \texttt{out})$, שצריך להתקיים בכל אחד מהצמתים. נאמר שצומת הוא Compliance אם הוא מקיים את C. נאמר שהחישוב הוא Compliance אם ה־Compliance מתקיים בכל אחד מהצמתים. כלומר, נגיד שההודעה m_{out} היא C-Compliant אם קיים גרף של צמתים מחשבים, שכל אחד מהם הוא C-Compliant.

דוגמאות למחרוזות הוכחה:

- הפלט הוא התוצאה של חישוב נכון של תוכנה מסוימת.

  כדי לוודא את ה־Compliance הזה, נריץ את התוכנה בעצמנו שוב, ונראה שקיבלנו פלט נכון.

- הפלט הוא התוצאה של חישוב נכון של תוכנה מסויימת החתומה על ידי מנהל מערכת כלשהו.

  כאן נעזרים באלגוריתם קריפטוגרפי, שמאפשר לנו חתימה דינאמית על תוצאות.

- הפלט הוא התוצאה של חישוב נכון של *Type-safe program* או תוכנה עם *Valid format proof*.

בחישוב מבוזר, כדי להוכיח Compliance, כל הודעה שעוברת בין המשתתפים מכילה מחרוזת הוכחה שמעידה על כך ש"עד כאן הכל בסדר". החישוב המבוזר מתבצע כמו קודם, רק שכל משתתף מחשב על הדרך גם מחרוזת הוכחה שמעידה על ה־Compliance.

8.2.1 דוגמה: מניעת זליגה בין חלקים של מערכת

יש חישוב שמכיל שני סוגים של קלטים: קלטים סודיים וקלטים בלתי־מסווגים (להלן בלמ"ס). כל קלט בלמ"סי נחתם על כך שהוא אינו מסווג. היינו רוצים לוודא שלא מוזרמים קלטים

מסווגים לתוך החלק הבלמ"סי של החישוב, או לוודא שהפלט הבלמ"סי אינו תלוי בקלטים הסודיים.

למעשה, C מאפשר לנו:

- חתימה של קלטים בלמ"סיים.
- אכיפה של Information Flow Control: נסכים לסמן פלט של חישוב כבלמ"ס רק אם כל הקלטים שלו הם בלמ"סיים.

ניתן לאכוף את יציאת הפלטים מהחישוב רק אם כל הקלטים שעליהם החישוב מסתמך הם בלמ"סיים.

8.2.2 דוגמה: סימולטורים ו־MMO[2]

יש סימולטורים מבוזרים: מודלים פיזיקליים, או עולמות וירטואליים.

איך המשתתפים יכולים להוכיח שהם שמרו על החוקים של העולם הווירטואלי? למשל, איך הם יכולים להוכיח שהם לא המציאו לעצמם מטבעות, או שהלכו דרך הקיר?

אפשר לאכוף את החוקים האלה אצל הלקוח. אבל למעשה אי אפשר לסמוך על הלקוח שיבצע את הדברים האלה כמו שצריך. לכן, ללקוח נותנים לבצע את החישובים הפחות־חשובים (כמו לרנדר את הגרפיקה, או לבצע חישובים של זמן־אמת), ואת החישובים הקריטיים (כמו בדיקת קיום של כסף) מבצעים על השרתים.

הבעיה בפתרון הזה היא העומס החישובי שהוא משרה על השרתים. זה מגביל את מספר המשתתפים האפשריים במשחק, ולמעשה מאט מאוד ומעיק על חווית המשתמש.

היינו רוצים לבזר את האכיפה של החוקים האלה, ולהעביר את הלוגיקה ללקוחות. למשל, היינו רוצים ששני שחקים ישחקו במטוס, כשהם מנותקים מהאינטרנט, ולאחר מכן כשכל אחד מהם יתחבר לאינטרנט בנפרד, הם יוכלו להוכיח את ההתקדמות שלהם במשחק גם כשלא היו מחוברים לאינטרנט.

זה סוג של C-Compliance שאוכף את חוקי הפיזיקה של העולם הווירטואלי.

8.2.3 דוגמאות נוספות

- אכיפת חוקים במערכות הפיננסיות.
- קוד שמכיל הוכחות לעצמו.

  למשל, חתימה על מנהלי התקנים (Drivers) שיכולים לרוץ ב־Ring 0. דוגמה נוספת ניתן למצוא בקוד NaCl של Google.
- סיווג של הודעת דואר אלקטרוני כדואר זבל.

  למשל, אם קיבלתי דואר אלקטרוני ממישהו שנתתי לו את הכתובת שלי, אז כנראה שלא מדובר בדואר זבל (לפחות לא בהגדרה הקלאסית שלו).

8.3 בניית מערכת שמכילה הוכחה למידע

הגדרה פשפט הוא טענה שעבורה יש עד תקין לנכונות הטענה.

עד תקין מוגדר על ישי פרוצדורת וידוא יעילה, שבהינתן המשפט ועד w, מחליטה אם לקבל את ההוכחה.

[2] ראשי תיבות של Massively Multiplayer Online.

כלומר:

$$\texttt{Verify}_{\mathcal{NP}}(w) = 1 \Rightarrow \exists \text{ valid } w$$

למעשה, ההגדרה דומה להגדרה של $\mathcal{NP}$.
ניתן גם להחליש את הדרישה: מוכיח פולט מחרוזת הוכחה π כך שאם המשפט אינו נכון, ההסתברות שהמוודא ישתכנע קטנה. כלומר:

$$\Pr\left[\texttt{Verify}_{\mathcal{PCP}}(p, \pi) = \text{accept}\right] > 2^{-1000} \Rightarrow \exists \text{ valid } w$$

הערה w התחלף ב־π, מכיוון שעכשיו אנחנו שולחים מחרוזת הוכחה ולא עד. זה ההבדל בין הדטרמיניסטיות להסתברותיות. המשמעות של $\mathcal{PCP}$ היא Probabilistically-Checkable Proof.
מבחינה מעשית, ההסתברותיות לא פוגעת בנכונות, כי הסיכויים לטעות נמוכים מאוד. מצד שני, מוודא הסתברותי יכול להיות הרבה יותר יעיל ממוודא של $\mathcal{NP}$.

8.3.1 $\mathcal{PCP}$ אקספוננציאלי

ניתן לבדוק באופן הסתברותי מעגל בולאני באמצעות בדיקת מחרוזת באורך אקספוננציאלי:

- ניתן לראות את המעגל כאוסף של שערים.
- ניתן לבטא כל שער כאילוץ לינארי.
- המוכיח לוקחח השמה מספקת לשער, ומקודד אותה באמצעות קוד Hadamard.

המוודא:

- מבחן קרבה: בדיקת מחרוזת ההוכחה לקרבה לקוד Hadamard (ניתן לעשות זאת באמצעות 3 שאילתות).
- מבחן עקביות: צריך לבחור תת־קבוצה של האילוצים הלינארים, ולקבל מהם אילוץ לינארי חדש. ניתן לבדוק שהאילוץ החדש מתקיים כמצופה.

8.3.2 $\mathcal{PCP}$ פולינומי

בהוכחות $\mathcal{PCP}$ באורוך פולינומי, זמן הווידוא יהיה יותר גבוה מזה של הווידוא ב־$\mathcal{PCP}$ אקספוננציאלי, אבל יותר נמוך מזה של הוכחות $\mathcal{NP}$.
למעשה, הבעיה אינה טריוויאלית. עוד מתחילת שנות ה־80 יש עבודות בנושא, ולמעשה רוב הפתרונות הקיימים הם פולינומיאליים, אבל בפרמטרים לא הגיוניים שאינם פיזיביליים לחישוב מעשי.
האינטואיציה נשארת דומה למקרה האקספוננציאלי: מוסיפים קוד לתיקון שגיאות, ומוודאים תת־קבוצה אקראית של ההודעה כדי לוודא את הנכונות. הבנייה עצמה מורכבת הרבה יותר, ולא נכנס כאן להסברים.

אלגוריתם 1.8 וידוא הוכחת $\mathcal{CS}$

1. המוכיח ייצר הוכחת $\mathcal{PCP}$ באורך פולינומי.
2. המוודא יקרא רק מעט מאוד אינדקסים רנדומיים.

   אז למה לשלוח את כל הוכחת ה־$\mathcal{PCP}$? אם המוכיח היה יודע מה המוודא הולך לקרוא, הוא היה יכול לעבוד עליו בקלות (יחסית).
3. הרעיון: נשתמש ברעיון של Fiat-Shamir.

   (א) המוכיח כותב מחרוזת הוכחה π ל־$\mathcal{PCP}$.

   (ב) המוכיח מחשב Merkle hash tree של π. נניח כי r הוא השורש.

   (ג) המוכיח משתמש ב־r כמקדם האקראי לבחירת האינדקסים.

   (ד) המוכיח שולח את האינקסים האלה למוודא, יחד עם הנתיבים לווידוא של עץ ה־Merkle.

   (ה) המוודא בודק את הנתיבים, ואז קורא למוודא $\mathcal{PCP}$.

8.3.3 הוכחות $\mathcal{CS}$ (Computationally-Sound)

נקל על עצמנו קצת יותר: אנחנו דורשים רק מוכיחים יעילים (בזמן פולינומי).
מוכיח יעיל יכול לרמות את המוודא, אבל אפשר לגרום לזה להיות יותר קשה באמצעות הגדלת פרמטרי הביטחון.

הגדרה לכל מוכיח P בזמן פולינומי:

$$\Pr\left[\texttt{Verify}_{\mathcal{CS}}\left(P,\pi\right)=1\right]>2^{-1000}\Rightarrow\exists\text{ valid }w$$

אנלוגיה אפשרית: הצפנה בטוחה לפי תורת המידע מול הצפנה בטוחה חישובית.

מבחינה מעשית, הוכחת $\mathcal{CS}$ היא לא פחות טובה מכל הוכחה אחרת, בגלל ההנחות החישוביות שהנחנו, כמו למשל $\mathcal{P}\neq\mathcal{NP}$, ושקיימת פונקציית Hash החסינה להתנגשויות (Collision Resistent). ההנחות החישוביות להוכחות $\mathcal{CS}$ דומות להנחות החישוביות שמניחים בכל עולם הקריפטוגרפיה המעשית.
אלגוריתם 1.8 מתאר את אופן הוכחת $\mathcal{CS}$. מה חסר לנו כאן? למעשה, שלחנו את עץ ה־Merkle, אבל לא הוכחנו שהוא קשור ל־π (מחרוזת ההוכחה של $\mathcal{PCP}$).
איך נעשה את זה? יש לנו עץ חישובים של פונקציית Hash. הערך אותו אנחנו בודקים (למשל האינדקס השלישי ב־π) למעשה מכתיב לנו נתיב בתוך העץ. לכל צומת בעץ, חסר לנו ערך אחד לחישוב ה־Hash (כדי שנוכל לוודא שאכן צדקנו). למעשה, המוכיח צריך לשלוח רק $2\log n$, כאשר n הוא מספר הצמתים בעץ. אם פונקציית ה־Hash היא חסינה להתנגשויות, אז התהליך הזה בטוח (כי אחרת המוכיח לא ידע לתת לנו את הערכים הנכונים).
הוכחה זו נקראת *Fiat-Shamir Compression*.

8.3.4 בעיות בבנייה

כל הבניות מניחות שאנחנו עובדים במעגל חשמלי. בפועל, רוב התוכנות שלנו נכתבו בשפות עיליות (כמו C או Java), והתרגום שלהן למעגלים מאוד לא טריוויאלי.

בשיעור הבא נראה איך אפשר לעשות את זה גם לתוכנות יותר מעשיות.

פרק 9

Integrity on Untrusted Platforms (המשך)[1]

9.1 בניית מערכת שמכילה הוכחה למידע (המשך)

9.1.1 בעיית $\mathcal{CSP}$

נכיל מאפיין נוסף שאותו נרצה לספק: הוכחת ידע (*Proof of knowledge*): כל פעם שהמוכיח שולח הוכחה שמספקת את המוודא, יהיה לנו מחלץ ידע (*Knowledge Extractor*) שיוציא עד w.
בצורה יותר פורמלית:

$$\Pr\left[\texttt{Verify}_{\mathcal{CSP}}\left(P_{\mathcal{CSP}},\pi\right)=1\wedge\texttt{KnowledgeExtractor}\left(P_{\mathcal{CSP}}\right)=\text{valid }w\right]\approx 1$$

ברור שזה יותר חזק מבעיית $\mathcal{CS}$. למעשה, בפועל לא נריץ את מחלץ הידע. הוא קיים רק בשביל רצודקציות לבטיחות.
במובן הכללי, ההגדרה אומרת כך: לכל מוכיח, קיים מחלץ, כך שאם המוודא השתכנע מהמוכיח, אז המחלץ מוציא עד שבסיכוי גבוה תקין. זה אומר שאנחנו לא יודעים אם המחלץ יכול לדבר עם המוכיח. לרוב, המחלץ אכן יכול לדבר עם המוכיח, וכך הוא יכול להסיק דברים.

אז איך נבצע את החישוב הכול? Alice מקבלת את x ויש לה את F, אז היא תחשב את $y=F(x)$ ובנוסף גם הוכחה π_y שמוכיחה ש־$y=F(x)$. Bob מוודא את y לפי π_y, ואז מחשב את $z=G(y)$, ומייצר הוכחה π_z לכך ש־$z=G(y)$ וגם שקיים כך π_y ש־$\texttt{Verify}\left("y=F(x)",\pi_y\right)=1$. Carol מקבלת את z ואת π_z ואז אמורה להאמין לכך ש־$z=G(F(x))$.
יש כאן בעיה: איך Bob יכול להוכיח שקיים π_y כך ש־$\texttt{Verify}\left("y=F(x)",\pi_y\right)$? הרי אפשר לשקר למוודא הטענות. הרי קיימות הוכחות שלא ניתן לחשב בייעילות, אבל הן

[1] סיכום חלקי לשיעור שהתקיים בתאריך 18.12.2012.

עדיין קיימות ועדיין מוכיחות. גם קיימות טענות שלא מוכיחות, אבל משכנעות את המוודא. לכן, Carol לא יכולה להאמין ל־Bob ש־y נכון, אבל היא יכולה לוודא ש־$z = G(y)$.

למעשה, Bob גם לא חייב לשלוח את y ל־Carol. הוא רק אומר לה ש־y קיים, אבל לא מספר לה מהו y. הוא רק אומר לה להאמין לו ש־y הוא בסדר. לכן, Bob רק הוכיח ש־z הוא בתמונה של G. לרוב, זו אינה טענה מעניינת, ונרצה לקבל יותר מזה.

אפשר לפתור את הבעיה בצורה פשוטה: Bob ישלח את π_y ל־Carol. אבל π_y הוא יותר מדי גדול, ולא היינו רוצים להעביר אותו ל־Carol. גם אם משרשרים הרבה הוכחות כאלה, אז אורך ההוכחה היה הופך לממש גדול.

הפתרון כאן יהיה הוכחת ידע: לא הוכחה שקיים העד, אלא מחלץ ידע שיכול לחלץ את π_y (במקרה שלנו). נכתוב את זה בצורה פורמלית:

$$\forall \tilde{P}_{\text{Bob}}.\exists E_{\text{Bob}}.\Pr\left[\begin{array}{c}\tilde{P}_{\text{Bob}} \xrightarrow{\tilde{\pi}} \texttt{Verify}_{\text{Carol}} \to 1 \\ \Downarrow \\ E_{\text{Bob}} \xrightarrow{w} \texttt{Verify}_{\mathcal{NP}}\text{Carol}\end{array}\right] > 1-\varepsilon$$

במקרה שלנו, $w = (y, \pi_y)$.

היינו רוצים להשתמש בהוכחות מבוססות $\mathcal{PCP}$ גם להוכחות ידע. הדרך היחידה הידועה להוכחת ידע שאינה איטראקטיביתצ היא של Micali, שמכילה עצי Merkle שפונקציית ה־Hash הבסיסית כוללת קריאות אקראיות לאורקל.

אבל מה הבעיה כאן? משפט ה־$\mathcal{PCP}$ אינו מתקיים עם השינוי הזה, כי אי אפשר לדבר עם אורקל חיצוני אקראי.

מסתבר שיש טריק שאפשר לעשות ששובר את הבעיה. לא נשתמש באורקל אקראי, אבל נשתמש במשהו אחר במקום: נשים את האקראיות רק בצד של המוכיח. הבעיה למעשה הייתה רק בצד של המוודא, כי אותו היינו צריכים להכניס למשפט ה־$\mathcal{PCP}$ בשביל הצעד הבא. למעשה, Alice משתמשת באקראיות לבניית עץ ה־Merkle שלה.

אבל איך Bob יכול לדעת ש־Alice השתמשה באקראיות טובה מספיק? האורקל יחתום על האקראיות שלו באמצעות אלגוריתם חתימה א־סיטמרי. נקרא לגורם הזה *SIR – Signd Input and Randomness*. איך זה יעבוד? הוא יקבל מחרוזת קלט x באורך s. הוא יחזיר מחרוזת אקראית r באורך s, וחתימה $\sigma = \texttt{Sign}_{SK}(x, r)$.

עכשיו, המוכיח של Alice יבקש אקראיות מה־SIR, ויספק ל־Bob גם את החתימה על כך שהאקראיות סופקה על־ידי השירות.

9.2 כנס

שבוע הבא יתקיים כנס באבטחת מידע באוניברסיטה.

אתר הכנס נמצא בכתובת:

http://cpiis.cs.tau.ac.il/cryptosec2012

פרק 10

Fully Homomorphic Encryption[1]

10.1 Motivation

Why would we need fully homomorphic encryption?

For example, where is my mail? It is in my smartphone, right in my pocket. Actually, Google has it. My e-mails are in the cloud server of Google.

But what if I don't want Google reading my mail? How can I keep my data private from the cloud?

Today, the cloud company tell us that they are not going to access our data, etc. However, it is not a real protection. We don't really want to trust them. In the case of Google, you actually signing them (in the end-user agreement) that they can read and access your mail. What do you do now? How would you protect your data?

The obvious approach is to encrypt the e-mails. However, processing the e-mails become really hard. For example, how do you expect the server to search in your mail messages?

The main idea of fully homomorphic encryption is that the server will compute blindfolded. The server does not necessarily need to know really information about the content of the messages. It may process the encrypted data, and produce an encrypted respose. More generally: we want a scheme that will support full functionallity.

10.2 Our Model

We will have two keys: a secret key and a public key. The evaluator will have a new kind of key: an *evaluation key* (evk). The evaluation key is a public key.

[1] סיכום לשיעור שהתקיים כחלק מכנס לאבטחת מידע באוניברסיטת תל־אביב, מתוך הרצאה של Zvike Brakerski מ־Stanford בתאריך 25.12.2012.

Now I can encrypt my input using my private key. The server will calculate the value of a function f with the encrypted input. The output of this evaluation will be the encryption of the output of applying f on the plain input.

Can it be done? Actually, it can. We think it is possible due to a research of late 2008.

10.3 Gentry's Breakthrough

The basic scheme is polynomial-based. The ciphertext could be represented as a polynomial, which can be represented as a high dimension vector.

You can add, multiply, etc. these vectors in a meaningfull way, such that the product of the vectors is the encryption of the product of the inputs.

As we saw in the previous lectures, this scheme is based on the fact that the ciphertext is in a limited range. Every operation on the ciphertext adds noise to the ciphertext, and therefore this scheme is limited in the number of allowed operation.

We could solve it using *bootstrapping*. It uses homomorphisms to reduce the noise of the ciphertext.

10.4 The FHE Gold Rush

- Make it simpler.
- Make it more secure.
- Make it practical.

10.5 Second Generation FHE Scheme

In this model, the polynomial rings or ideals are not needed.

In this scheme, the ciphertext is also a vector. In order to add ciphertexts, we add vectors. In order to multiply ciphertexts, we multiply vectors.

How do we multiply vectors? We use the outer products. If $\vec{x} = (x_1, x_2, x_3)$ and $\vec{y} = (y_1, y_2, y_3)$, then:

$$\vec{x} \otimes \vec{y} = (x_1y_1, x_1y_2, x_1y_3, x_2y_1, x_2y_2, x_2y_3, x_3y_1, x_3y_2, x_3y_3)$$

The problem here is that the dimension of the vector grows (really fast) in each multiplication operation. We have to find a solution that shrinks back the vector size. In addition, the noise in the multiplied vector will grow. We can handle it.

10.5.1 The Scheme

The secret key is a vector $\vec{s} = \{0,1\}^N$ ($N \approx 128$). The ciphertext is a vector $\vec{c} \in (-1,1]^N$.

What message is encoded in $\vec{c}$? For example, if $\vec{s} = (1,0,0,1)$ and $\vec{c} = (-0.2, 0.7, 0.6, 0.3)$. Define: $I = \{1 \leq i \leq N \mid s_i = 1\}$. Look at:

$$\sum_{i \in I} c_i$$

- If it is closer to even, $m = 0$.
- If it is closer to odd, $m = 1$.

In our case, $-0.2 + 0.3 = 0.1$, and thus the message was 0.

10.5.2 Correctness

Why is the multiplication correct? We have to get a ciphertext, that when decrypted should get the output of both inputs.

It is easy to see that the relevant elements in the vectors product aregood in all the cases (remembering that if an even number is multiplied, it gives us an even number).

We may look at it as an equation of all the elements. We want to reduce the number of participating elements (and thus allow us reducing the vector size. For this, we can use re-linearization.

10.5.3 Re-Linearization

Idea: What if we hat $\vec{g} = \texttt{Enc}(s_1, s_2)$ in evk?

$$\underbrace{s_1 g_1 + s_2 g_2 + \cdots}_{\text{degree 1}} \approx \underbrace{s_1 s_2}_{\text{degree 2}}$$

Then:

$$(\cdots) c_1 c_2 + \overbrace{(s_1 g_1 + s_2 g_2 + \cdots)}^{} c_1 c_2 + (\cdots) c_1 c_3 + \cdots$$

Back to degree 1!

10.6 Conclusion

How close are we?

Current state of the art in optimizations: implementation of AES-128 computation. The running time is around 8 days on machine with 256GB RAM (92 blocks in parallel – amortized 2 hours per block).

Why are we not getting FHE as good as regular encryption (such as RSA)?

- Need to convert arbitrary functions into sequences of additions and multiplications.
- Reducing nose and degree requires auxiliary information. Currently this information is huge.
- Information content per ciphertext is low.
- Our basic building block, "lattice based cryptography", is still slow in practice.

10.7 Further Reading

- http://tiny.cc/fheblog1
- http://tiny.cc/fheblog2

פרק 11

הבטחת תכונות בטיחות של חישוב[1]

11.1 תכונות של חישוב

בטבלה 11.1 אפשר לראות הרבה תכונות לא ברורות של חישוב.

[1] טבלה מהשיעור שהתקיים בתאריך 01.01.2013.

Primitive	Attacks		Guarantees		Functionallity		Communication	Assumptions
	Leakage	Tampering	Correctness	Secrecy	Function Class	Output Form		
FHE	Any	None	No	Yes	Circuits	Encrypted	Minimal	Computational
Arguments (CS/ proofs/ PCD/ SNARG)	Any	Any	Yes	No	RAM, distributed	Plaintext	Minimal	Exotic computational/oracle
MPC	Any	Any	Yes	Yes	Ant	Plaintext	Heavy interaction	Mild Computational
Garbled Circuits	Any	Any	No	Yes	Circuits	Plaintext	Preprocessing + Minimal	Mild Computational
Leakage resilience	Varies	None	Yes	Tes	Varies	Plaintext	Minimal	Varies
Tamper resilience	Varies	Varies	Varies	Varies	Varies	Plaintext	Minimal	Varies
Obfuscation	Any	Any	Yes	Yes	Yes	Plaintext	Minimal	0=1
TPM								Secure Hardware

טבלה 11.1: טבלה עם הרבה מידע לא ברור

פרק 12

Trusted Computing Architectures & Leakage Resilience[1]

TPM 12.1

השם המדוייק הוא *Trusted Computing Architecture*, שהוגדר על ידי Trusted Computing Group. אנחנו מדברים על רכיבי Trusted Platform Module.

הרעיון הבסיסי הוא להוסיף רכיב חדש לכל מחשב, בעלות נמוכה, שיאפשר מגוון רחב של פעולות והגנות. למשל:

- Boot מאובטח:

  לוודא שהגרסה הנכונה של הפלטפורמה עלתה, או איזה מערכת הפעלה עלתה. זה טוב למשל נגד הפעלה של מערכת ההפעלה במכונה וירטואלית שמוציאה סודות ממערכת ההפעלה.

- Attestation (עדות):

  לספק הוכחה למישהו אחר שאני מריץ גרסה מסויימת של הפלטפורמה. למשל, בשביל DRM, או בשביל מנהלי IT של רשת ארגונית.

- הגנה על אחסון.

  מערכת ההפעלה מספקת הגנה על האחסון באמצעות מערכת הרשאות, אבל רק כשהיא רצה. אם מחברים דיסק של מחשב אחד למחשב אחר, מערכת ההפעלה השונה לא מחוייבת לספק את ההגנות של מערכת ההפעלה המקורית. ה־TPM אוכף את כללי הגישה לנתונים בכל מצב.

ה־TPM הוא רכיב חדש המחשב, בעלות נמוכה (בערך $0.3 לרכיב), והוא רץ בתדר של 33MHz. הוא מחוייט להרבה רכיבים אסטרטגיים במחשב, כמו המעבד, הזיכרון, הכוננים, הרשת ועוד. התוכנות יכולות לתקשר עם הרכיב ולהיעזר בו בביצוע פעולות. מעבר לרכיב

[1] סיכום לשיעור שהתקיים בתאריך 08.01.2013.

עצמו, גם התוכנות דורשות שינויים כדי להסתגל אליו (למשל, מערכת ההפעלה או ה־BIOS), כפי שנראה בהמשך.

12.1.1 מבנה ה־TPM

ה־TPM יושב על ה־Bus של המחשב. בתוכו יש זיכרון קטן שאינו נדיף, מנוע קריפטוגרפי שניתן להשתמש בו עם אלגוריתמים שונים, כמה אוגרים (Registers) ועוד כמה דברים פחות מעניינים.

מה יש בזיכרון?

- Endorsement Key (או בקיצור EK): מפתח RSA שמוכיח מי היצרן.
- Storage Root Key (או בקיצור SRK): משמש לאחסון מוצפן. ניתן לשנות את המפתח על ידי המשתמש.
- סיסמת הבעלים.

כל המידע השמור בזיכרון לעולם לא יוצא החוצה מהרכיב.

האוגרים של ה־TPM נקראים *Platform Configuration Registers* (או בקיצור PCR). יש לפחות 16 אוגרים על הרכיב, וכל אחד מהם מכיל Digest Hash (בדרך כלל SHA-1). חשוב שפונקציית ה־Hash תהיה חסינה להתנגשויות.

הגישה ל־PCR פשוטה, באמצעות שתי פונקציות ב־API של TPM:

- $\texttt{TPM\_Extend}(n, D)$

  מבצע $\texttt{PCR}[n] \leftarrow \texttt{SHA}-1(\texttt{PCR}[n] \,|D)$.
- $\texttt{TPM\_PcrRead}(n)$

  מחזיר את הערך של $\texttt{PCR}[n]$.

למה זה טוב? כפי שניתן לראות באיור 12.1, קיבלנו את התכונה המעניינת שניתן בעזרת השורש S_3 לקבע את כל התוכן של העץ x_1, x_2, x_3. מתכונת החסינות להתנגשויות של פונקציית ה־Hash, נקבל כי כל יריב לא יצליח להגיע ל־S_3. ככה אפשר לשכנע גורמים שונים בטענות שונות (למשל בגרסת מערכת ההפעלה).

פונקצייה נוספת ל־PCR־ים היא לקבוע ערך חדש לאוגר PCR כלשהו.

12.1.2 סדר עליית המחשב (Boot)

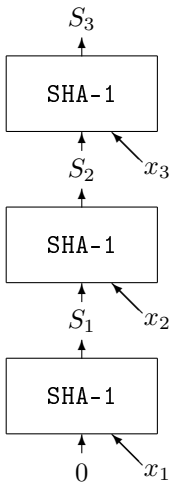
סדר עליית המחשב (ה־Boot) הוא דוגמה נוספת לאופן התקשורת של המערכת עם ה־TPM כדי להוכיח דברים. אלגוריתם 1.12 מתאר איך משתמשים ב־TPM (או ליתר דיוק, ב־PCR־ים) בעליית המחשב.

למעשה, יצרנו שרשרת של אמון (כפי שניתן לראות באיור 12.2). בסוף תהליך ה־Boot, ה־PCR מכיל שרשרת של Hash־ים של הקוד של כל אחד מהרכיבים שהשתתפו בתהליך. אם ה־Hash חסין להתנגשויות, אז לא ניתן לזייף הרצה של קוד זדוני.

איך נשתמש בערכים ב־PCR לאחר ה־Boot? אלגוריתם 2.12 מתאר את אופן השימוש ב־TPM וב־PCR־ים שבו לצורך הצפנת האחסון.

שימושים נוספים ל־TPM:

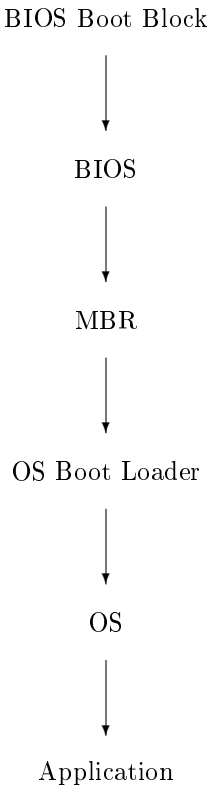
- נעילת תוכנות על מחשב (כמו מערכת ההפעלה, ה־MBR ועוד).



איור 12.1: מבנה של עץ האמון ב־PCR.

אלגוריתם 1.12 ה־Boot מול ה־TPM.

1. ה־BIOS Boot Block יקרא ל־`TPM_Startup` כדי לאפס את ה־PCR־ים.
2. ה־BIOS Boot Block ישלח את הקוד של עצמו לאחד ה־PCR־ים.
3. ה־BIOS Boot Block ישלח את הקוד של ה־BIOS לאחד ה־PCR־ים.
4. ה־BIOS Boot Block יפעיל את ה־BIOS עצמו.
5. ה־BIOS ישלח את הקוד של ה־MBR (Master Boot Record) לאחד ה־PCR־ים.
6. ה־BIOS יקרא לקוד שב־MBR.
7. ה־MBR ישלח את הקוד של ה־OS Boot Loader לאחד ה־PCR־ים.
8. ה־MBR יקרא ל־OS Boot Loader.
9. ה־OS Boot Loader מדווח על מערכת ההפעלה.
10. ה־OS Boot Loader מריץ את מערכת ההפעלה.
11. מערכת ההפעלה מדווחת על האפליקציה, וכן הלאה.



איור 12.2: שרשרת האמון שנוצרה בתהליך ה־Boot עם ה־TPM.

אלגוריתם 2.12 שימוש ב־TPM לצורך הצפנת האחסון

1. נאתחל את ה־TPM: נקרא לפונקציה `TPM_TakeOwnership` פעם אחת, בעת קניית המחשב.

   הפונקציה הזו מאתחלת מפתחות (כמו EK) שנחוצים לשימוש ב־TPM.

2. נארוז מידע בצורה מוצפנת.

   לצורך כך, נשתמש בשתי פונקציות:

   - `TPM_Seal` שמקבלת:

     `keyhandle` מפתח ההצפנה.

     `KeyAuth` סיסמה לשימוש ב־`keyhandle`.

     `PcrValues` כדי לאמת ולהשתמש בנתונים.

     `datablob` לכל היותר 256 בתים (2048 סיביות), שנועדו להצפנת המפתח בהצפנה סימטרית (כמו AES).

     הפונקציה מחזירה Encrypted Blob (כאשר Blob הוא קיצור של Binary Large Object, כשבפועל הוא Ciphertext).

3. מכיוון שה־TPM הוא רכיב קטן, איטי וטיפש יחסית, נצפין את המידע עם מפתח אקראי להצפנה סימטרית, ול־TPM ניתן שיצפין את המפתח הסימטרי.

4. הדרך היחידה שתהיה לפענח את הקבצים היא באמצעות פענוח מפתח ההצפנה הסימטרית, בעזרת ה־TPM, לפי ערכי ה־PCR־ים, באמצעות `TPM_Unseal`. אם ערכי ה־PCR־ים יהיו שגויים, `TPM_Unseal` ייכשל.

- חתימה על מפתחות SSL פרטיים.

  המטרה: רק Apache (למשל) שלא עבר שינויים יכול לגשת למפתח הפרטי.

  הבעיה: אי אפשר לעדכן את הגרסה של Apache. באופן דומה, גם אי אפשר לשנות את הקונפיגורציה שלו. זוהי בעייה כללית עם מערכות תוכנה, וזה מנע מעבר לשימוש ב־TPM למשך זמן רב.

דוגמה נוספת לשימוש ב־TPM היא אפליקציית ענן. נניח שאנחנו בוטחים בספק הענן, אבל אנחנו חוששים שיבוא מישהו ויגנוב את הקופסה מהספק. אז אפשר להצפין את נתוני ה־VM באופן כזה שה־VM יכול להתפענח בקלות רק על שרת תקין, והספק יוכל לגשת בקלות למידע.

12.1.3 תקיפות

להלן תקיפות אפשריות להגנות שמושגות על ידי רכיבי ה־TPM:

- לשבש את ה־BIOS, או PCR Reset.

  דרך אחרת לגרימה ל־PCR Reset היא באמצעות נפילת מתח, או קריאה ל־`TPM_Init`.

- Software Verifiction

  כדי לחתום על תוכנה, אנחנו צריכים לדעת שאנחנו חותמים על הדבר הנכון. למשל, אם מתגלה באג במערכת ההפעלה, יוצא Patch, ואז עושים Unseal ו־Seal מחדש. גם אם התהליך הזה מאובטח, תוקף שיצליח למצוא את ה־Blob של מערכת ההפעלה הישנה (כמומר, לפני העדכון), יוכל להריץ אותה, ועדיין לקבל את מפתח הפענוח של הדיסק, ויוכל להשתמש בבאג (או בחולשה) של מערכת ההפעלה הישנה.

  זה נקרא התקפת Rollback.

- Side Channel (לא על ה־TPM), או התקפות Pribing שונות.

כדי להתגונן מחלק מהתקיפות, יצרניות המעבדים המובילות (Intel ו־AMD) מאפשרות Dynamic Root of Trust Measurement (או בקיצור DRTM), שמאפס את ה־TPM: עושים Reset ל־CPU ול־PCR־ים 0 עד 17, טוענים Secure Loader ל־Instruction Cache, מרחיבים את `PCR`[17] עם הקוד של ה־Secure Loader ואז קוראים לו.
זה מונע שיבושים ב־BIOS Boot Loader, ומעכשיו הוא לא ה־Root of Trust. זה גם מונע התקפות `TPM_Init`, שקובע את `PCR`[17] להיות -1. זאת פקודת מעבד חדשה (`senter` ב־Intel ו־`skinit` ב־AMD).

12.1.4 Attestation

איך אפשר לבטוח בחישוב שבוצע במחשב שלא סומכים עליו? ראינו שיטות אלגוריתמיות. נראה איך עושים את זה בעזרת TPM.
מטרה: להוכיח לגורם מרוחק איזה תוכנה רצה על המחשב.
שימושים:

1. בנקים מאפשרים העברת כספים רק אם המשתמש מוכיח לבנק שהוא מריץ מערכת הפעלה בטוחה.

2. חיבור מחשבים לרשת ארגונית רק אם הם "בסדר".

3. משחקים: אפשר להתחבר לשרת משחקים ברשת רק אם הלקוח אינו שונה.

4. DRM.

איך זה עובד? ניזכר כי ל־TPM יש מפתח EK פרטי. המפתח הפומבי של EK חתום על ידי מפתח של יצרן ה־TPM.

שלב 1: ניצור Attestation Identity Key (AIK). בשלב זה לא חשוב איך יוצרים את המפתח, ולאיזה משתמש (זהות) הוא משוייך. המפתח הפרטי ידוע רק ל־TPM, ואי אפשר לחלץ אותו משם. המפתח הציבורי מפורסם ונחתם על ידי ה־EK הפרטי.

שלב 2: נחתום על ה־PCR־ים באמצעות `TPM_Quote`. הפרמטרים של הפונקציה הם:

`keyhandle` באיזה AIK להשתמש.

`KeyAuth` סיסמה לשימוש ב־`keyhandle`.

`PcrList`

`Challenge` 20 בתים שהגיעו מהשרת המרוחק.

`Userdata` נתונים נוספים לשימוש בחתימה.

הפונקציה מחזירה את המידע החתום והחתימה עליו.

בפועל, השרת שולח לאפליקציה Challenge של 20 בתים. הלקוח מייצר זוג מפתחות, פרטי וציבורי, וחותם עליו עם `TPM_Quote`. בשלב הבא יתבצע Key Exchange (כמו SSL) עם המפתח והחתימה. השרת יבדוק את מנפיק החתימה, ושערכי ה־PCR־ים בחתימה משכנעים שהאפליקציה אצל הלקוח תקינה. ה־Attestation חייב לכלול החלפת מפתחות ובידוד של האפליקציה משאר המערכת. בשלב האחרון, כל התקשורת בין השרת ללקוח תהיה מוצפנת עם המפתח שנוצר.

12.1.4.1 שימוש ב־Nexus OS – Attestations

הבעיה: כשעושים Hash לקוד של אפליקציה, יש הרבה מאוד קונפיגורציות אפשריות.

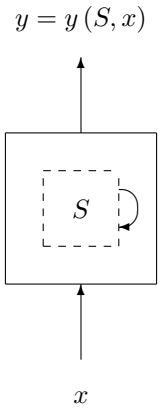
הגישה: נעיד על תכונות של האפליקציה, במקום על הקוד שלה. למשל: האפליקציה אינה כותבת לדיסק.

זה נתמך במערכת ההפעלה Nexus. משפט עדות כללי: "ה־TPM אומר שביצע Boot ל־Nexus, Nexus הריץ Checker עם Hash X, והוא אומר שלאפליקציה A יש את התכונה P".

12.1.4.2 התחייבות ה־TPM

אם נחשף מפתח פרטי של ה־Endorsement של TPM, אז כל תשתית העדות נהרסת, כי אפשר לעשות אמולציה ל־TPM עם ה־EK הפרטי שלו, ואפשר להעיד על כל דבר בלי באמת לבדוק את זה.

לכן, Certificate Revocation חשובה מאוד לעדות TCG.



איור 12.3: מעגל לפי ההנחה

12.2 Leakage Resilience

ההנחה: יש לנו מעגל (עם מצב S), שמקבל קלט x ומחזיר פלט $y = y(S, x)$ (ראה איור 12.3). אנחנו רוצים לבנות מעגל שיש לו מצב S', עם אותה פונקציונליות כמו המעגל המקורי, שלא מדליף מידע.

ההנחות:

1. המעגל רץ במשך הרבה מחזורי שעון.
2. בכל מחזור שעון:
   (א) התוקף בוחר את הקלט של המעגל.
   (ב) התוקף בוחר תקיפה למעגל: דליפה ו/או שיבוש מסוג כלשהו.
   (ג) התוקף רואה את הפלט של המעגל, ואת הדליפה שלו.
   (ד) מצב הזיכרון משתנה.

   יש לנו הנחה של אטומיות בכל מחזור שעון.

תכונת הבטיחות: כל מעגל בולאני מקבל קלט x עם זיכרון m ומחזיר פלט y. תמיד קיים אלגוריתם או סימולטור לרכיב שיכול ללמד מתקיף, בדיוק כמו מתתקיף למעגל עצמו.

פרק 13

[1]Leakage Resilience

יש לנו מעגל עם מצב s, שמקבל קלט x ומחזיר פלט $y = y(s,x)$[2]. המעגל רץ בהרבה איטרציות. יש גם מעגל לוגי שמדמה אותו.

יש יריב שבכל איטרציה יכול לראות את הקלט, את הפלט, ויכול לבחור דליפה ולראות את מה שהיא נותנת לו.

ההגדרת הבטיחות: קיים אלגוריתם שיכול ללמוד מהסימולטור בדיוק את מה שהיריב יכול ללמוד מהמעגל. באופן פורמלי: T שומר על הפרטיות אם לכל מעגל C קיים סימולטור Sim כך שלכל יריב Adv ולכל מצב התחלתי s_0, $Sim^{Adv,C[s_0]}$ דומה ל־view של היריב כשהוא תוקף את $C'[s'_0]$. $C'[s'_0]$ אמורה להתנהג כמו "קופסה שחורה וירטואלית" של $C[s_0]$, גם כאשר מדברים עלתקיפות של ערוצי צד. C' הוא $Obfuscation$ (ערפול) של C. T מקבל את C ומחזיר את C'.

למעשה, האובפוסקציה היא לא מדברת על מקרים מציאותיים. למשל, אי אפשר לבחון (to probe) את כל החוטים (wires) במעגל. לכן, צריך להפשיט קצת את המודל.

סכימות פשוטות:

1. דליפת Sum-of-wires (*Dual-rail logic*).
2. דליפת Sum-of-wires-transitions (*Dual-rail precharge logic*).

[1] סיכום חלקי לשיעור שהתקיים בתאריך 15.01.2013.

[2] ראה איור 12.3.