digital whisper Technologic Papers

RSA Side Channel Attack

מאת אבי פדר ודניאל יוחנן

הקדמה

במהלך ההיסטוריה, לא היו פעמים רבות שבהם יצאה טכנולוגיה חדשה לשוק ולאחר שהטכנולוגיה יצאה לאור, פרסמו ארגוני ביון ברחבי העולם שהטכנולוגיה ידועה להם ובשימוש מעשי במשך עשרות שנים. אחת הפעמים הבודדות שזה קרה היתה בשנת 1997. כאשר פרסם מטה התקשורת של המודיעין הבריטי GCHQ, מסמך בו נטען כי רעיון המפתח הפומבי (נחזור למושג הזה עוד) היה ידוע להם כעשור לפני שהתפרסם.

הם גילו לטענתם גם את RSA וגם את דיפי־הלמן, אלגוריתמים שלציבור יצאו לאור בשנת 1976 ואחרי, אך שמרו את הדבר בסוד. גם ה־NSA טען שגילה את RSA הרבה לפני שנודע לציבור, אולם התגלית סווגה כסוד לאומי.

במאמר זה אנחנו נממש מתקפת ערוץ צד (מסוג Timing attack, כמו בפעם הקודמת) לצד שימוש ב-Eembedded chips.

נקדים ונאמר שהמטרה שלנו במאמר אינה להרחיב על RSA, מי שרוצה ללמוד על כך יותר יכול לקרוא על כך בקישורים שנצרף תוך כדי.

המטרה שלנו היא להנגיש את המתקפה ע"י מימוש פשוט שלה ובעצם לתת לאנשים תשובה בסיסית לשאלה למה לא משתמשים ב-RSA פשוט כמו שהוא.

לאורך המאמר נבנה ביחד את המערכת ואת הקוד. המאמר מניח שיש לכם הכרות עם RSA, עם תכנות בסיסי ועם ארדואינו. אם אין לכם את הידע הזה, ממליץ לכם לקרוא בקישורים שאנחנו מצרפים בין לבין על RSA או במאמר הקודם על ארדואינו ו-Side-Channel Attacks.

תזכורת - Timing Side Channel Attack

כפי שכולנו מבינים, כדי שתוכנה תוכל לבצע פעולות מסויימות, היא צריכה זמן מעבד שבו היא תבצע חישובים. את זמן העבודה של המעבד ניתן למדוד. וכך, במידה ואנו מודעים לפרמטרים הרלוונטים, כמו האלגוריתמים או ארכיטקטורת המעבד שבה המערכת משתמשת, ניתן להסיק מסקנות על המערכת או על הפעולות שנעשו בה. למתקפות מהסוג הזאת קוראים Timing attack.

קיימות המון מתקפות Timing, גם על אלגוריתמים מפורסמים של הצפנה כמו RSA שאותה נממש היום.

אז מה בעצם אנחנו נעשה?

במהלך המאמר אנחנו נבנה מערכת חתימה על הודעות שמשתמשת ב-RSA textbook. לאחר מכן על המערכת הזאת אנחנו נממש Timing attack.

קישור לכל הקוד שנכתוב במהלך המאמר ניתן למצוא כאן.

מהו RSA?

RSA היא מערכת הצפנת מפתח ציבורי דטרמיניסטית הראשונה שהומצאה והיא עדיין בשימוש נרחב כיום. ב־RSA, כבכל מערכת מפתח ציבורי, מפתח ההצפנה אינו סודי והוא שונה ממפתח הפענוח שנשמר בסוד, לכן היא נקראת אסימטרית. האסימטריה ב־RSA נובעת מהקושי המעשי (והכמעט בלתי אפשרי) שבפירוק לגורמים של מספר פריק, שהוא כפולה של שני ראשוניים גדולים (מדובר על בעיה פתוחה בתורת המספרים).

כלומר, קל יחסית ליצור שני מספרים ראשוניים p, q ולחשב את המכפלה שלהם N = pq. אך בהינתן N קשה למצוא את גורמיו p ו-q. ההצפנה משתמשת בערך ציבורי, או במפתח, המופץ וידוע לכל מי שרוצה לשלוח הודעה. הפענוח מתבצע באמצעות מפתח פרטי הנשמר בסוד על ידי הנמען המיועד ולא ניתן להסיק אותו מהמפתח הציבורי. הצפנה כזאת עובדת מבלי לדרוש משני הצדדים המעורבים לשמור על סוד מוסכם - המפתח הפרטי לעולם לא צריך להישלח לשולח.

למרות חוזק מתמטי אדיר זה, מחקרים הראו כי ניתן לשחזר מפתחות פרטיים של RSA מבלי לשבור ישירות את RSA אלא באמצעות Timing attack.

במתקפה מסוג זה התוקף מתבונן בזמן הריצה של אלגוריתם הצפנה ובכך מסיק את הפרמטר הסודי הכרוך בפעולות. למרות שמקובל להסכים כי RSA מוגן מפני התקפה ישירה, הפגיעות של RSA למתקפות תזמון אינה ידועה כל כך ולעתים קרובות מתעלמים ממנה.

כעת נממש מתקפה מסוג זה. אך לפני זה נקדים ונסביר כיצד מממשים מערכת הצפנה של RSA (אצלנו היא תמומש בתוך לוח Arduino) ותוך כדי נגיע למתקפה.

כפי שכתבנו, במערכת RSA יש שני מפתחות א-סימטריים, אחד פרטי - d ואחד ציבורי - e. והכל בעולם מודולו N (על עולם האריתמטיקה המודולרית ניתן לקרוא כאן), שהוא מכפלת הגורמים שהזכרנו.

בשביל להצפין הודעה m, מערכת ההצפנה מבצעת את הפעולה $m^d \ mod(n)$ (חזקה ומודולו) שלה נקרא modular exponent. את הפעולה modular exponent נבצע באמצעות אלגוריתם Repeated Squaring שהוא שיטה לביצוע modular exponent בצורה יעילה. על השיטה הזאת נפרט כבר, מכיוון ששם המתקפה שלנו מתקיימת. Repeated Squaring משתמש בפעולת הכפל. ומכיוון שמדובר בכפל מספרים גדולים ממש, הוא משתמש בשיטה נוספת שנקראת Mongomery Product, שמבצע פעולת כפל של מספרים גדולים בצורה יעילה.

Mongomery Product דורש הכנה מוקדמת של מספר פרמטרים. ואותם נקבל מאלגוריתם nPrime. באלגוריתמים Mongomery Product ו-nPrime- לא נתעמק כאן, מכיוון שהם רק מבצעים את פעולת הכפל בצורה יעילה ולא משפיעים על מימוש המתקפה (לפחות לא באופן ישיר). מידע נוסף עליהם ניתן למצוא כאן.

Repeated Squaring

כפי שאמרנו Repeated Squaring הוא שיטה לביצוע modular exponent בצורה יעילה והיא עומדת בבסיס המתקפה שלנו. בשיטה זו אנו נמיר את d (המפתח הפרטי) למספר בינארי ואז נסרוק את מחרוזת הביטים משמאל לימין.

בכל ביט אנו נעלה את m בריבוע (Mongomery Product). ובנוסף, אם הביט הינו 1, נכפיל את הערך שקיבלנו עם ההודעה המקורית (גם פה Mongomery Product). כמובן שבכל איטרציה (כאשר סורקים את הביטים משמאל לימין) מבצעים הפחתה (%n) במידת הצורך כדי להישאר בעולם המודולו n.

```
// Compute y = x^d (mod N)
// where, in binary, d = (d0,d1,d2,…,dn) with d0 = 1
s = x
for i = 1 to n
    s = s^2 (mod N)
    if di == 1 then
        s = s · x (mod N)
    end if
next i
return s
```

היתרון בשיטה נעוץ בכך שבכל שלב אנו מסתכלים רק על ביט בודד ובכך שבסוף כל שלב אנו מבצעים הפחתה. ככה אנו נשארים תמיד במספרים נמוכים (נמוכים ביחס למספרים שאותם אנו כופלים) שקל לבצע עליהם פעולות.

נראה דוגמה כיצד Repeated Squaring הופך לנו את הפעולה $12^{45}\ mod(40)$ לפעולה פשוטה:

במקום לחשב משהו קצת לא אפשרי כמו:

$$12^{45} = 36572619880088371967140823026550308340274372280 32$$

ואז לחשב את התוצאה:

$$36572619880088371967140823026550308340274372280 32\ mod(40) = 32$$

נפעל בצורה כזאת:

$$12^{45}\ mod(40)$$

נמיר את החזקה למספר בינארי ונסרוק אותו משמאל לימין:

$$(45)_{10} = (101101)_2$$

הביט הראשון הוא תמיד 1 ונתעלם ממנו. הביט השני הוא 0 ולכן רק נעלה בריבוע:

$$12^2 = 144 = 24\ [mod(40)]$$

הביט השלישי הוא 1 ולכן גם נעלה בריבוע וגם נכפיל במספר המקורי:

$$24^2 * 12 = 6912 = 32\ [mod(40)]$$

הביט הרביעי הוא 1 ולכן גם נעלה בריבוע וגם נכפיל במספר המקורי:

$$32^2 * 12 = 12288 = 8\ [mod(40)]$$

הביט החמישי הוא 0 ולכן רק נעלה בריבוע:

$$8^2 = 64 = 24\ [mod(40)]$$

הביט השישי הוא 1 ולכן גם נעלה בריבוע וגם נכפיל במספר המקורי:

$$24^2 * 12 = 6912 = 32\ [mod(40)]$$

ניתן לראות שהגענו לתוצאה בצורה פשוטה הרבה יותר.

אבל, ההפחתה הזאת שמבצעים בסוף כל שלב - היא נקודת החולשה של המערכת שננצל במתקפה.

אצלנו במערכת ה-Arduino יקבל ערכים ויחתום עליהם (כלומר יצפין אותם) באמצעות המפתח הפרטי. את הקוד של החתימה ב-Arduino ניתן למצוא כאן והוא יראה כך:

```
uint64_t MongomeryProduct(uint64_t a, uint64_t b, uint64_t n, uint64_t nprime, uint64_t r)
{
    //This function calc the montgomery product of numbers a and b
    uint64_t t = a * b;
    uint64_t t1 = t % r;
    uint64_t m = t1 * nprime % r;


    uint64_t t_div_r = t / r;
    uint64_t mn_div_r = (m * n) / r;

    uint64_t t_apr_r = t % r;
    uint64_t mn_apr_r = (m * n) % r;


    uint64_t u = t_div_r + mn_div_r  +1;

    if (u < n)
        return u;
    delay(100);
    return u - n;
}

uint64_t modexp(uint64_t a, String exp, uint64_t n, uint64_t r, uint64_t k) {
    //This function encrypt/decrypt message m by rsa protocol
    uint64_t a_ = (a * r) % n;
    uint64_t x_ = (1 * r) % n;

    for (int i = exp.length() - 1; i >= 0; i--)
    {
        x_ = MongomeryProduct(x_, x_, n, k, r);
        if (exp[i] == '1')
            x_ = MongomeryProduct(a_, x_, n, k, r);

    }

    return MongomeryProduct(1, x_, n, k, r);
}

void nPrime(uint64_t n)
{
    //This function calc the r and the k for the montgomery product
    uint64_t k = (log(n)/log(2)) + 1;
    uint64_t r = pow(2, k);
    uint64_t rInverse = ModInverse(r, n);
    uint64_t nPrime = (r * rInverse - 1) / n;
    result[0] = nPrime;
    result[1] = r;
}
```

נסביר את עיקרי הקוד:

ניתן לראות פונקציה בשם modxp, היא בעצם מצבעת את $m^d \ mod(n)$ באמצעות האלגוריתמים שהצגנו. הפרמטרים המתקבלים הם:

- a - ההודעה שאותה נחתום (m).
- exp - החזקה (d), שהיא המפתח הפרטי. היא מתקבלת כמחרוזת של ביטים ולא כמספר (בשביל אלגוריתם Repeated Squaring).
- n - עולם המודולו.
- r, t- הם פרמטרים מקדימים שהזכרנו שמסייעים לפונקציה והם מגיעים מהפונקציה nPrime.

בריבוע האדום ניתן לראות את אלגוריתם Repeated Squaring. כפי שניתן לראות בקוד, יש עוד המון מתמטיקה מסביב. כל המתמטיקה היא חלק מהאלגוריתמים שהזכרנו והיא לא קשורה ישירות למתקפה. על כל המתמטיקה ניתן לקרוא פה, פה ופה. לצערנו, כפי שאתם רואים, אנחנו לא נרחיב על כל המתמטיקה מאחורי RSA. בעיקר כדי להשאר בגבולות המאמר :)

הרעיון העומד מאחורי המתקפה

נניח שאנו נשלח שתי הודעות Y, Z כך שמתקיים - $Y^3 < \mathrm{N}$ וגם מתקיים - $Z^2 < \mathrm{N} < Z^3$. כלומר, Y לא יצטרך הפחתה גם אם נעלה אותו בריבוע ונכפיל בערך המקורי שלו (כלומר נעלה בשלישית) מכיוון שהוא ישאר קטן יותר מעולם המודולו n. ולעומת זאת, Z לא יצטרך הפחתה רק אם נעלה אותו בריבוע. אבל ברגע שנכפיל אותו אח"כ בערך המקורי שלו (כלומר נעלה בשלישית) הוא יצטרך הפחתה מכיוון שהוא יהיה גדול יותר מעולם המודולו n.

כעת לאחר שאנו שולחים את ההודעות לחתימה, ניתן לחלק את המצב לשני מקרים:

- **במקרה הראשון** - ביט החזקה באלגוריתם Repeated Squaring הוא 1, לכן נצטרך להכפיל את ההודעה הנחתמת בהודעה המקורית ולא רק להעלות אותה בריבוע. במקרה כזה, משך זמן החתימה של הודעה Z ייקח יותר זמן. כי כאשר מעלים את הודעה Z בשלישית היא תהיה גדולה מעולם המודולו n. לעומת זאת, הודעה Y שלא תהיה גדולה מעולם המודולו n גם כאשר מעלים אותה בשלישית, לא תדרוש הפחתה. לכן זמן החתימה שלה יהיה קצר יותר.

- **במקרה שני** - ביט החזקה באלגוריתם Repeated Squaring הוא 0, לכן נצטרך רק להעלות את ההודעה הנחתמת בריבוע, בלי להכפיל את ההודעה הנחתמת בהודעה המקורית. במקרה כזה זמן החתימה של שתי ההודעות יהיה שווה מכיוון ששתיהן לא יצטרכו הפחתה.

כעת, כששלחנו את ההודעות לחתימה ואנו יודעים מהם זמני הריצה שלוקח לחתום אותן, נוכל להשוות בין זמני הריצה של Y וZ.

אם זמני החתימה הם פחות או יותר שווים, נדע שהביט הבא הוא 0 (כי לא הייתה הפחתה להודעה Z ולכן לא לקח לה יותר זמן). ולעומת זאת אם זמני החתימה יהיו שונים, נדע שהביט הבא הוא 1 (כי להודעה Z הייתה הפחתה ולכן זמן החתימה שלה ארוך יותר).

את התהליך הזה נעשה שוב ושוב עבור כל הביטים במפתח, עד שנמצא את המפתח הפרטי שמתאים למפתח הציבורי שברשותנו.

תשתית למתקפה

כדי שנוכל לבצע את המתקפה בצורה יעילה, נצטרך לבצע את הפעולות שהסברנו עם אלפי הודעות. לשם כך נבנה מערכת תקשורת נוחה עם לוח ה-Arduino שלנו - שהוא מערכת החתימה שלנו.

את התקשורת איתו נבצע באמצעות סקריפט שנכתוב בפיתון (נשתמש בספריה PySerial). הסקריפט יתקשר עם הפורט הסריאלי של ה-Arduino. באמצעות כך נשלח ללוח הודעות לחתימה ונקבל ממנו את הערכים החתומים. במקביל הקוד ימדוד את זמן החתימה במיקרו שניות.

עקב מגבלות של לוח ה-Arduino לעבוד עם מספרים של 64 ביט, הוא יקבל את ההודעה לחתימה כמחרוזת של ביטים, ימיר אותה למספר, יחתום אותה. ולאחר מכן ימיר אותה בחזרה למחרוזת של ביטים ויחזיר את החתימה. את שלושת הערכים האלו (ההודעה המקורית, ההודעה החתומה והזמן) נשמור בתוך קובץ שלאחר מכן נתבסס עליו לביצוע המתקפה כפי שיוסבר בהמשך.

הקוד שלנו יראה כך:

```
def write(data):
    while not arduino.writable():
        pass
    arduino.write(data.encode())


def read():
    while not arduino.readable():
        pass
    data = ""
    while data == "":
        data = arduino.readline().decode()
    return data
```

כפי שניתן לראות, יש לנו פונקציה write שמקלת נתונים ומעבירה אותם ללוח. בנוסף יש לנו פונקציה read שממתינה לנתונים שיחזרו מהלוח.

ניתן למצוא את הקוד כאן.

תהליך המתקפה

כפי שהסברנו, בכדי שיהיה חומר לניתוח הזמנים ולמציאת המפתח הפרטי, נשלח אלפי הודעות רנדומליות למערכת החתימה שלנו ובמקרה שלנו ללוח ה-Arduino, על מנת שיחתום אותן באמצעות המפתח הפרטי ויחזיר לנו את הערך חתום. כמובן שעבור כל הודעה נמדוד כמה זמן לקח למערכת לחתום אותה.

אחרי שיהיה ברשותנו מאגר של אלפי הודעות, החתימות של אותן הודעות וזמן החתימה שלהן, נוכל לבצע את הניתוח. בשלב הראשוני ידוע לנו שהביט הראשון של המפתח הפרטי הוא 1 ואנחנו רוצים לגלות כל פעם מה הביט הבא במפתח. כעת ניקח את רשימת ההודעות ששלחנו ונפעיל על כל אחת מההודעות את האלגוריתם Repeated squaring עד הביט שידוע לנו. עבור כל אחת מההודעות נבדוק האם בביט הבא בהנחה שהוא 1 יש הפחתה או לא (כלומר, האם היה צורך ב-% או לא).

כך נחלק בעצם את הרשימה של אלפי ההודעות שלנו לשתי קבוצות:

- **קבוצה א'** - כל ההודעות שבהן היה צורך בהפחתה אם הביט הבא הוא 1.
- **קבוצה ב'** - שאר ההודעות. כלומר ההודעות שבהן לא היה צורך בהפחתה אם הביט הבא הוא 1.

כעת נחשב את ממוצע זמן הריצה עבור כל קבוצה לפי הזמנים שמופיעים במאגר שלנו (לא לפי הזמן שלקח כעת).

לאחר מכן נבדוק, אם הפרש הזמנים בין הקבוצות שחילקנו הינו זניח, המשמעות היא שאין באמת הגיון כלשהו בחלוקה שעשינו (כלומר לחלק מההודעות עשינו הפחתה סתם, ללא צורך. ולכן לקח יותר זמן לחתום אותן) ולכן ההשערה שלנו שהביט הוא 1 שגויה והביט הבא הוא 0.

אך לעומת זאת, אם הפרש הזמנים בין הקבוצות שחילקנו אינו זניח אלא משמעותי, ניתן להסיק שיש הגיון בחלוקה שביצענו (כלומר ניתן להסיק שבאמת פעלנו נכון כאשר לקבוצה אחת של הודעות עשינו הפחתה ולקבוצה השנייה לא) ולכן באמת הביט הבא הוא 1 כפי שהנחנו.

בניית מאגר ערכים למתקפה

כעת נעבור לפונקציה הראשית: הפונקציה הראשית תגריל כמה אלפי מספרים בגדלים שונים, מ-1 ועד n. הסיבה שנגריל רק עד n היא מכיוון שזה עולם המודולו שלנו. לכן אם נגריל יותר מ-n נבצע הפחתה שלא לצורך - מבלי קשר למתקפה.

עבור כל מספר שהתוכנית הראשית תגריל, היא תשלח אותו לחתימה ותמדוד כמה זמן היא ממתינה לתשובה. לאחר מכן היא תכתוב את כל המידע לתוך קובץ שישמש אותנו לניתוח הזמנים ומציאת המפתח הפרטי.

הקוד של התוכנית הראשית יראה כך:

```
n = 3839256683
e = 548447537

with open('timeData.txt', 'w') as f1:
    f1.writelines("N,E\n")
    f1.writelines(str(n) + "," + str(e) + "\n")
    f1.writelines("message,signature,duration\n")
    for i in range(1, 9000):
        m = random.randrange(1, n)
        t1 = datetime.now()
        write(longToBytes(int(m)))
        c2 = stringToInt(read())
        t2 = int((datetime.now() - t1).total_seconds() * 1000000)
        f1.writelines(str(m) + "," + str(c2) + "," + str(t2) + "\n")
```

ניתן לראות בקוד שבתוך הלולאה אנו מגרילים ערך כפי שהסברנו לפני, חותמים אותו (באמצעות הלוח) ומקבלים את התשובה. בין לבין אנו מודדים זמנים ואת הכל כותבים בסופו של דבר לתוך הקובץ timeData.txt.

סה"כ נקבל בסוף קובץ עם 9000 שורות שיראה כך:

```
N,E
3839256683,548447537
message,signature,duration
1825426693,481823779,2280893
2110686704,3072422398,2497166
1334428208,1599528745,2093178
1265195384,2942524450,2186568
2409436678,1375084233,2093802
401252935,2464631800,2186973
1695206886,2565208005,2690551
2933055133,329494375,2593669
```

בתחילת הקובץ שמרנו את המפתח הציבורי ואת המודולו N.

כמו כן בכל שורה ניתן לראות את ההודעה, ההודעה חתומה ואת הזמן במיקרו שניות שלקח למערכת לחתום אותה. כל הנתונים האלו ישמשו את הסקריפט הבא שלנו שינתח את הקובץ וימצא מהו המפתח הפרטי. זמן ההכנה של מאגר כזה הוא כשעה (תלוי בגודל המפתח וכמה דברים נוספים).

את המאגר הזה ומאגרים נוספים ניתן למצוא כאן.

ניתוח המאגר

כעת ניקח את המאגר וננתח אותו באמצעות סקריפט נוסף.

בכל שלב כפי שהסברנו, הסקריפט יניח שהביט הבא הוא 1 וינסה לחלק את ההודעות החתומות לפי זה (קבוצה אחת להודעות שהיו צריכות הפחתה וקבוצה שניה להודעות שלא היו צריכות הפחתה בזמן החתימה). כמובן שלצורך כך הסקריפט המפענח יחתום בעצמו לפי אלגוריתם RSA שממומש עם Repeated Squaring ועם Mongomery Product.

לכן הקוד החותם (מצפין) שלנו יראה בדיוק כמו שראינו למעלה בלוח ה-Arduino אך עם תוספת של דגל נוסף בשם red שמסמל האם הייתה הפחתה או לא. ככה שביחד עם תוצאת החתימה נדע לאיזה קבוצה לסווג את ההודעה:

```
def rsa_sim(m, d, n, k, r, j):
    """
        This function simulate a encrypt/decrypt message m by rsa protocol
        :param m: The message we want to encrypt/decrypt
        :param d: The key
        :param n: The modulo
        :param k: The k for montgomery product
        :param r: The r for montgomery product
        :param j: The bit we want to test if we have reduce or not
        :return: The encrypt/decrypt of message m and True/False if we have reduce or not
        """
    m_ = (m * r) % n
    x_ = (1 * r) % n
    new_d = d[:j]
    new_d += '1'
    red = False
    for i in range(0, len(new_d)):
        x_, _ = MontgomeryProduct(x_, x_, n, k, r)
        if new_d[i] == '1':
            x_, red = MontgomeryProduct(m_, x_, n, k, r)
    x, _ = MontgomeryProduct(x_, 1, n, k, r)
    return x, red
```

בריבוע האדום אנו "קובעים" מהו המפתח שעכשיו נעבוד עליו. שזה בעצם שרשור של המפתח שמצאנו עד עכשיו בתוספת ההנחה שהביט הבא הוא 1 כפי שהסברנו.

הפונקציה הבאה מקבלת כל פעם הודעה מתוך המאגר, שולחת לפונקציה rsa_sim ומקבלת בחזרה האם התקיימה הפחתה או לא.

לפי זה היא מסווגת את ההודעות לשתי קבוצות, red אם הייתה הפחתה ו-nored אם לא הייתה הפחתה.

כפי שרואים בקוד:

```python
def split_messages(d, n, k, r, bit, dataList):
    """
    This function split the messages to two groups. reduce in bit(bit) or not
    :param d: The key
    :param n: The modulo
    :param k: The k for montgomery product
    :param r: The r for montgomery product
    :param bit: The bit we want to test if we have reduce or not
    :param dataList: The messages
    :return: two groups. reduce in bit(bit) or not
    """
    red = []
    nored = []
    for m in dataList:
        c, bucket = rsa_sim(m[0], d, n, k, r, bit)
        if bucket:
            red.append(m)
        else:
            nored.append(m)
    return red, nored
```

הפונקציה הבאה היא הפונקציה החשובה שלנו ולכן נעבור עליה שורה אחר שורה. היא נקראת מה-main ותפקידה לעבור על המאגר ולמצוא את המפתח בפועל.

```python
def RSATimingAttack(n, data, ratio):
    """
    This function analysis the time that took to encrypt the messages and find the private key
    :param n: The modulo
    :param data: the messages with the time
    :param ratio: the delta
    """
    (r, k) = nPrime(n)
    private_key = '1'
    bit = 1
    finished = False
    while not finished and bit < 32:
        (red, nored) = split_messages(private_key, n, k, r, bit, data)
        red_avg = calcAvg([m[2] for m in red])
        nored_avg = calcAvg([m[2] for m in nored])
        print("Difference of averages:", abs(red_avg - nored_avg))

        if abs(red_avg - nored_avg) > ratio:
            private_key += '1'
            print("The next bit is probably 1.")
        else:
            private_key += '0'
            print("The next bit is probably 0.")
        print("The private key until now is: ", private_key)
        print()

        if testKey(data, private_key, n, k, r):
            print("We did it! The private key is: \t", private_key)
            finished = True

        bit += 1

    if not finished:
        print("can't find the private key...")
```

בשורה 139 ניתן לראות שהפונקציה מקבלת שלושה פרמטרים:

- n - עולם המודולו.
- data - מאגר ההודעות שאותן נבחן.
- ratio - טווח הסטייה שלפיו נקבע האם צדקנו בהנחה שהביט הוא 1 או שהביט צריך להיות 0. בסוף נצלול אל השימוש בו וכיצד קובעים אותו.

לאחר מכן ב-46 נכין את המשתני עזר ל-Mongomery Product באמצעות nPrime כפי שכבר ראינו לפני. בשורה הבאה נאתחל את המשתנה שישמור את המפתח הפרטי לאורך הדרך ושורה אחרי נאתחל מונה שסופר כמה ביטים כבר עברנו. לאחר מכן בשורות 170 - 150 נכנס ללולאה שעובדת עד שנמצא את המפתח הפרטי (או עד שנגיע ל-32 ביט - כדי לדעת שמשהו לא עובד).

בתוך הלולאה בשורה 151 קודם כל נסווג את ההודעות לשתי קבוצות כפי שהסברנו, קבוצה red שלה הייתה הפחתה ו-קבוצה nored שלה לא הייתה הפחתה.

כעת בשורות 153 - 152 נחשב את ממוצע הזמנים עבור כל קבוצה ולאחר מכן בשורה 156 נבדוק האם יחס הזמנים בין הקבוצות לא זניח (גדול מ-ratio) ולכן אכן הביט הוא 1 (שורה 157) או שהוא זניח (קטן מ-ratio) ולכן הביט הוא 0 (שורה 160). כיצד לקבוע את ratio נסביר בהמשך.

לבסוף בשורה 165 אנו נפעיל את הפונקציה testKey שתבדוק האם צדקנו במפתח עד עכשיו ואם כן - נעצור את הלולאה.

הפונקציה testKey פשוט לוקחת מספר ערכים מהמאגר, חותמת אותם עם המפתח שמצאנו ומחזירה האם התוצאה נכונה או לא לפי מה ששמור במאגר.

כעת נריץ את הסקריפט ונראה כיצד הוא מוצא את המפתח הפרטי.

דוגמת הרצה

בדוגמת ההרצה שלנו ה-ratio יהיה 150000. את הקוד נריץ על המאגר שבנינו לפני בפרק של בניית מאגר ערכים למתקפה:

```
N,E
3839256683,548447537
message,signature,duration
1825426693,481823779,2280893
2110686704,3072422398,2497166
1334428208,1599528745,2093178
1265195384,2942524450,2186568
2409436678,1375084233,2093802
401252935,2464631800,2186973
1695206886,2565208005,2690551
2933055133,329494375,2593669
```

נזכיר שהמאגר מכיל נתוני חתימה של 9000 הודעות. כעת נריץ את הסקריפט (זמן הריצה הוא בערך חצי דקה) ונקבל המון שורות כאלה:

```
The first bit is 1

Difference of averages: 187986.2268589628
The next bit is probably 1.
The private key until now is:  11

Difference of averages: 173582.78072837694
The next bit is probably 1.
The private key until now is:  111

Difference of averages: 137093.0452079922
The next bit is probably 0.
The private key until now is:  1110
```

עד שלבסוף נקבל:

```
Difference of averages: 206346.6489106221
The next bit is probably 1.
The private key until now is:  11100100110101001000010001010001

We did it! The private key is:   11100100110101001000010001010001
```

נסביר מה קיבלנו:

ניתן לראות שבכל שלב, הסקריפט ידפיס לנו שלוש שורות: בשורה הראשונה ההפרש בין ממוצע הזמנים עבור שתי הקבוצות ולאחר מכן בשורה הבאה האם הביט הוא 0 או 1. נזכיר שזה נקבע לפי הערך ratio (שבמקרה שלנו הוא 150000) ותלוי האם הפרש ממוצע הזמנים בין הקבוצות לא זניח (גדול מ-ratio) ולכן אכן הביט הוא 1 או שהוא זניח (קטן מ-ratio) ולכן הביט הוא 0.

בשורה הבאה הסקריפט ידפיס את המפתח שהוא מצא עד כה.

לבסוף, בשלב מסוים, הפונקציה testKey זיהתה שמצאנו את המפתח הפרטי ולכן היא עצרה את הלולאה והודפסה לנו ההודעה על סיום הפעולה והמפתח:

```
We did it! The private key is:   11100100110101001000010001010001
```

קביעת הערך ratio

קביעת הערך ratio שישמש את הסקריפט תוך כדי ניתוח הזמנים על מנת לקבוע האם הפרש הזמנים זניח או לא זאת פעולה מסובכת .

הערך ratio נקבע לפי חישובים מתמטיים שמתחשבים בזמני הריצה של המערכת, זמן ביצוע החתימה, זמן ביצוע הפחתה, המהירות של המערכת (אצלנו גם השימוש בלוח ה-Arduino שמשפיע מאוד על המהירות) וכו'. באופן כללי ניתן לומר, שזה תלוי בכמה זמן לוקח לנו לחתום על ערך בודד ולקבל את התוצאה.

כמובן שבמקום זה אפשר להריץ את הקוד מספר פעמים, בכל פעם עם ערך ratio אחר עד שנמצא את הערך הנכון (כשנמצא את המפתח הפרטי). אומנם זה ייקח קצת זמן, אבל בסוף מדובר על 5 דקות של ריצה במקום חצי דקה וזה לא משמעותי.

דרך נוספת היא להתבונן בנתונים ולנסות להסיק מהם מהו הערך הנכון. כך נעשה את זה: קודם כל, נציב ב-ratio את הערך 0 ונריץ שוב את המערכת, כמובן שהיא תיכשל במציאת המפתח:

```
can't find the private key...
```

אבל, נסתכל על הזמנים שהודפסו לנו וננסה להסיק מהם מה הערך ratio שלנו.

ננסה לזהות שני קבוצות של זמנים, קבוצה אחת עם ערכים גבוהים יותר וקבוצה שניה עם ערכים נמוכים יותר, את הערך ratio נקבע איפשהו ביניהם.

```
The first bit is 1

Difference of averages: 187986.2268589628
The next bit is probably 1.
The private key until now is:  11

Difference of averages: 173582.78072837694
The next bit is probably 1.
The private key until now is:  111

Difference of averages: 137093.0452079922
The next bit is probably 1.
The private key until now is:  1111

Difference of averages: 147840.38634238113
The next bit is probably 1.
The private key until now is:  11111

Difference of averages: 133634.03459492
The next bit is probably 1.
The private key until now is:  111111

Difference of averages: 141838.72920653317
The next bit is probably 1.
The private key until now is:  1111111

Difference of averages: 127732.18294484867
The next bit is probably 1.
The private key until now is:  11111111

Difference of averages: 130305.78003233159
The next bit is probably 1.
The private key until now is:  111111111

Difference of averages: 137619.8641706151
The next bit is probably 1.
The private key until now is:  1111111111

Difference of averages: 139925.38081441494
The next bit is probably 1.
The private key until now is:  11111111111

Difference of averages: 144675.34778241627
The next bit is probably 1.
The private key until now is:  111111111111

Difference of averages: 145315.79071280127
The next bit is probably 1.
The private key until now is:  1111111111111

Difference of averages: 137270.6971478518
The next bit is probably 1.
The private key until now is:  11111111111111

Difference of averages: 136953.3723948421
The next bit is probably 1.
The private key until now is:  111111111111111

Difference of averages: 140107.39883236773
The next bit is probably 1.
The private key until now is:  1111111111111111

Difference of averages: 131728.17272960348
The next bit is probably 1.
The private key until now is:  11111111111111111

Difference of averages: 140500.97501439275
The next bit is probably 1.
The private key until now is:  111111111111111111

Difference of averages: 135371.55694211507
The next bit is probably 1.
The private key until now is:  1111111111111111111
```

```
Difference of averages: 142729.97881431598
The next bit is probably 1.
The private key until now is:  11111111111111111111

Difference of averages: 132948.27755187592
The next bit is probably 1.
The private key until now is:  111111111111111111111

Difference of averages: 134367.15433216142
The next bit is probably 1.
The private key until now is:  1111111111111111111111

Difference of averages: 145426.18276158022
The next bit is probably 1.
The private key until now is:  11111111111111111111111

Difference of averages: 131342.06003474537
The next bit is probably 1.
The private key until now is:  111111111111111111111111

Difference of averages: 136155.43027170328
The next bit is probably 1.
The private key until now is:  1111111111111111111111111

Difference of averages: 130312.2138843555
The next bit is probably 1.
The private key until now is:  11111111111111111111111111

Difference of averages: 132867.6554789669
The next bit is probably 1.
The private key until now is:  111111111111111111111111111

Difference of averages: 126829.95983718289
The next bit is probably 1.
The private key until now is:  1111111111111111111111111111

Difference of averages: 138589.06965380115
The next bit is probably 1.
The private key until now is:  11111111111111111111111111111

Difference of averages: 147822.24046116136
The next bit is probably 1.
The private key until now is:  111111111111111111111111111111

Difference of averages: 135589.44690011628
The next bit is probably 1.
The private key until now is:  1111111111111111111111111111111

Difference of averages: 122560.57186330017
The next bit is probably 1.
The private key until now is:  11111111111111111111111111111111

can't find the private key...
```

(באדום ניתן לראות שעם ratio = 0 - הסקריפט נכשל)

אם נביט בזמנים שהודפסו, נראה שיש לנו קבוצה נמוכה יותר של זמנים שהם באזור 130000 וקבוצה של זמנים יותר גבוהים באזור 15000. לכן נקבע את ratio להיות איפשהו באמצע, למשל 140000.

נריץ את הסקריפט שוב ונראה שגם הוא נכשל:

```
def main():
    difference = 140000
RSATimingAttack() › if not finished
findTheKey
The private key until now is:  1110010011011010000010110000001 0

can't find the private key...
```

באדום ניתן לראות את הערך ratio ואת ההודעה על כך שהמפתח הפרטי לא נמצא. לכן נחזור לזמנים וננסה לנתח אותם שוב בצורה חדה יותר.

אם נבחן את הזמנים שהודפסו שוב, נראה שלא דייקנו מספיק בחלוקה שלנו. ניתן לראות שאומנם יש לנו קבוצה נמוכה יותר של זמנים שהם באזור 130000, אך הקבוצה של הזמנים היותר גבוהים היא באזור 160000 ולא באזור 15000.

לכן נקבע את ratio להיות שוב איפשהו באמצע, הפעם 150000. כמובן שניתן לחלק את הזמנים לשתי קבוצות ולמצוא את האמצע ביניהם עם אלגוריתם שזה תפקידו (למשל knee) אבל ניתן להסתדר בלי זה.

נריץ את הסקריפט שוב ונקבל את הפתח הפרטי. כלומר צדקנו בניחוש הערך ratio:

```
def main():
    difference = 150000
RSATimingAttack()
findTheKey
The private key until now is:  1110010011010100100001000101000 1

We did it! The private key is:   1110010011010100100001000101000 1
```

באדום ניתן לראות את הערך ratio ואת המפתח הפרטי וההודעה על כך שהוא נמצא. הצלחנו לנחש את הערך ratio.

סיכום

במאמר זה עסקנו במתקפת ערוץ צד. כפי שהראנו, ניתן לבצע מתקפת ערוץ צד על RSA וע"י כך למצוא את המפתח הפרטי. במהלך המאמר התנסנו שוב בשימוש בלוח Arduino. עליו ממשנו מערכת הצפנת RSA, שאותה תקפנו.

הפרויקט היה חוויתי ומהנה והעמיד אותנו מול אתגר חדש שאיתו התמודדנו והוא מימוש מתקפה על RSA. מדובר במתקפה מורכבת שדרשה מאיתנו ללמוד כיצד מערכות RSA ממומשות וכיצד לנצל את נקודת התורפה של צורת המימוש על מנת לבצע מתקפת ערוץ צד ולמצוא את המפתח הפרטי.

קצת עלינו

אבי פדר, בן 22, משתתף בתוכנית סייבר עילית וסטודנט שנה ד' להנדסת תוכנה אשכול סייבר במרכז האקדמי לב. מתעתד להתגייס בקרוב ומחפש את מקומי בצבא כעת. מוזמנים גם לעקוב אחרי ב-linkedin.

דניאל יוחנן, בן 21, עתודאי שנה ד' להנדסת תוכנה אשכול סייבר במרכז האקדמי לב ובנוסף עובד בתחום ה-AI. מתעתד להתגייס בקרוב.

אם יש לכם שאלות, הערות או כל דבר אחר נשמח לשמוע מכם באימייל:

Avifeder99@gmail.com

Danielyochanan91@gmail.com

נוסיף שלקח לנו המון זמן לחקור, לפתח ולכתוב את המאמר. זה היה אתגר מעניין שקידם אותנו מאוד ונמליץ עליו גם לכם. שמחנו מאוד לעשות את זה ומקווים שייצא לנו שוב בעתיד.

ונסיים בתודה ל**מר אריה הנל**, מרצה במרכז האקדמי לב שאיתו למדתנו על כל זה.

ובתודה ענקית לעורכי המגזין שבזכותם יש לכולנו תוכן עשיר, איכותי ואמין בצורה נגישה ונוחה.

מקורות

https://www.digitalwhisper.co.il/files/Zines/0x4F/DW79-2-SideChannel.pdf

https://github.com/avifeder/SideChannelAttack/tree/main/Part3

https://en.wikipedia.org/wiki/Modular_arithmetic

https://en.wikipedia.org/wiki/Montgomery_modular_multiplication

https://en.wikipedia.org/wiki/Exponentiation_by_squaring

https://en.wikipedia.org/wiki/Modular_exponentiation

https://pyserial.readthedocs.io/en/latest/pyserial.html

https://www.cs.sjsu.edu/faculty/stamp/students/article.html

http://www.mat.ucm.es/congresos/mweek/XII_Modelling_Week/Informes/Report5.pdf

https://slideplayer.com/slide/4976535/

https://en.wikipedia.org/wiki/Side-channel_attack