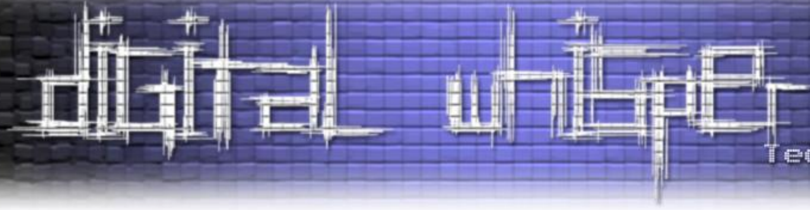


התקפות ידועות על Elliptic Curve Cyptography

מאת אלי קסקי

הקדמה

בשנים האחרונות גישת ה-Elliptic Curve Cryptography נהייתה פופולרית יותר ויותר בעולם בשל היעילות שלה והבטיחות הגבוהה שלה. המוטיבציה שלי ללמידה על הנושא הזה באה בעקבות תחרויות Capture The Flag (CTF) שאני נוהג להשתתף בהן באופן קבוע. הקטגוריה האהובה עליי בתחרויות אלה היא קריפטוגרפיה, ובכללי כל העולם הזה מאוד מעניין אותי. היכולת להחביא מידע מבלי שאחרים יוכלו לקרוא אותו, או להצפין הודעה, להפיץ אותה, ולהיות בטוח שרק מי שאמור לקבל אותה יוכל להבין אותה, אלה נושאים שאני אוהב ללמוד ולעסוק בהם.

אך תמיד כשנתקלתי באתגר CTF שעוסק ב-Elliptic Curve Cryptography ויתרתי עליו מראש כי מעולם לא למדתי את הנושא הזה. עד שיום אחד החלטתי לשנות את זה. תהליך הלמידה היה קשה. בכל מה שקשור למתקפות, לא הצלחתי למצוא חומרים בשפה פשוטה שמסבירים קונספטים בצורה אינטואיטיבית. מצאתי בעיקר מאמרים אקדמיים עם מתמטיקה ברמה הרבה יותר מדי גבוהה להבנה שלי. באיזשהו שלב הגעתי למסקנה שאין טעם להבין לעומק כל הוכחה או תכונה מתמטית, ועדיף להתייחס לתופעות מתמטיות כ-Black Box. בדיעבד, ככל שהתקדמתי בלמידה ראיתי שעוד אנשים ממליצים על כך גם, בהנחה שהקורא מתעניין ביישום הקריפטוגרפי ולא בגאומטריה האלגברית עצמה.

המטרה של מאמר זה היא להנגיש את החומר שמצאתי ובתקווה להעביר אותו בצורה יותר ברורה יחסית לצורה שבה הוא קיים כיום באינטרנט. בנוסף, כתבתי אותו גם כדי לעשות לעצמי סדר בראש ולהסביר לעצמי דברים כדי שאני באמת אבין את הנושא הזה לעומק.

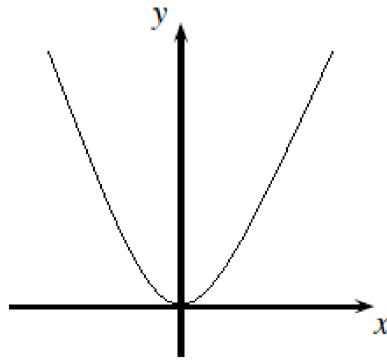
במאמר זה אני אציג מה הן עקומות אליפטיות, את הפעולות הבסיסיות שמוגדרות בהן, ומה ניתן לעשות איתן בהקשר קריפטוגרפי. עיקר המאמר מורכב מדוגמאות של מתקפות ידועות על מימושים או שימושים שגויים בהן. לכל אורך המאמר אני משתדל להפריד את ההסבר לחלק אינטואיטיבי ו-High Level, ולחלק מתמטי שיותר נכנס לפרטים. הקורא מוזמן להתמקד באיזה מהחלקים שמעניינים אותו באותו המקום, ולדלג על החלקים שפחות.

קריאה מהנה!

מבוא לעקומות אליפטיות

עקומה אלפיטית

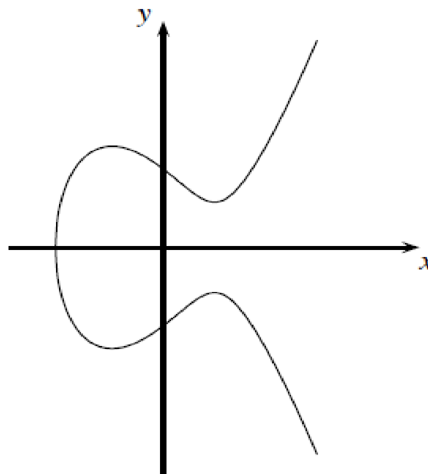
באופן כללי עקומה אליפטית היא איזשהו קו "עקום", כלומר לא ישר. דוגמא לכך היא הפרבולה, שהמשוואה שלה היא מהצורה $y = ax^2 + bx + c$ והיא נראית כך:



בהקשר של קריפטוגרפיה, נהוג להשתמש בעקומות אליפטיות שהמשוואה שלהן היא מהצורה:

$$y^2 = x^3 + ax + b$$

כך למשל נראית עקומה אליפטית המתאימה למשוואה $y^2 = x^3 - 3x + 3$:



משוואת העקומה מגדירה לנו את היחס בין ערך ה-x של נקודה על העקומה לערך ה-y של אותה הנקודה. בהקשר קריפטוגרפי, מגבילים את x, y, a, b להיות מספרים שלמים, ומגבילים את החישובים להיות מודולו מספר p ראשוני גדול כלשהו. אז משוואת העקומה האליפטית היא:

$$y^2 = x^3 + ax + b \ (mod\ p)$$

הדבר הזה גורם לכך שיש לנו מספר סופי של נקודות על העקומה. בשפה מתמטית, העקומה מוגדרת להיות מעל שדה סופי מסדר p. כתוצאה מכך עכשיו לא בהכרח לכל ערך x תהיה נקודה מתאימה על העקומה - כי יכול להיות שערך ה-y המתאים לו הוא לא מספר שלם.

נקודות על העקומה

אוסף הנקודות על העקומה מורכב מזוגות של מספרים שלמים (x, y) המקיימים את משוואת העקומה. בנוסף לנקודות האלה, מגדירים נקודה נוספת מיוחדת בשם "אינסוף" (Infinity), והיא מסומנת ב-$\mathcal{O}$. בשפה מתמטית, נקודה זו היא האיבר הנייטרלי של אוסף הנקודות על העקומה ביחס לפעולת החיבור, שאותה נגדיר מיד. מספר הנקודות שנמצאות על העקומה (כולל הנקודה $\mathcal{O}$) נקרא "הסדר של העקומה".

אבחנה נוספת היא שעקומות אליפטיות הן סימטריות ביחס לציר ה-X. מה שאומר שאם הנקודה $P = (x, y)$ נמצאת על העקומה, אז גם הנקודה $-P = (x, -y)$ נמצאת על העקומה. למעשה, נקודות אלה נחשבות "הופכיות" אחת של השניה (מכאן הסימון "$-P$" לנקודה השניה), ותוצאת פעולת החיבור ביניהן מוגדרת להיות האיבר הנייטרלי $\mathcal{O}$.

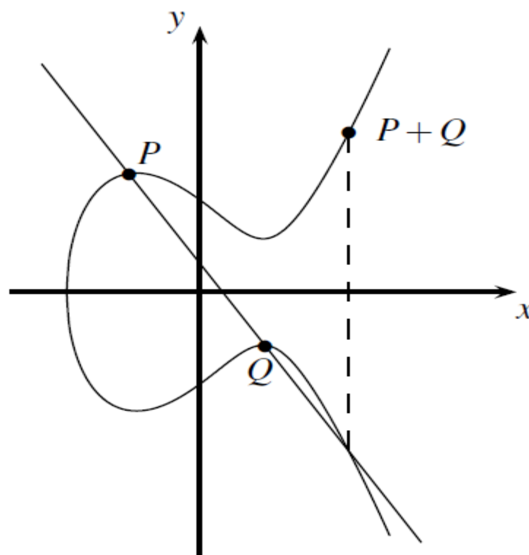
משפט בשם Hasse's Theorem מספק הערכה של $\#E$, הסדר של העקומה, והוא סדר גודל של $\Theta(p)$. ליתר דיוק:

$$p + 1 - 2\sqrt{p} \leq \#E \leq p + 1 + 2\sqrt{p}$$

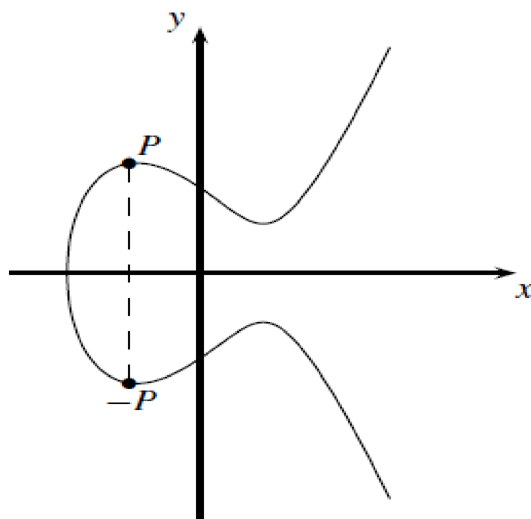
חיבור בין נקודות על העקומה

בהינתן שתי נקודות על העקומה, ניתן להגדיר פעולת חיבור ביניהן, שתוצאתה היא נקודה שלישית שגם היא נמצאת על העקומה. כדי למצוא את הנקודה הזאת בצורה גאומטרית, מעבירים קו בין שתי הנקודות הנתונות, וממשיכים אותו עד שהוא חותך את העקומה בנקודה שלישית. את הנקודה הזאת משקפים ביחס לציר ה-X, והנקודה המתקבלת מוגדרת להיות תוצאת החיבור.

להלן תרשים שמציג איך בהינתן נקודות P ו-Q ניתן למצוא את הנקודה $P + Q$:



שאלה שיכולה לעלות מתיאור זה היא מה קורה אם הקו שמעבירים בין שתי הנקודות לא חותך את העקומה שוב? במקרה זה אומרים שהקו חותך את העקומה ב"אינסוף", ותוצאת החיבור היא הנקודה $\mathcal{O}$. נשים לב שמקרה זה קורה אם הקו שמעבירים הוא אנכי, כלומר מנסים לחבר נקודה P עם הנקודה ההופכית שלה, $-P$:

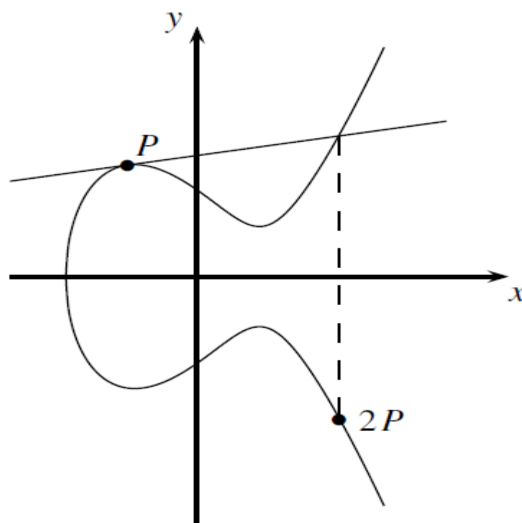


מכאן נגזרות שתי זהויות בסיסיות, לכל נקודה P מתקיים:

$$P + \mathcal{O} = P$$

$$P + (-P) = \mathcal{O}$$

שאלה נוספת שעולה מהתיאור הגיאומטרי היא איך מחברים נקודה עם עצמה? ראינו שכדי לחבר בין שתי נקודות שונות P ו-Q, מעבירים ביניהן קו ומסתכלים על נקודת החיתוך של ההמשך שלו עם העקומה. אינטואיטיבית, נשאיר את P קבועה, ונסתכל על הקו שנוצר לנו תוך כדי שאנחנו "מקרבים" את Q אל P יותר ויותר, עד ש-Q תתלכד עם P. מה שנקבל זה קו שיותר ויותר "משיק" לעקומה בנקודה P, וזה בדיוק הקו שנסתכל עליו כשנרצה לחבר את P עם עצמה:



כדי לחבר נקודה P עם עצמה, מעבירים משיק לעקומה בנקודה P, וממשיכים אותו עד שהוא חותך את העקומה בנקודה שניה. את הנקודה הזאת משקפים ביחס לציר ה-X, והנקודה המתקבלת מוגדרת להיות תוצאת החיבור. נהוג לסמן את תוצאת החיבור כ $P + P = 2P$. גם הפעם אם המשיק לא חותך את העקומה בנקודה שניה אז אומרים שהוא חותך את העקומה ב"אינסוף", ותוצאת החיבור היא הנקודה $\mathcal{O}$.

התיאורים הגאומטריים הויזואלים ממחישים בצורה נחמדה ועוזרים לנו להבין איך חיבור בין נקודות עובד. אבל איך בפועל מחשבים את זה? משוואות מתמטיות!

בהינתן הנקודות $P = (x_P,\ y_P)$ ו- $Q = (x_Q,\ y_Q)$, תוצאת החיבור היא הנקודה $R = (x_R,\ y_R)$ כך ש:

$$x_R = \lambda^2 - x_P - x_Q \ (mod\ p)$$

$$y_R = \lambda(x_P - x_R) - y_P \ (mod\ p)$$

כש-λ מוגדרת להיות שיפוע הקו שמחבר בין הנקודות, אם הן שונות, ושיפוע המשיק לעקומה בנקודה, אם מחברים את הנקודה לעצמה. ובאופן פורמלי:

$$\lambda = \begin{cases} \dfrac{y_P - y_Q}{x_P - x_Q} \quad (mod\ p) \quad ; \quad if\ P \neq Q \\ \dfrac{3{x_P}^2 + a}{2y_P} \quad (mod\ p) \quad ; \quad if\ P = Q \end{cases}$$

פרטי החישובים המתמטיים של פעולת החיבור בין נקודות הם לא קריטיים להמשך המאמר. לצורך העניין, ניתן להסתכל על חיבור בין נקודות כקופסא שחורה שמקבלת שתי נקודות על העקומה ומחזירה נקודה שלישית שנמצאת גם היא על העקומה.

כפל נקודה על העקומה בקבוע

ראינו שניתן לחבר נקודה P לעצמה, ואת הנקודה המתקבלת סימנו ב $2P$. אם לנקודה זו נחבר את הנקודה P שוב, תתקבל נקודה שנסמנה ב $3P$, וכך הלאה. בצורה זו אפשר להגדיר "כפל" של נקודה בקבוע, על ידי חיבור חוזר של הנקודה עם עצמה (באופן דומה לכפל בין מספרים):

$$nP = \underbrace{P + P + \cdots + P}_{n\ times}$$

לכאורה כדי להכפיל נקודה כלשהי במספר n אנחנו צריכים לבצע n פעולות חיבור בין נקודות. זה כי בהינתן נקודת התחלה, קשה לדעת מראש איפה תיפול הנקודה ה"אחרונה", בלי להגיע אליה "צעד אחרי צעד". חישוב שכזה יהיה מאוד לא יעיל, כי n יכול להיות גדול מאוד.

לצורך כך קיים האלגוריתם Double And Add, שבו מתחילים את התהליך מהנקודה P, ואז לכל ביט בייצוג הבינארי של n, מכפילים את הנקודה הנוכחית ב-2 (כלומר מחברים אותה לעצמה), ומוסיפים אותה

לתוצאה אם ערך הביט הוא 1. סיבוכיות זמן הריצה של אלגוריתם זה היא $O(\log n)$, והוא מאפשר להכפיל נקודות במספרים גדולים מאוד ביעילות.

תכונה חשובה של כפל נקודה שנשתמש בה בהמשך היא שלכל נקודה P וזוג מספרים a, b מתקיים:

$$b(aP) = (ba)P = (ab)P = a(bP)$$

אינטואיטיבית, נניח שמתחילים מנקודה P, עושים איתה a צעדים, ומגיעים לנקודה aP. מנקודה זו עושים b צעדים ב"גודל" a ומגיעים לנקודה $b(aP)$. באותה המידה בתרחיש אחר אפשר להתחיל מהנקודה P, לעשות איתה b צעדים ולהגיע לנקודה bP. מנקודה זו לעשות a צעדים ב"גודל" b ולהגיע לנקודה $a(bP)$. בשני התרחישים עשינו כמות זהה של ab צעדים מהנקודה P בסך הכל, ולכן בשני התרחישים הגענו לאותה הנקודה הסופית. במתמטית, כפל נקודה בקבוע הוא אסוציאטיבי.

נקודת גנרטור

אם נתחיל מנקודה P כלשהי ונחבר אותה לעצמה שוב ושוב ושוב, בכל צעד שכזה נגיע לנקודה חדשה כלשהי על העקומה. בגלל שיש מספר סופי של נקודות על העקומה, באיזשהו שלב יווצר לנו "מעגל" ונחזור שוב לנקודות שכבר הגענו אליהן קודם. ליתר דיוק, באיזשהו שלב נגיע לנקודה $-P$, בצעד הבא נגיע לנקודה $\mathcal{O}$, ובצעד שאחריו נגיע בחזרה לנקודה P, שממנה התחלנו.

לנקודה שיוצרת "מעגל" שכזה קוראים גנרטור, או איבר יוצר, כי ממנה יוצרים את שאר הנקודות שב"מעגל", ונהוג לסמן אותה באות G (ל-Generator). מספר הנקודות ב"מעגל" (כולל הנקודה $\mathcal{O}$) נקרא "הסדר של הגנרטור G", ונהוג לסמן אותו באות n. כל נקודה על העקומה יוצרת "מעגל" כלשהו. במתמטית, קבוצת הנקודות ב"מעגל" הזה היא חבורה ציקלית.

תכונה מעניינת שנובעת מכך היא שכפל נקודה G בסדר שלה n נותן לנו את נקודת האינסוף:

$$nG = \mathcal{O}$$

הבעיה הקשה

"בהינתן נקודות P ו-Q המקיימות $Q = xP$ עבור x כלשהו, קשה למצוא את x."

ובמילים, נניח שמישהו התחיל מנקודת התחלה כלשהי, לקח ממנה מספר מסוים של צעדים, והגיע לנקודה סופית. בהינתן נקודת ההתחלה והנקודה הסופית, איך אפשר לדעת כמה צעדים הוא לקח?

התשובה לכך היא לא כל כך אינטואיטיבית, כי קשה לחזות מראש מנקודה כלשהי מה יהיו הנקודות שתתקבלנה בצעדים שלוקחים ממנה. פתרון נאיבי יכול להיות להתחיל מ-P בעצמנו, להתקדם ממנה צעד אחד בכל פעם ולספור את הצעדים שאנחנו עושים, עד שנגיע ל-Q. הסיבוכיות של פתרון זה היא $O(x)$ והוא לא מעשי במידה וידוע ש-x הוא מספר גדול, למשל אם x הוא באורך של 256 ביט.

הבעיה הזאת נקראת "בעיית הלוגריתם הדיסקרטי בעקומות אליפטיות", או: Elliptic Curve Discrete Logarithm Problem (ECDLP), והיא בעיה קשה. אבל עד כמה היא קשה?

בהקשר קריפטוגרפי, נהוג למדוד "קושי של בעיות", או "חוזק של מערכת קריפטוגרפית", בעזרת מדד שנקרא Security Level. במדד זה אומרים שלבעיה יש "בטיחות של n ביטים" אם המתקפה הידועה הטובה ביותר פותרת את הבעיה ב-$O(2^n)$ צעדים.

כיום, האלגוריתם הטוב ביותר שפותר את בעיית ECDLP עושה זאת בסיבוכיות של $O(\sqrt{n})$, כש-n הוא הסדר של הנקודה P, והוא מבצע זאת על ידי מתקפת Meet In The Middle. כשבוחרים נקודה שהסדר שלה n גדול מספיק, זה לא מעשי, ומכאן החוזק של הבעיה.

לדוגמא, אם בוחרים n באורך 256 ביט, מקבלים שלבעיית ECDLP יש בטיחות של 128 ביט. לשם השוואה, כדי להשיג בטיחות זהה של 128 ביט בהצפנת RSA, שמתבססת על בעיית פירוק מספר לגורמים ראשוניים, צריך להשתמש במפתח ציבורי באורך 3072 ביט. מה שהופך את השימוש בעקומות אליפטיות ליחסית יותר יעיל מבחינה חישובית.

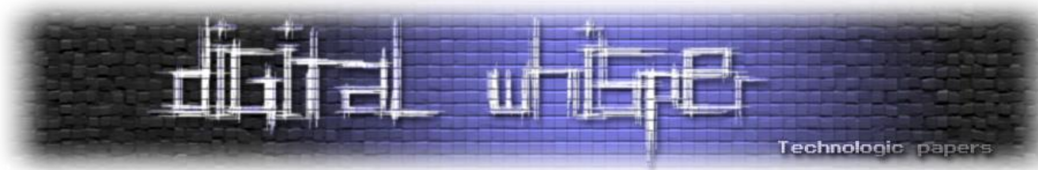
עקומות אליפטיות בהקשר של קריפטוגרפיה

אחרי כל המבוא הזה לעולם העקומות האליפטיות, נעבור לראות מה אפשר לעשות איתן בהקשר קריפטוגרפי. כידוע, מערכות קריפטוגרפיות בדרך כלל מתבססות על "בעיה קשה" שקשה לפתור. למשל RSA עם בעיית הפירוק מספר לגורמים ראשוניים שהזכרנו, או פרוטוקול Diffie-Hellman עם בעיית הלוגריתם הדיסקרטי. מערכת קריפטוגרפית שמתבססת על בעיית ECDLP בעקומה אליפטית שייכת למשפחת Elliptic Curve Cryptography, או בקיצור ECC.

שימוש ראשון לעקומות אליפטיות - הסכמה על סוד משותף

נתחיל בסיפור. דמיינו שאתם נמצאים במסיבה - חדר מלא באנשים, שבו כולם יכולים לדבר עם כולם וכולם שומעים את כולם. בחדר נמצאים גם אליס ובוב, שלא נפגשו לפני כן מעולם. בוב מוצא חן בעיני אליס, והיא מעוניינת להזמין אותו לדייט. אליס קצת מתביישת, ולכן היא רוצה להעביר לבוב את המסר הסודי הזה בלי שכל שאר אורחי המסיבה ישמעו אותה. אליס ובוב לא תיאמו ביניהם שום דבר מראש, וכל מה שאליס תגיד לבוב יישמע על ידי כל שאר האורחים במסיבה. איך אליס יכולה להעביר לבוב את המסר בלי שאף אחד אחר ישמע אותו?

אם עניתם "עקומות אליפטיות", אז אתם צודקים!



אליס תבחר עקומה אליפטית כלשהי וגנרטור שנמצא בה, ותגיד אותם לבוב. ספציפית, אליס תעביר לבוב (ולכל שאר האנשים בחדר) את שני ערכי משוואת העקומה a, b, את המודולוס p, ואת הגנרטור G. בנוסף לכך, אליס תגריל ערך כלשהו d_A בתחום $1 \le d_A \le n-1$ כאשר n הוא הסדר של G. הערך d_A נקרא המפתח הפרטי של אליס. אליס תחשב את הנקודה $A = d_A G$, שנקראת המפתח הציבורי של אליס, ותגיד אותה לבוב. באופן דומה, בוב יגריל מפתח פרטי d_B, יחשב את הנקודה $B = d_B G$, שהיא המפתח הציבורי של בוב, ויגיד אותה לאליס.

אליס תיקח את המפתח הציבורי של בוב, תכפיל את הנקודה הזו במפתח הפרטי שלה, ותגיע לנקודה שלישית $P_A = d_A B$. באותו אופן, בוב יקח את המפתח הציבורי של אליס, יכפיל את הנקודה במפתח הפרטי שלו, ויגיע לנקודה שלישית משלו $P_B = d_B A$. אם נבחן את הנקודות שאליס ובוב הגיעו אליהן בנפרד, נגלה שהם הגיעו לאותה נקודה! עובדה זו נובעת מתכונת האסוציאטיביות של כפל נקודה בקבוע שראינו קודם:

$$P_A = d_A B = d_A(d_B G) = d_B(d_A G) = d_B A = P_B$$

בסוף כל התהליך, אליס ובוב הצליחו להגיע להסכמה על נקודה כלשהי על העקומה, ובשום שלב אף אחד מהם לא העביר לשני את הנקודה הזו. המידע שכולם שמעו הוא: a, b, p, G, A, B. אדם שנמצא בחדר והאזין למידע הזה, לא יכול למצוא בעזרתו את הנקודה שאליס ובוב הסכימו עליה.

הסיבה לכך היא שאם אדם נוסף בחדר ירצה למצוא את הנקודה הזו, הוא יצטרך לדעת את המפתח הפרטי של אליס או את המפתח הפרטי של בוב, כדי להכפיל בהם את B או את A. כדי לגלות את המפתח הפרטי של אליס למשל, הוא יסתכל על $A = d_A G$, כי זה המידע היחיד שהועבר ו"מכיל בתוכו" את המפתח הפרטי של אליס. בהינתן G ו-$d_A G$, למצוא את d_A שקול ללפתור את בעיית הלוגריתם הדיסקרטי בעקומות אליפטיות, שזו כאמור בעיה קשה.

לפרוטוקול היפה הזה קוראים: Elliptic Curve Diffie-Hellman (ECDH).

שימוש בסוד המשותף להמשך התקשורת

לא סיימנו את הסיפור שלנו. אמנם אליס ובוב הסכימו על נקודה סודית משותפת, אך אליס עדיין לא הזמינה את בוב לדייט שהיא כל כך רצתה.

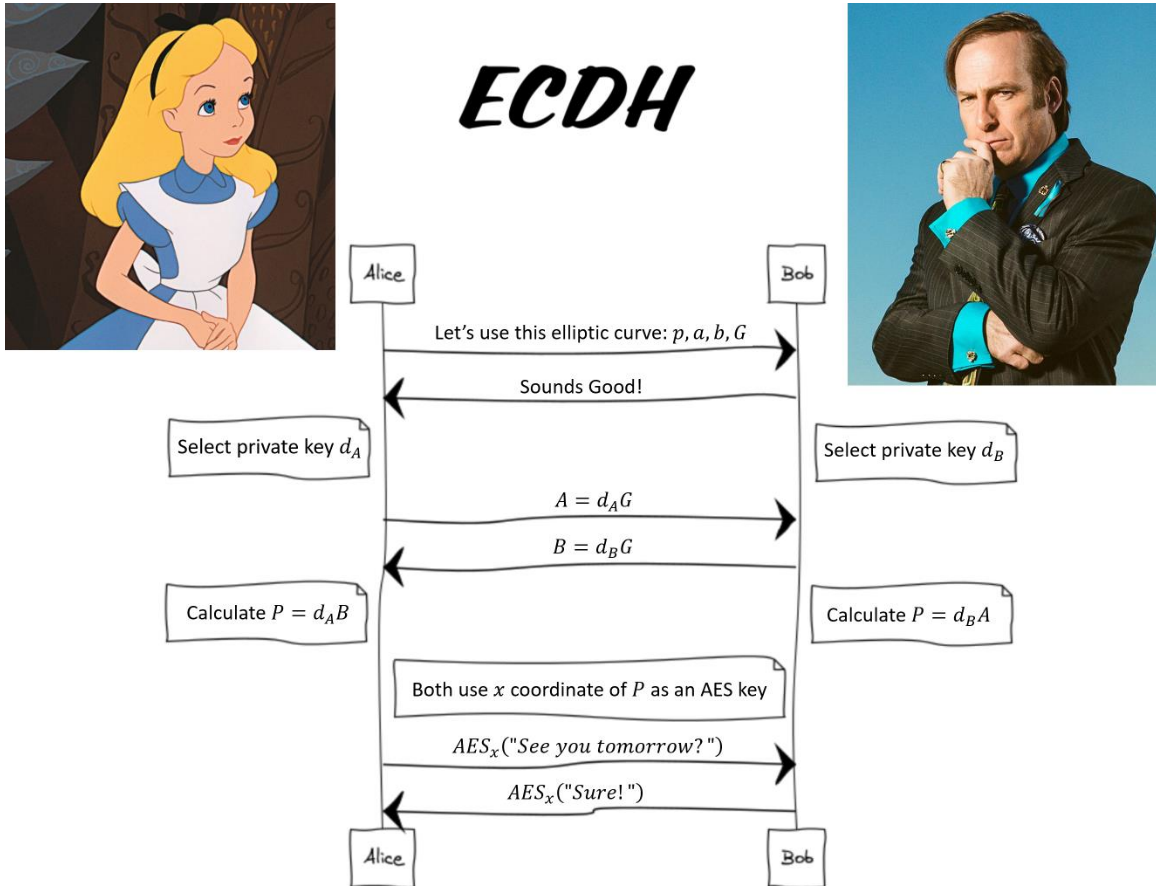
אחרי שהצדדים הסכימו ביניהם על נקודה סודית משותפת, הם יכולים להשתמש בה כמפתח הצפנה של שיטת הצפנה כלשהי, למשל AES, ומאותו הרגע לדבר ביניהם באופן מוצפן.

מעשית, נהוג לקחת את אחד מערכי ה-x או ה-y של הנקודה, ולהשתמש בו. בשביל לשמור על בטיחות, מומלץ לעשות hash על הערך שנבחר ולהשתמש רק בתוצאה כמפתח הצפנה. בפועל, לפעמים הערך שמתקבל גדול מדי למפתח הצפנה, ומשתמשים רק בחלק ממנו. למשל אם פונקציית ה-hash

שמשתמשים בה היא SHA-1, אורך הפלט שלה הוא 160 ביטים, בעוד להצפנת AES דרושים רק 128 ביטים. במקרה כזה נהוג להשתמש רק ב-128 ביטים מתוך ה-160, ולזרוק את השאר לפח.

בכל מקרה, בשלב זה אליס ובוב מסכימים על מפתח הצפנה, והם היחידים שיודעים אותו. מנקודה זו והלאה הם מתקשרים באופן מוצפן, וכל מי שמאזין להם בחדר לא יכול להבין מה הם אומרים.

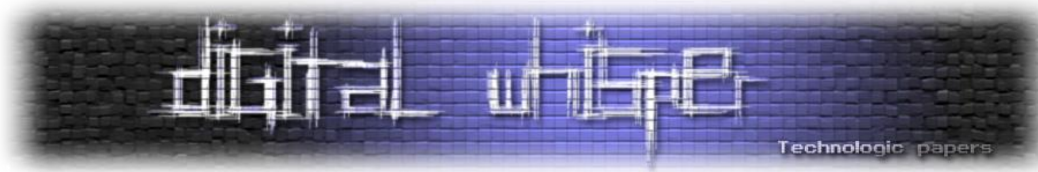
להלן תרשים של הפרוטוקול:



בעזרת המפתח המוסכם, אליס מצפינה את המסר "היי בוב, מתאים לך שנשב ביחד בבית קפה מחר בערב?", ומעבירה את הצופן לבוב. בוב מפענח את המסר בעזרת המפתח שהוא יודע גם. אליס מקווה שבוב יענה לה בחיוב, אבל זה כבר לא חלק מהפרוטוקול.

הדימיון בין Elliptic Curve Diffie-Hellman ל-Diffie-Hellman

בפרוטוקול Diffie-Hellman (DH) המפורסם, הצדדים משדרים באופן גלוי מספר ראשוני p וגנרטור g שנמצא בחבורה המתאימה לערך p. אליס מגרילה מפתח פרטי a באופן אקראי ומשדרת באופן גלוי את המפתח הציבורי שלה $A = g^a \ (mod\ p)$. באופן דומה בוב מגריל מפתח פרטי b באופן אקראי ומשדר באופן גלוי את המפתח הציבורי שלו $B = g^b \ (mod\ p)$. לאחר מכן אליס לוקחת את המפתח הציבורי של בוב, ומעלה אותו בחזקת המפתח הפרטי שלה, ובכך מחשבת את הערך $K = B^a = \left(g^b\right)^a =$



$g^{ab}\ (mod\ p)$. באותו אופן בוב מחשב את הערך $K = A^b = (g^a)^b = g^{ab}\ (mod\ p)$. בסוף התהליך אליס ובוב הצליחו להסכים על ערך K משותף, מבלי ששידרו אותו ביניהם.

תוקף שמאזין להם לא יכול למצוא את K בהינתן הערכים ששודרו p, g, A, B. כדי לעשות זאת, הוא יצטרך לפחות את אחד המפתחות הפרטיים של אליס או של בוב. כדי לחשב את המפתח של אליס למשל, הוא יצטרך למצוא את a בהינתן g ו-$g^a\ (mod\ p)$, וזאת בעיה קשה. בעיה זו נקראת בעיית הלוגריתם הדיסקרטי, Discrete Logarithm Problem (DLP).

קיים דימיון מאוד ברור בין DH, שמתבסס על DLP, לבין ECDH, שמתבסס על ECDLP (בגדול הם אותו דבר, רק עם EC לפני). בשני הפרוטוקולים, שני צדדים שמדברים ביניהם יכולים להסכים על איזשהו ערך סודי משותף, מבלי שהם תיאמו ביניהם מראש שום דבר. כל מי שיאזין להודעות בין הצדדים יהיה חשוף למידע הפומבי שהם מעבירים ביניהם, אך לא יוכל להגיע בעצמו לערך הסודי המשותף ביניהם.

שימוש שני לעקומות אליפטיות - חתימה על הודעה

נמשיך את הסיפור שלנו, נניח שאליס ובוב יצאו לדייט שלהם ובילו ערב נחמד ביחד. למחרת אליס מקבלת הודעה שאומרת "היי אליס, כאן בוב, מאוד נהנתי איתך אתמול ואשמח להיפגש איתך שוב בסוף השבוע הקרוב". אליס חושדת שלא בוב הוא זה שכתב את ההודעה, כי היא יודעת שבוב כל כך נהנה איתה אתמול, שהוא לא יחכה עד סוף השבוע להיפגש איתה, אלא ירצה להיפגש איתה כבר מחר! איך אליס תוכל לוודא שבוב הוא זה שכתב את ההודעה?

אם עניתם "עקומות אליפטיות", אז אתם צודקים שוב!

ניתן לנצל את הקושי של בעיית ECDLP גם כדי לחתום על הודעות. במהלך הדייט שלהם, בוב ואליס הסכימו על עקומה אליפטית כלשהי וגנרטור G שנמצא בה. בוב הגריל ערך כלשהו d_B, שנקרא המפתח הפרטי של בוב, וחישב את הנקודה $P_B = d_B G$, שנקראת המפתח הציבורי של בוב. בוב נתן לאליס את המפתח הציבורי שלו כדי שאיתו היא תוכל לוודא בהמשך אם הודעה שהיא תקבל אכן נחתמה על ידו.

נניח שבוב רוצה לחתום על הודעה m מסוימת. הוא יחשב את הערך $z = hash(m)$, על ידי שימוש בפונקציית hash בטוחה כלשהי, וישמור מהתוצאה כמות ביטים כאורך של n, הסדר של הגנרטור G. בוב יגריל ערך כלשהו k בתחום $1 \leq k \leq n-1$. לאחר מכן בוב יחשב את הנקודה $kG = (x_1,\ y_1)$, יקח את קואורדינטת ה-x שלה, ויחשב את $r = x_1\ \ (mod\ n)$. לבסוף בוב יחשב את הערך $s =\ k^{-1}(z + rd_B)$. החתימה על ההודעה m מוגדרת להיות זוג הערכים r ו-s שחושבו.

נניח שאליס קיבלה הודעה m מסוימת, וחתימה שלה שמורכבת מזוג הערכים r ו-s. אליס רוצה לוודא שאכן בוב הוא זה שחתם על ההודעה. אליס תחשב את הערך $z = hash(m)$ באופן זהה לבוב. לאחר מכן אליס תחשב את הערכים $u_1 = zs^{-1}$ ואת $u_2 = rs^{-1}$. לבסוף אליס תשתמש במפתח הציבורי של

בוב, P_B, ותחשב את הנקודה $u_1G + u_2P_B = (x_1, y_1)$. החתימה תיחשב חוקית אם מתקיים $r \equiv x_1 \pmod{n}$. הסיבה שהחישוב הזה נכון היא שמתקיים:

$$u_1G + u_2P_B = zs^{-1}G + rs^{-1}P_B = s^{-1}(zG + rP_B) = s^{-1}(zG + rd_BG) =$$

$$= s^{-1}(z + rd_B)G = k(z + rd_B)^{-1}(z + rd_B)G = kG$$

אם החתימה חוקית, קואורדינטת ה-x של הנקודה הזו אכן אמורה להיות r כפי שערך זה הוגדר להיות בחתימה על ההודעה. יש לציין שסדר הגנרטור G, שמסומן באות n, צריך להיות מספר ראשוני, וזאת כדי שאכן יהיה אפשר לחשב את המספרים ההופכיים בנוסחאות החתימה והאימות.

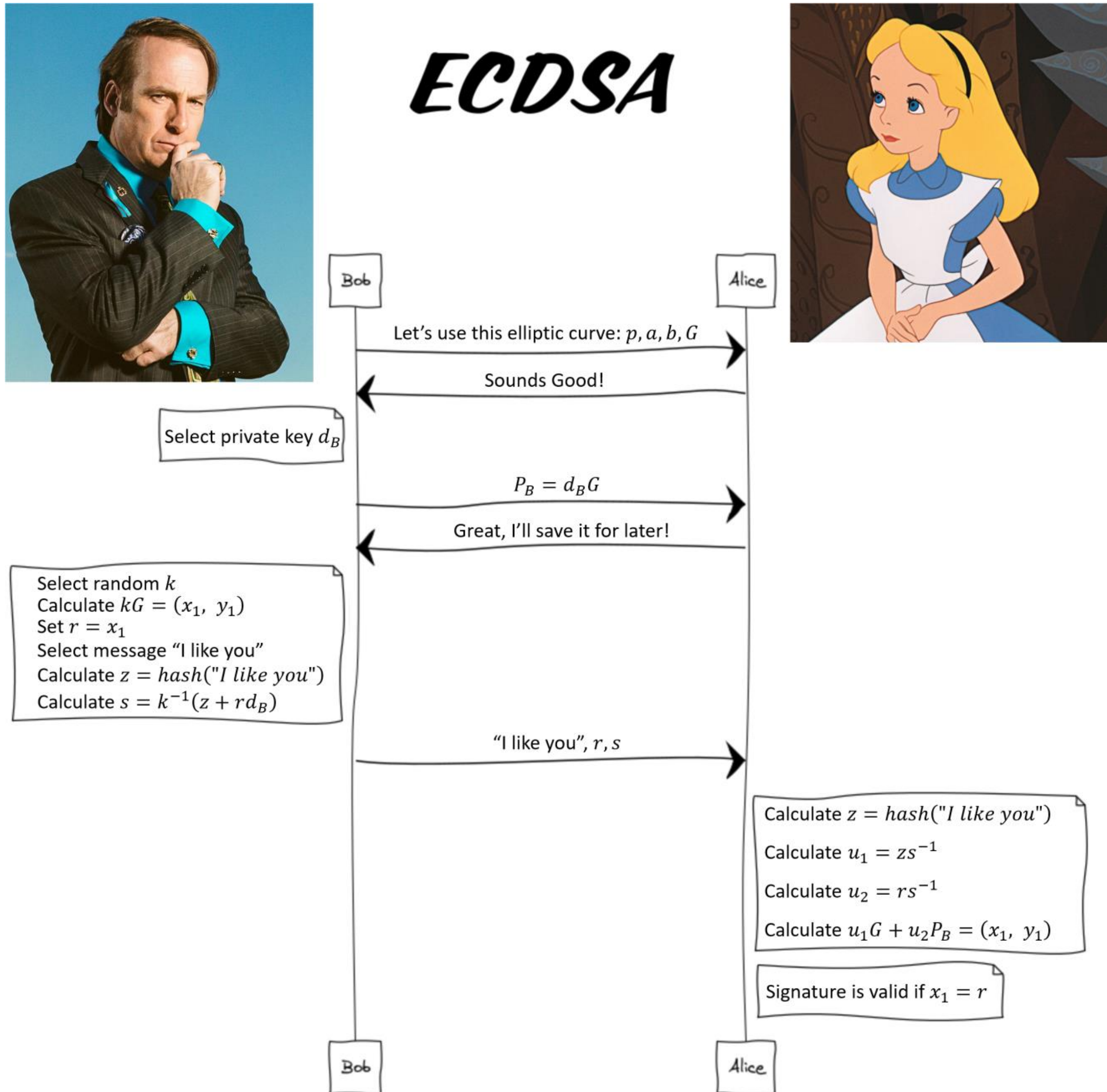
ניתן לראות שרק מי שמחזיק את המפתח הפרטי d_B יכול ליצור חתימה שתהיה חוקית עבור המפתח הציבורי P_B. תוקף שאין לו את הערך d_B, לא יכול לחשב ערך s שיתאים לחישוב שבאימות החתימה עם P_B. אם התוקף ירצה ליצור חתימה שמתאימה להודעה כלשהי, הוא יצטרך לפתור את בעיית ECDLP, כלומר למצוא את המפתח הפרטי d_B בהינתן G ו-$P_B = d_BG$, וזו בעיה קשה.

פרוטוקול החתימה הזה נקרא Elliptic Curve Digital Signature Algorithm, או בקיצור ECDSA. הפרוטוקול מבטיח שההודעות שנחתמו לא שונו או זויפו, ובנוסף לכך מבטיח שמי שחתם על ההודעה לא יכול להתכחש לכך שהוא יצר אותה.

בניגוד לפרוטוקול ECDH, שבו הצדדים לא היו צריכים לתאם ביניהם מראש שום דבר, בפרוטוקול ECDSA הצדדים חייבים להסכים מראש על מפתח ציבורי. רק אחרי שכל צד יודע בוודאות שהמפתח הציבורי שהוא מחזיק בידו אכן מתאים לאדם שהוא רוצה לתקשר איתו, ניתן להשתמש בפרוטוקול. אחרת, אין שום משמעות לאימות החתימה עם המפתח הציבורי שיש ברשות כל צד.

בחזרה לסיפור שלנו. אליס יודעת בוודאות שהמפתח הציבורי P_B שברשותה אכן שייך לבוב, כי בוב מסר לה אותו במפורש בדייט שלהם. אליס מנסה לאמת בעזרתו את ההודעה ומגלה שאין התאמה. כמובן! מישהו אחר כתב את ההודעה וחתם עליה, בדיוק כפי שאליס חשדה.

להלן תרשים של הפרוטוקול:



הדימיון בין ECDSA ל-ElGamal

בפרוטוקול ElGamal לחתימה על הודעות, הצדדים מסכימים על מספר ראשוני גדול p ומספר גנרטור g. הצד שמעוניין לחתום על הודעות מגריל ערך כלשהו d בתחום $1 \leq d < p - 1$, שנקרא המפתח הפרטי, מחשב את הערך $y = g^d \ (mod\ p)$, שנקרא המפתח הציבורי, ומפרסם אותו.

כדי לחתום על הודעה m מסוימת, הוא יחשב את הערך $z = hash(m)$, ויגריל ערך כלשהו k בתחום $1 \leq k < p - 1$, הזר ל- $(p - 1)$. הוא יחשב את הערך $r = g^k \ (mod\ p)$, ואת הערך $s = k^{-1}(z - dr) \left(mod\ (p - 1)\right)$. החתימה על ההודעה m מוגדרת להיות זוג הערכים r ו-s שחושבו.

צד שקיבל הודעה m מסוימת, וחתימה שלה שמורכבת מזוג הערכים r ו-s, משתמש במפתח הציבורי y כדי לאמת את החתימה על ידי חישוב הערכים $u_1 = r^s y^r$ ו- $u_2 = g^z$. החתימה תיחשב חוקית אם $u_1 = u_2$. זאת כי לפי הגדרת s מתקיים $s = k^{-1}(z - dr)$, ולכן $ks = z - dr$, ומכאן $z = ks + dr$. אז:

$$u_2 = g^z = g^{ks+dr} = g^{ks} g^{dr} = \left(g^k\right)^s \left(g^d\right)^r = r^s y^r = u_1$$

תוקף לא יכול ליצור חתימה חוקית עבור המפתח הציבורי y ללא ידיעת המפתח הפרטי d. כדי להשיג את המפתח הפרטי בהינתן המפתח הציבורי, התוקף יצטרך לפתור את בעיית DLP, וזו בעיה קשה.

גם כאן יש דימיון ברור בין ECDSA, שמתבסס על ECDLP, ל-ElGamal, שמתבסס על DLP. בשני המקרים הצדדים צריכים לתאם ביניהם מראש מפתח ציבורי, ונדרש להגריל ערך k בצורה אקראית בכל פעם שרוצים לחתום על הודעה חדשה. כמו כן, בשני המקרים תוקף שמאזין להודעות בין הצדדים לא יכול להסיק מידע שימושי שיאפשר לו לזייף חתימות בעצמו.

תקיפת מערכות ECC

ראינו איך אפשר להשתמש בעקומות אליפטיות במערכות קריפטוגרפיות לצורך הסכמה על ערך סודי ולצורך חתימה על הודעות. כמו כל דבר בחיים, כשבאים לממש משהו בפועל, דברים לא תמיד עובדים כמתוכנן. בשאר המאמר אני אציג דרכים שונות לתקוף מערכות קריפטוגרפיות מבוססות ECC שנעשה בהן שימוש לא נכון על ידי המשתמש, או שהן מומשו בצורה לא בטיחותית.

באופן טבעי, חילקתי את החלק הזה למתקפות על ECDH ולמתקפות על ECDSA. בשני המקרים נגיד ש"הצלחנו" במתקפה אם נמצא את המפתח הפרטי של אחד הצדדים, ושם נעצור. במקרה של ECDH זה מספיק כי מהמפתח הפרטי אפשר להגיע לערך הסודי המשותף ולכל המידע שהוצפן בעזרתו בהמשך. במקרה של ECDSA זה מספיק כי מהמפתח הפרטי אפשר לחתום על הודעות כרצוננו.

SageMath

SageMath היא תוכנה מתמטית חינמית ומבוססת קוד-פתוח. ניתן לכתוב בה בסינטקס כמעט זהה ל-Python, וניתן גם להשתמש בה כספריה ב-Python. הספריה הזו ממשת בשבילנו פונקציות שרלוונטיות לעקומות אליפטיות ולכן היא שימושית מאוד לצורך החישובים שאנחנו צריכים לעשות בהקשר של ECC. במסגרת מאמר זה אני מספק קטעי קוד שכתובים בספריה הזו. אני מצאתי שהכי פשוט להתקין אותה על מערכת הפעלה Ubuntu, ספציפית בגרסא 22.04. כדי להתקין אותה, יש להריץ את הפקודה:

```
sudo apt install sagemath
```

כדי להריץ קובץ שמכיל קוד, יש לשמור את הקובץ עם סיומת sage. ולהריץ את הפקודה:

```
sage file.sage
```

בנוסף ניתן להשתמש ב-Interpreter בצורה דינמית, באופן דומה ל-Interpreter של Python, על ידי הרצת הפקודה:

```
sage
```

ניתן גם ליצור קבצי py. שמבצעים בהם import לספריה sage.all, ולהריץ אותם עם הפקודה:

```
python3 file.py
```

יש לשים לב שכשמריצים קובץ עם הפקודה:

```
sage
```

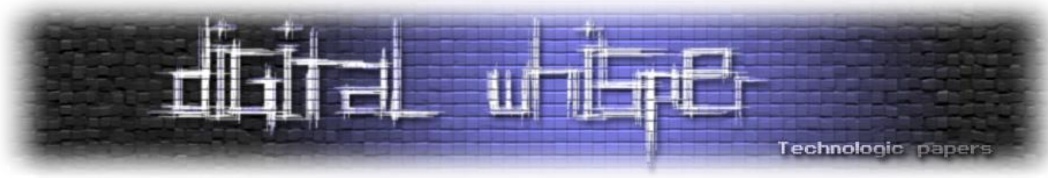
הסימון ^ מסמן חזקה, בעוד שכשמריצים אותו עם:

```
python3
```

הסימון הזה מתפרש כ-xor.

במאמר זה אני משתמש בעיקר בפונקציות הבאות ב-SageMath:

- E.gens() - מציאת גנרטורים בעקומה E
- G.order() - חישוב הסדר של הגנרטור G
- n*G - הכפלת הגנרטור G במספר n
- n.factor() - מפרק את המספר n לגורמים הראשוניים שלו - הפונקציה מחזירה רשימה של זוגות $(p,\ e)$ כך ש-p הוא גורם ראשוני, ו-e הוא האקספוננט שלו, כלומר כמות הפעמים ש-p מופיע בפירוק של n
- crt - פתרון מערכת משוואות של משפט השאריות הסיני



מתקפות ידועות על ECDH

בחירת מודולוס קטן מדי

אולי השימוש השגוי ל-ECDH שאותו הכי קל לתקוף הוא בחירה של גנרטור מסדר n קטן מדי. כאמור, ניתן לפתור את בעיית ECDLP בסיבוכיות של $O(\sqrt{n})$. כש-n קטן מדי, למשל באורך 32 ביט, אז זה נהיה מעשי (Feasible) לפתור את הבעיה הזאת. יש מספר אלגוריתמים שפותרים את הבעיה, ביניהם Baby-Step Giant-Step, Pollard's Rho, ו-Pollard's Lambda. ניתן להריץ את האלגוריתמים האלה כקופסא שחורה בעזרת SageMath, על ידי שימוש בפונקציה discrete_log:

```
import random
p = random_prime(2^32)
a = random.randrange(p)
b = random.randrange(p)
E = EllipticCurve(GF(p), [a,b])
G = E.gens()[0]
n = G.order()

private_key = random.randrange(n)
A = private_key * G

found_key = G.discrete_log(A)
assert found_key * G == A
assert private_key == found_key
print("success!")
```

בקטע קוד זה אנחנו בוחרים פרמטרים לעקומה בצורה אקראית תחת המגבלה ש-p באורך 32 ביט. מגבלה זו מבטיחה לנו שכמות הנקודות על העקומה היא $O(2^{32})$ ולכן הסדר של כל נקודה בה הוא לכל היותר $O(2^{32})$ גם כן. לאחר מכן יוצרים את העקומה, בוחרים גנרטור כלשהו בה, מגרילים מפתח פרטי בצורה אקראית, ומחשבים את המפתח הציבורי. לבסוף, מתוך הגנרטור והמפתח הציבורי, מחשבים את הלוגריתם הדיסקרטי כדי למצוא את המפתח הפרטי, ומוודאים שהמפתח שנמצא הוא אכן נכון. לקוד זה לוקח לכל היותר מספר שניות למצוא את המפתח הפרטי.

סדר הגנרטור הוא מספר חלק (Smooth)

כזכור, סדר של גנרטור מוגדר להיות מספר הנקודות ב"מעגל" שנוצר כשמחברים את נקודת הגנרטור עם עצמה שוב ושוב, ונהוג לסמן אותו ב-n. אם n הוא מספר שניתן לפרק להרבה גורמים ראשוניים, אז ניתן לפתור את ECDLP ביעילות. מספר כזה נקרא מספר חלק (Smooth Number), ולצורך העניין זהו מספר שמתפרק למספיק גורמים ראשוניים שכל אחד מהם קטן מספיק כדי שהמתקפה שלנו תעבוד. ההגדרה הרשמית למספר חלק היא קצת שונה ולא רלוונטית בשבילנו.

אינטואיטיבית, עושים זאת על ידי "תקיפה" של כל אחד מהגורמים הראשוניים בנפרד. בהינתן נקודה G שיוצרת "מעגל" גדול מאוד, ונקודה כלשהי P ב"מעגל" כך ש $P = kG$. ניתן לפרק את ה"מעגל" הגדול לכמה "מעגלים" קטנים, שכל אחד בגודל של גורם ראשוני אחד של n. בכל "מעגל" קטן ניתן להתאים ל- G ו-P נקודות G' ו-P' שנמצאות ב"מעגל" הקטן ומקיימות $P' = k'G'$. בגלל שה"מעגל" קטן, ניתן בקלות יחסית לפתור את הבעיה ולמצוא את k'. לבסוף, אפשר לאחד את כל ה-k'-ים הקטנים שמצאנו לכדי k המבוקש ב"מעגל" המקורי.

האלגוריתם שמבצע את מה שתיארתי נקרא Pohlig-Hellman Algorithm. סיבוכיות זמן הריצה שלו היא $O(\sqrt{p_{max}})$ כש-p_{max} הוא הגורם הראשוני הגדול ביותר בפירוק של n. זה גם הגיוני, כי החלק הכי "כבד" באלגוריתם הוא פתרון בעיית ECDLP ב"מעגל" הגדול ביותר מבין ה"מעגלים" הקטנים. לשם הדוגמא, יכול להיות ש-n הוא מספר באורך 128 ביט, והוא מתפרק לגורמים ראשוניים שהגדול בהם הוא בכלל באורך 30 ביט. האלגוריתם מקטין את סיבוכיות פתרון הבעיה מסדר גודל של 2^{64} פעולות לסדר גודל של 2^{15} פעולות, כלומר הופך את פתרון הבעיה מלא-מעשי למעשי (Feasible).

למזלנו הפונקציה discrete_log של SageMath מבצעת את האלגוריתם הזה במימוש שלה, ולכן כדי להריץ את המתקפה ניתן פשוט לקרוא לפונקציה:

```
p = 18374030529116688990089487930285841 1333
a = 13
b = 37
E = EllipticCurve(GF(p), [a,b])
G = E(123764810000715262449972298016641419881,
144640915410606177233842123838934486566)
n = G.order()
print("number of bits in n:", n.nbits())
print("n's factors:", n.factor())
print("number of bits in n's greatest factor:", n.factor()[-1][0].nbits())

import random
private_key = random.randrange(n)
A = private_key * G
print("Calculating discrete_log...")
found_key = G.discrete_log(A)
assert found_key * G == A
assert private_key == found_key
print("success!")
```

בקטע קוד זה מגדירים עקומה אליפטית וגנרטור בה, ומדפיסים את הפירוק שלו לגורמים ראשוניים. מה שמודפס למסך הוא:

```
number of bits in n: 128
n's factors: 2 * 3 * 13 * 101 * 211 * 21141581 * 38581057 * 60652309 *
2234328781
number of bits in n's greatest factor: 32
Calculating discrete_log...
success!
```

ניתן לראות שאמנם הסדר של הגנרטור הוא באורך 128 ביט, אך הוא מתפרק לגורמים ראשוניים כך שהגורם הראשוני הגדול ביותר שלו הוא באורך 32 ביט.

לאחר מכן בדיוק כמו במתקפה הקודמת - בוחרים מפתח פרטי אקראי, מחשבים ממנו מפתח ציבורי, ואז בהינתן הגנרטור והמפתח הציבורי מחשבים בחזרה את המפתח הפרטי ומוודאים שהוא נכון.

אמנם סיימנו, אך לא ראינו איך המעגלים ה"קטנים" מוגדרים, איך מתאימים ל-G ו-P נקודות G' ו-P', ואיך משלבים את הפתרונות הקטנים לפתרון גדול. אני אנסה להסביר זאת כאן בצורה אינטואיטיבית, כי מתבססים על כך גם במתקפה הבאה.

נניח שיש לנו "מעגל" מסדר $3x5x7 = 105$, והגנרטור שלו הוא G. נגדיר נקודה $G' = (5x7)G = 35G$, ונסתכל על ה"מעגל" שנוצר ממנה. אם נתקדם מ-G' "צעד" אחד, כלומר נחבר את G' לעצמה, זה יהיה שקול ללהתקדם 35 צעדים מהנקודה $35G$ ב"מעגל" המקורי, ונגיע לנקודה $2G' = 70G$. אם נתקדם מנקודה זו עוד "צעד", נגיע לנקודה $3G' = 105G = \mathcal{O}$, ואם נתקדם ממנה עוד "צעד", נגיע לנקודה $4G' = 35G = G'$, כלומר בחזרה לנקודת ההתחלה. ה"מעגל" שנוצר מ-G' הוא מסדר 3, וזה לא במקרה, כי על "מעגל" מסדר 105 אפשר לעשות בדיוק 3 "צעדים" בגודל 35. באופן דומה ניתן ליצור "מעגל" מסדר 5 על ידי הגדרת הנקודה $G' = (3x7)G = 21G$, ומעגל מסדר 7 על ידי הגדרת $G' = (3x5)G = 15G$.

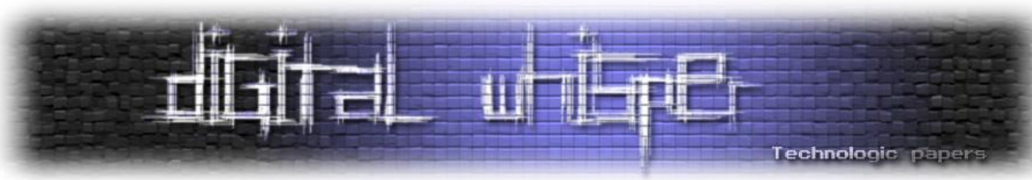
כשמסתכלים על זה בכיוון ההפוך זה נהיה מעניין יותר. נניח שב"מעגל" המקורי לקחנו n צעדים מהנקודה G והגענו לנקודה nG. אם גם ב"מעגל" הקטן ניקח n צעדים מהנקודה G', נגיע לנקודה $n'G'$ כך ש $n \equiv n' \ (mod\ 3)$. ולמה זה מעניין? כי הסדר של G' הוא הרבה יותר קטן מהסדר של G ולכן בהינתן G' ו-$n'G'$, אנחנו יכולים למצוא בקלות יחסית את n'. אם נעשה את זה, ונעשה את זה גם לשני הגורמים הראשוניים האחרים של סדר ה"מעגל" שלנו, שהם 5 ו-7, יהיו לנו את הערכים הבאים:

$$n \equiv \ n'_1 \ (mod\ 3)$$
$$n \equiv \ n'_2 \ (mod\ 5)$$
$$n \equiv \ n'_3 \ (mod\ 7)$$

מתוך שלושת הערכים האלה, ניתן בקלות למצוא את n על ידי שימוש במשפט השאריות הסיני, ובכך בעצם לפתור את הבעיה המקורית.

סדר הגנרטור הוא מספר כמעט-חלק, והמפתח הפרטי קטן

נניח שבאופן דומה למתקפה הקודמת, היינו מקבלים עקומה שהסדר של הגנרטור בה מתפרק לגורמים ראשוניים. אבל הפעם, הגורם הראשוני הגדול ביותר הוא גדול מדי כדי שיהיה מעשי לפתור את ECDLP. למשל, אם סדר הגנרטור הוא באורך 256 ביט, אך הגורם הראשוני הגדול ביותר הוא באורך 128 ביט. האלגוריתם Pohlig-Hellman Algorithm יצטרך סדר גודל של 2^{64} פעולות כדי למצוא את המפתח הפרטי, שזה לא מעשי.



אם אנחנו יודעים שהמפתח הפרטי שהשתמשו בו הוא יחסית קטן, ניתן בכל זאת למצוא אותו ביעילות. נניח שהמפתח הפרטי באורך 64 ביט (במקום 256 ביט). כשיוצרים את המפתח הציבורי, מכפילים את הגנרטור במפתח הפרטי ומקבלים נקודה כלשהי ב"מעגל" שהגנרטור יוצר. אמנם ה"מעגל" בגודל של בערך 2^{256} נקודות, אך הנקודה הזאת "תיפול" איפושהו ב-2^{64} הנקודות ה"ראשונות". אין "אינטרקציה" בין המפתח הפרטי לנקודות ב"מעגל" שמתאימות לערכים גדולים יותר.

ניתן להריץ את Pohlig-Hellman Algorithm, אבל "לוותר" בו על ה"מעגלים" הגדולים מדי, בתנאי שמכפלת סדרי ה"מעגלים" הנותרים תהיה באורך של לפחות אורך המפתח הפרטי. אם נמצא מספיק גורמים ראשוניים קטנים, שהמכפלה שלהם לפחות באורך 64 ביט, אז ה"מעגלים" המתאימים להם יהיו מספיקים כדי לבצע את אותה המתקפה שראינו קודם.

אם קודם היו לנו חיים קלים מבחינת כתיבת קוד, הפעם נצטרך לממש דברים בעצמנו, כי הפונקציה discrete_log של SageMath לא יודעת שצריך "לוותר" על חלק מהגורמים ראשוניים. קטע הקוד הבא עושה את זה:

```
p = 88664572752015126127869404674421545790506871948117527783533589813159111825511
a = 13
b = 37
E = EllipticCurve(GF(p), [a,b])
G = E(19374976316789648652022260955836934561553454311144967863145605756652014623129,
68630819472054489323664324766002023315775509214344811025345735680440707888471)
n = G.order()

print("Number of bits in n:", n.nbits())
factors = n.factor()
print("n's factors:", factors)

PRIVATE_KEY_BIT_SIZE = 64
import random
private_key = random.randrange(2^PRIVATE_KEY_BIT_SIZE)
P = private_key * G

print("We know that the private key is", PRIVATE_KEY_BIT_SIZE, "bits long")
print("Lets find which of the factors of G's order are relevant for finding the private
key")
# find factors needed such that the order is greater than the secret key size
count_factors_needed = 0
new_order = 1
for p, e in factors:
    new_order *= p^e
    count_factors_needed += 1
    if new_order.nbits() >= PRIVATE_KEY_BIT_SIZE:
        print("Found enough factors! The rest are not needed")
        break
factors = factors[:count_factors_needed]
print("Considering these factors:", factors)

print("Calculating discrete log for each quotient group...")
subsolutions = []
subgroup = []
for p, e in factors:
    quotient_n = (n // p ^ e)
    G0 = quotient_n * G # G0's order is p^e
    P0 = quotient_n * P
    k = G0.discrete_log(P0)
```

```
    subsolutions.append(k)
    subgroup.append(p ^ e) # k the order of G0

print("Running CRT...")
found_key = crt(subsolutions, subgroup)
assert found_key * G == P
assert private_key == found_key
print("success!")
```

בקטע קוד זה מגדירים עקומה אליפטית וגנרטור בה, ומדפיסים את הפירוק שלו לגורמים ראשוניים. מה שמודפס למסך הוא:

```
Number of bits in n: 256
n's factors: 2 * 3 * 29 * 2699 * 28751 * 831913766251 * 92996710252298530263979
* 848787825227814786043072304642711
```

הסדר של הגנרטור הוא באורך 256 ביט, והוא מתפרק למספר גורמים ראשוניים, כך ששני הגדולים בהם הם באורכים 77 ביט ו-107 ביט. הם מספיק גדולים כדי שלא יהיה מעשי לפתור בהם את ECDLP. לאחר מכן מגרילים מפתח פרטי באורך 64 ביט בצורה אקראית, ומחשבים מפתח ציבורי. בשלב הבא "אוספים" מספיק גורמים ראשוניים עד שמקבלים סדר שאורכו לפחות 64 ביט. מה שמודפס למסך הוא:

```
We know that the private key is 64 bits long
Lets find which of the factors of G's order are relevant for finding the private key
Found enough factors! The rest are not needed
Considering these factors: [(2, 1), (3, 1), (29, 1), (2699, 1), (28751, 1),
(831913766251, 1)]
```

ניתן לראות ששני הגורמים הגדולים מיותרים, והגורם הגדול ביותר שנשארנו איתו הוא באורך 40 ביט. בשלב הבא, לכל אחד מהגורמים שנשארנו איתם, מחשבים נקודות G' ו-P' כפי שהסברתי קודם, ועבור כל אחד מהם פותרים את ECDLP. את התוצאות ואת הגורמים הראשוניים שומרים ברשימות subsolutions ו-subgroups בהתאמה. לבסוף, מאחדים את כל התוצאות בעזרת משפט השאריות הסיני לכדי המפתח הפרטי, ומוודאים שהוא אכן נכון.

חוסר בדיקה שנקודה נמצאת על העקומה

אם נחזור להגדרת חיבור בין נקודות בעקומות אליפטיות, ניתקל בתכונה מעניינת והיא שבפעולת חיבור אין שימוש בערך b, אלא רק בערכים a ו-p. הדבר גורם לכך שפעולת חיבור בין נקודות שנמצאות על גבי עקומה אחת יכולה להיות בעלת משמעות גם עבור עקומה אחרת, ששונה ממנה בערך b בלבד. הדבר הזה נכון כמובן גם עבור כפל נקודה במספר. אם המשתמש לא מוודא שהנקודה שהוא מקבל מהצד השני בתור מפתח ציבורי נמצאת על העקומה שלו, אז הוא חושף את עצמו למתקפה מסוג Invalid Curve Attack.

נניח ששני צדדים הסכימו על עקומה אליפטית $E1$ כלשהי. תוקף יכול ליצור עקומה $E2$, שיש לה ערכי a ו-p זהים ל-$E1$ אך ערך b שונה. בעקומה $E2$ התוקף יבחר נקודה P שהסדר שלה קטן, למשל 3. כמובן

שהנקודה P לא תהיה על $E1$, כי היא מקיימת משוואה עם ערך b שונה מזה שב-$E1$. התוקף ישלח את הנקודה P בתור המפתח הציבורי שלו אל המשתמש. נניח שהמשתמש לא טורח לוודא שהנקודה שהוא קיבל אכן נמצאת על העקומה $E1$ שהצדדים הסכימו עליה. המשתמש יקח את המפתח הציבורי שהוא קיבל מהתוקף, יכפיל אותו במפתח הפרטי שלו, ויגיע לנקודה שהיא אמורה להיות הנקודה הסודית המשותפת כפי שראינו בהגדרת הפרוטוקול ECDH. מבחינת המשתמש, הוא יחשב את פעולת הכפל על גבי העקומה $E1$. אך בגלל שהנקודה P כלל לא נמצאת עליה, אלא על $E2$, המשתמש בעצם יחשב את פעולת הכפל על העקומה $E2$. בהמשך, המשתמש יסתמך על הנקודה הסודית המשותפת כדי להמשיך את התקשורת עם התוקף. נניח שהצדדים משתמשים בקואורדינטת ה-x של הנקודה בתור מפתח הצפנה של AES. במקרה זה, המשתמש יצפין איזושהי הודעה שהוא רוצה להעביר לתוקף וישלח לו את ההודעה המוצפנת.

מכיוון שהסדר של P הוא 3, יש רק 3 אפשרויות לנקודה המשותפת שהמשתמש יחשב. התוקף יעבור על 3 הנקודות האפשריות, וימצא מי מביניהן מתאימה למפתח שמפענח בהצלחה את ההודעה המוצפנת שהגיעה מהמשתמש. בהינתן הנקודה הזו ונקודת ההתחלה P, התוקף יכול להסיק את תוצאת שארית החלוקה של המפתח הפרטי במספר 3. התוקף יכול לשלוח למשתמש נקודות P זדוניות נוספות, עם סדרים הולכים וגדלים, למשל 5, 7, וכך הלאה. באופן זה התוקף יכול לאסוף מספיק ערכים שמייצגים שאריות חלוקה של המפתח הפרטי במספרים קטנים. לבסוף התוקף ישתמש במשפט השאריות הסיני כדי לחשב את המפתח הפרטי של המשתמש, באופן זהה למה שראינו במתקפה הקודמת.

הנה הסבר יותר אינטואיטיבי: תוקף יכול לספק למשתמש נקודה על "מעגל" קטן מאוד, למשל באורך 2. המשתמש יתקדם ב"מעגל" הזה מספר צעדים כלשהו ויגיע לנקודת היעד. התוקף יודע את נקודת היעד של המשתמש, שיכולה להיות אחת מבין 2 אפשרויות בסך הכל. לכן התוקף יכול לדעת אם המשתמש התקדם מספר זוגי או אי-זוגי של צעדים על גבי המעגל. התוקף יכול לספק למשתמש נקודות על "מעגלים" נוספים - באורכים 3, 5, 7, וכך הלאה. עד שיהיו לתוקף מספיק גורמים שכאלה, שכל אחד מהם מכיל מידע מועט על מספר הצעדים שהמשתמש התקדם. לבסוף התוקף יכול לשלב את כל הערכים האלה לכדי מספר הצעדים המדויק שהמשתמש התקדם, שזהו המפתח הפרטי שלו.

קטע הקוד הבא מדגים את המתקפה:

```
from ecdsa.ecdsa import generator_128r1, curve_128r1
from Crypto.Util.number import long_to_bytes
from Crypto.Util.Padding import pad, unpad
from Crypto.Cipher import AES
import random

# Select a curve and generator
curve = curve_128r1
G = generator_128r1
n = G.order()
p = curve.p()
a = curve.a()

# This is the private key of the other side, we don't know it and don't use it!
private_key = random.randrange(n)
```

```python
# Both sides encrypt and decrypt data the same way
# key is the shared point's x coordinate, IV is point's y coordinate
def encrypt_data(shared_point, message):
    if shared_point.is_zero():
        x, y = 0, 0
    else:
        x, y = shared_point.xy()
    key = long_to_bytes(int(x)).rjust(16, b"\x00")
    iv = long_to_bytes(int(y)).rjust(16, b"\x00")
    cipher = AES.new(key, AES.MODE_CBC, iv)

    message = pad(message.encode(), 16)
    return cipher.encrypt(message)

def decrypt_data(shared_point, enc_message):
    if shared_point.is_zero():
        x, y = 0, 0
    else:
        x, y = shared_point.xy()
    key = long_to_bytes(int(x)).rjust(16, b"\x00")
    iv = long_to_bytes(int(y)).rjust(16, b"\x00")
    cipher = AES.new(key, AES.MODE_CBC, iv)

    decrypted = cipher.decrypt(enc_message)
    return unpad(decrypted, 16)

def ECDH(A):
    # Send our public key to the other side
    # Have them reach the shared point and
    # Send us an encrypted message using the shared point as key

    # This part takes place remotely and is unknown to the attacker
    shared_point = private_key * A
    message = "Inconceivable!"
    return encrypt_data(shared_point, message)

def brute_force_encrypted_message(A, encrypted_message, max_order):
    # Returns n such that n*A matches the key used to encrypt the message
    for i in range(1, max_order):
        shared_point = i * A
        try:
            # If both padding is correct and all characters are ascii
            # Then it is probably the correct encryption key
            decrypted = decrypt_data(shared_point, encrypted_message)
            decrypted = decrypted.decode()
            return i
        except:
            continue
    raise Exception("Did not find a value for one of the encrypted messages")

def find_curves_with_small_subgroup(p, a, max_order):
    # Yield tuples of (order, point) such that the point is
    # on a curve with the same a & p values, but different b
    # and the point's order is <= max_order
    orders_found = set()
    b = 0
    while True:
        b += 1
        if b == p:
            # Ran out of b values
            break
        if (4*a^3 + 27*b^2) % p == 0:
```

```
            # Curve is singular
            continue

        E = EllipticCurve(GF(p), [a, b])
        for _ in range(100):
            R = E.random_point()
            n = R.order()
            for f, e in n.factor():
                if f in orders_found:
                    continue
                if f > max_order:
                    break

                # Create a point with order f
                orders_found.add(f)
                P = (n // f) * R
                assert P.order() == f
                yield (f, P)

subsolutions = []
subgroup = []
max_order = 10000
upto = 1
for order, A in find_curves_with_small_subgroup(p, a, max_order):
    upto *= order
    print("Found point with order", order, "so now can find keys of size up to", upto)

    # Send this point as our public key and get an encrypted message from other side
    encrypted_message = ECDH(A)

    # Find the value n such that: private_key = n (mod order)
    key_mod_order = brute_force_encrypted_message(A, encrypted_message, max_order)

    # Save result to be used in CRT later
    subsolutions.append(key_mod_order)
    subgroup.append(order)

    # Found enough values to calculate private key
    if upto >= n:
        break

print("Found enough values! Running CRT...")
found_key = crt(subsolutions, subgroup)
print("Found private key", found_key)
assert private_key == found_key
print("success!")
```

בקטע קוד זה בוחרים עקומה וגנרטור בה, המשתמש בוחר מפתח פרטי בצורה אקראית ומשתמש בו לצורך כל השימושים בפרוטוקול ECDH. הפונקציה find_curves_with_small_subgroup מוצאת זוגות של נקודות וסדר, כך שהסדר של כל נקודה הוא יחסית קטן, והנקודה נמצאת על עקומה כלשהי ששונה מהעקומה המקורית בערך b בלבד. הקוד מייצר זוגות כאלה עד שמספיק זוגות נמצאו. לכל זוג, שולחים למשתמש את המפתח הציבורי ומקבלים ממנו הודעה מוצפנת.

מבצעים Brute Force על ההודעה המוצפנת כדי למצוא את ערך המפתח הפרטי של המשתמש, מודולו הסדר הנוכחי. שומרים את כל התוצאות האלה, ולבסוף משתמשים במשפט השאריות הסיני כדי לחשב את המפתח הפרטי של המשתמש ומוודאים שהוא נכון. במקרה זה הצדדים הסכימו ביניהם שהתקשורת

תיעשה ב-AES עם מפתח ההצפנה שהוא קואורדינטת ה-x של הנקודה המשותפת, ו-IV שהוא קואורדינטת ה-y של הנקודה המשותפת.

מה שמודפס למסך הוא למשל:

```
Found point with order 2 so now can find keys of size up to 2
Found point with order 197 so now can find keys of size up to 394
Found point with order 4729 so now can find keys of size up to 1863226
Found point with order 5 so now can find keys of size up to 9316130
Found point with order 7 so now can find keys of size up to 65212910
Found point with order 251 so now can find keys of size up to 16368440410
Found point with order 3853 so now can find keys of size up to 63067600899730
Found point with order 3 so now can find keys of size up to 189202802699190
Found point with order 31 so now can find keys of size up to 5865286883674890
Found point with order 41 so now can find keys of size up to 240476762230670490
Found point with order 53 so now can find keys of size up to 12745268398225535970
Found point with order 109 so now can find keys of size up to 1389234255406583420730
Found point with order 59 so now can find keys of size up to 81964821068988421823070
Found point with order 67 so now can find keys of size up to 5491643011622224262145690
Found point with order 139 so now can find keys of size up to 763338378615489172438250910
Found point with order 47 so now can find keys of size up to 35876903794927991104597792770
Found point with order 683 so now can find keys of size up to 24503925291935817924440292461910
Found point with order 739 so now can find keys of size up to 18108400790740569446161376129351490
Found point with order 13 so now can find keys of size up to 235409210279627402800097889681569370
Found point with order 19 so now can find keys of size up to 4472774995312920653201859903949818030
Found point with order 181 so now can find keys of size up to 809572274151638638229536642614917063430
Found enough values! Running CRT...
Found private key 317478138448458883020444580626044847652
success!
```

סיבוכיות המתקפה היא $O(n_{max})$ כאשר n_{max} הוא הסדר הגדול ביותר מבין הסדרים של הנקודות הזדוניות. זה כי החלק הכי "כבד" במתקפה הוא ה-Brute Force על ה"מעגל" הגדול ביותר מבין ה"מעגלים" הקטנים, ולמזלו של התוקף הוא יכול לשלוט בערך זה כמעט לחלוטין. לכן המתקפה הזאת יחסית יעילה מבחינת סיבוכיות. כאמור שורש הבעיה במקרה זה היא שהמשתמש לא בודק שהנקודה שהוא קיבל בכלל נמצאת על העקומה שהוא עובד איתה. בנוסף, המשתמש משתמש באותו מפתח פרטי בכל שימוש חדש של ECDH, שזה לא כל כך בטיחותי.

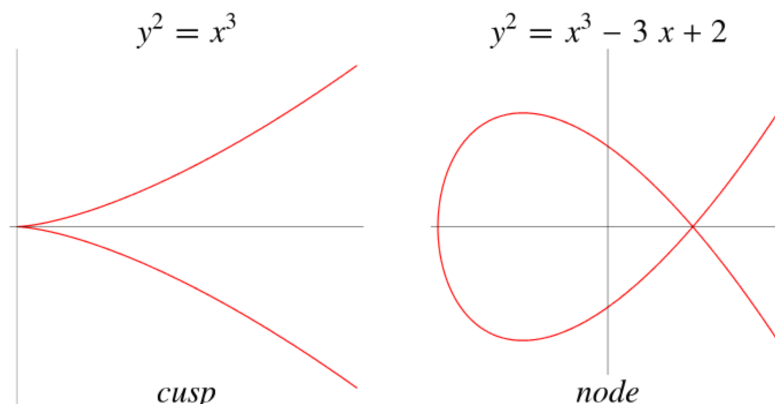
העקומה סינגולרית

אחת התכונות החשובות שעקומה אליפטית צריכה לקיים כדי להיות בטוחה קריפטוגרפית היא שהיא תהיה לא-סינגולרית. עקומה לא-סינגולרית היא עקומה שערך מסוים שלה, שנקרא ה"דיסקרימיננטה" של העקומה, שונה מאפס. זה מתקיים כשהפרמטרים a ו-b שלה מקיימים את אי השוויון:

$$4a^3 + 27b^2 \neq 0$$

בעקומה שלא מקיימת את אי השוויון הזה יש נקודה "בעייתית" שנקראת נקודה סינגולרית. קיימים שני סוגים של נקודות כאלה - node ו-cusp. נקודה מסוג node נמצאת על עקומה שיש בה מין לולאה כך שהיא חותכת את עצמה בנקודה הסינגולרית, וניתן להעביר דרך נקודה זו שני משיקים שונים לעקומה.

נקודה מסוג cusp היא נקודה שבה העקומה "מחודדת" (Sharp), כאילו יוצאות ממנה שתי זרועות אך יש בה רק משיק אחד:



[המחשה של עקומות סינגולריות מתוך https://mathworld.wolfram.com/EllipticDiscriminant.html]

בנקודות מסוג node יש שורש כפול, לכן אפשר לייצג את משוואת העקומה בצורה:

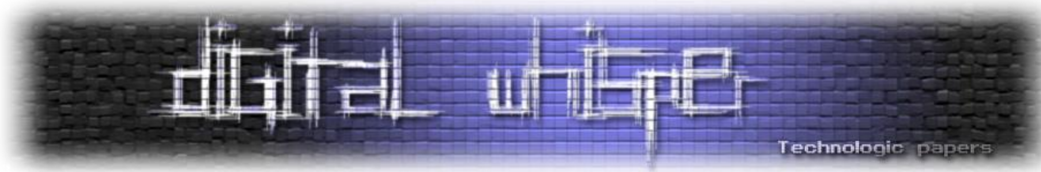
$$y^2 = (x - x_0)^2(x - x_1) \ (mod\, p)$$

ניתן "להזיז" את העקומה שמאלה על ידי החלפת המשתנה x במשתנה $(x + x_0)$ ולהגיע לצורה:

$$y^2 = x^2(x + x_0 - x_1) \ (mod\, p)$$

כך שעכשיו הנקודה הסינגולרית נמצאת בראשית הצירים. ניתן לקחת את הערך המספרי של $t = (x_0 - x_1)$ ולהשתמש בו כדי ליצור מיפוי בין נקודות על העקומה למספרים שלמים, כך שפעולת החיבור בין נקודות בעקומה תהיה שקולה לפעולת הכפל בין מספרים. לכל נקודה $(x,\ y)$ נתאים את המספר $\frac{y+\sqrt{t}x}{y-\sqrt{t}x}$. בפרט, לזוג נקודות G ו-Q כך ש $Q = nG$ נוכל להתאים מספרים g ו-q כך שיתקיים $q \equiv g^n \ (mod\, p)$, וזו כבר בעיית DLP "רגילה". כדי להמחיש את התהליך הזה, הוספתי קישור לדוגמא עם מספרים קטנים ברפרנסים בסוף המאמר. במיפוי שעשינו השתמשנו במשוואות הישרים $y + \sqrt{t}x$ ו-$y - \sqrt{t}x$, ואלה הישרים שמתאימים לשני המשיקים שניתן להעביר בנקודה הסינגולרית (אחרי ש"הזזנו" את העקומה), וזו בעצם הסיבה שניתן להשתמש במתקפה הזו.

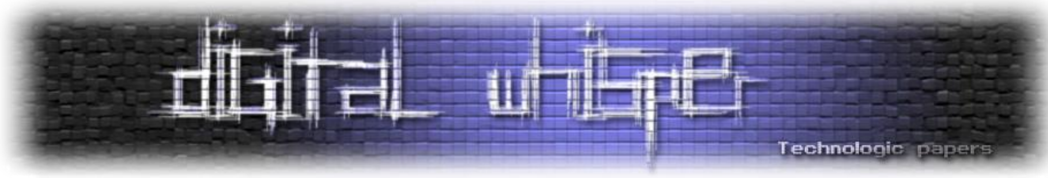
בעיית DLP כזאת ניתן לפתור באופן יעיל בעזרת אלגוריתם Pohlig-Hellman שכבר ראינו קודם, כי ניתן להפעיל אותו על גם מספרים רגילים במקום על נקודות בעקומה. בהקשר של נקודות, ראינו שהאלגוריתם הזה שימושי כשהסדר של הגנרטור הוא מספר חלק. בשונה מ"מעגל" נקודות על עקומה, שלו יש סדר כלשהו, בשדה מספרים שלמים מודולו מספר ראשוני p הסדר הוא $p - 1$. אם המספר $p - 1$ הוא מספר חלק, אז האלגוריתם יפתור את בעיית DLP ביעילות, וכך נמצא את המפתח הפרטי n.



הקוד הבא מבצע את המתקפה:

```
p = 1023607756169275769833854642603075344069139889946410834883718414176012375894887
a = -3
b = 2
assert (4*a^3 + 27*b^2) % p == 0

Gx = 177767113569874684756871012512942413225552915391411233783483524024781986964
Gy = 678642431430762579010888255422566678137582185588499347358652177173745476221 7
Qx = 4554146869535447131724812314637660983990939885004539637793130080863506495083 6
Qy = 4219190988572810527971802702508392309228261849745160116240559499179237653006 6

x = GF(p)["x"].gen()
f = x^3 + a*x + b
roots = f.roots()

assert len(roots) == 2 # two roots, so one must be double
if roots[0][1] == 2:
    double_root = roots[0][0]
    single_root = roots[1][0]
else:
    double_root = roots[1][0]
    single_root = roots[0][0]

print("double root:", double_root)
print("single root:", single_root)

# map G and Q to the new "shifted" curve
Gx = (Gx - double_root)
Qx = (Qx - double_root)

# Transform G and Q into numbers g and q, such that q=g^n
t = double_root - single_root
t_sqrt = t.square_root()

def transform(x, y, t_sqrt):
    return (y + t_sqrt * x) / (y - t_sqrt * x)

g = transform(Gx, Gy, t_sqrt)
q = transform(Qx, Qy, t_sqrt)
print("g:", g)
print("q:", q)

# Find the private key n
print("Factors of p-1:", factor(p-1))
print("Calculating discrete log for g and q...")
found_key = discrete_log(q, g)
print("Found private key:", found_key)

from Crypto.Util.number import long_to_bytes
print("The secret is:", long_to_bytes(found_key).decode())
```

בקטע קוד זה מגדירים את הפרמטרים של העקומה האליפטית, ומוודאים שהיא אכן סינגולרית. מוצאים את שורשי הפולינום המתאים לעקומה, ומזהים מי מביניהם הוא השורש הכפול. משתמשים בשורש הכפול כדי "להזיז" את העקומה, ולהגיע לנקודות G ו-Q "מוזזות". לאחר מכן מחשבים מהשורשים שמצאנו את $\sqrt{t}$ ומשתמשים בו כדי למפות את הנקודות G ו-Q למספרים g ו-q. מדפיסים את הפירוק של $p-1$ לגורמים הראשוניים שלו (כדי לוודא שאכן ניתן לפתור את DLP ביעילות). לבסוף מחשבים את DLP ומפרשים את התוצאה כמחרוזת.



מה שמודפס למסך הוא:

```
double root: 1
single root:
10236077561692757698338546426030753440691398899464108348837184141760123758948 5
g: 79308184675041981395063385790064051127319168083579208141274962436724168376607
q: 72551144069373709737718398534799929820619379063890479978458954196900267190559
Factors of p-1: 2 * 41 * 2422091127107 * 3224683479179 * 3224849279789 * 3269304069319
* 3792634171577 * 3997021218613
Calculating discrete log for g and q...
Found private key:
30943506368388267314266516224984737426569114488424608324579076903023329506337
The secret is: Digital Whisper is pretty great!
```

הפעם החבאתי מסר במפתח הפרטי עצמו. יש לשים לב שבגלל שמדובר בעקומה סינגולרית, לא ניתן ב-SageMath ליצור אותה בצורה רגילה, להגדיר עליה נקודות ולבצע איתן פעולות כמו שעשינו קודם. לכן בקוד זה הגדרתי את הקואורדינטות של הנקודות כמשתנים קבועים. כדי לחשב את הנקודה Q הכפלתי בעצמי את הגנרטור במפתח הפרטי בעזרת מימוש משלי של אלגוריתם Double And Add.

העקומה סופר-סינגולרית

בהינתן עקומה אליפטית מודולו p, וגנרטור שהסדר שלו הוא n, ה-Embedding Degree שלה ביחס לגנרטור מוגדר להיות המספר k הקטן ביותר המקיים את המשוואה $p^k \equiv 1 \ (mod\ n)$. על ידי טרנספורמציות מסוימות, ניתן להמיר את בעיית ECDLP בבעיית DLP בשדה מסדר p^k. הערך k הוא בדרך כלל מספר גדול מאוד (באותו אורך כמו p עצמו), אך כאשר הוא קטן יחסית (נניח קטן מ-6), העקומה נקראת סופר-סינגולרית ונהיה מעשי לפתור את בעיית DLP הזו ביעילות. מתקפה זו נקראת MOV attack, על שם שלושת הממציאים שלה (Menezes-Okamoto-Vanstone).

הטרנספורמציות שהזכרתי הן פונקציות שמקבלות שתי נקודות, ומתאימות להן מספר כלשהו בשדה המספרים המרוכבים. טרנספורמציות שניתן להשתמש בהן הן Weil Pairing או Tate Pairing, ונשתמש בהן כקופסא שחורה. טרנספורמציה T שכזאת מקיימת לכל זוג נקודות P,Q ומספרים m,n את התכונה הבאה:

$$T(mP,nQ) = T(P,Q)^{mn}$$

לכן בהינתן שתי נקודות G ו-$Q = mG$, נוכל לבחור בצורה אקראית נקודה שלישית R ולחשב את $g = T(G,R)$ ואז $q = T(Q,R) = T(mG,R) = T(G,R)^m = g^m$. מכאן ניתן לפתור את בעיית DLP על g ו-q בשדה מסדר p^k, ובכך למצוא את המפתח הפרטי m. צירפתי ברפרנסים בסוף המאמר קישור להסבר יותר מפורט על המתמטיקה שמאחורי מתקפה זו.

הקוד הבא מבצע את המתקפה:

```
p = 682209701131405092329016993551
a = -35
b = 98
E = EllipticCurve(GF(p), [a, b])
G = E(516365702870683577608927237052,
     524474557735717484100814381066)

# Find embedding degree k
Gn = G.order()
k = 1
while p^k % Gn != 1:
    k += 1
print("Found k:", k)

# Select private key, and calculate public key Q
private_key = 5072587499125503347
Q = private_key * G

# Define new curve mod p^k and the points on it
Ek = EllipticCurve(GF(p ^ k), [a, b])
Gk = Ek(G)
Qk = Ek(Q)
Rk = Ek.random_point()

# Find a point T with order d such that d divides G's order
m = Rk.order()
d = gcd(m, Gn)
Tk = (m // d) * Rk
assert Tk.order() == d
assert (Gn*Tk).is_zero() # Point INFINITY

# Using T, pair G and Q to integers g and q such that q=g^n (mod p^k)
g = Gk.weil_pairing(Tk, Gn)
q = Qk.weil_pairing(Tk, Gn)
# Alternatively:
#g = Gk.tate_pairing(Tk, Gn, k)
#q = Qk.tate_pairing(Tk, Gn, k)

# Make sure the pairing did not break anything
assert g ^ private_key == q

print("Calculating private key...")
found_key = q.log(g)
assert found_key == private_key
print("success!")

from Crypto.Util.number import long_to_bytes
print("The private key is:", long_to_bytes(found_key).decode())
```

בקטע קוד זה מגדירים עקומה וגנרטור שלה, ומחשבים את ערך ה-Embedding Degree שלו, שערכו במקרה זה הוא 2 ולכן זה מעשי לבצע את המתקפה. מגדירים עקומה זהה לעקומה המקורית, פרט לכך שהחישובים בה נעשים מודולו p^k במקום מודולו p. שתי הנקודות G ו-Q נמצאות גם כן על העקומה החדשה. לאחר מכן מוצאים נקודה שלישית שהסדר שלה מחלק את n.

בעזרת הנקודה השלישית, ממפים את הנקודות G ו-Q למספרים g ו-q ומחשבים את הלוגריתם הדיסקרטי עבורם. לבסוף מוודאים שהתוצאה שהתקבלה אכן נכונה.

מה שמודפס למסך הוא:

```
Found k: 2
Calculating private key...
success!
The private key is: Festivus
```

מבחינה חישובית קיימים היום אלגוריתמים מסוג Index Calculus שמאפשרים לפתור את בעיית DLP בצורה יעילה יחסית, והם עושים זאת בסיבוכיות $e^{O\left((\log p^k)^{1/3}(\log\log p^k)^{2/3}\right)}$. אולי הביטוי הזה נראה מפחיד, אבל בהשוואה לאלגוריתמים של ECDLP שסיבוכיותם היא $O(\sqrt{p}) = e^{O(\log p)}$, רואים שיותר קל לפתור את בעיית DLP, וזאת בהנחה שה-Embedding Degree (שמסומן ב-k) הוא אכן קטן.

העקומה אנומלית

אם לעקומה מסוימת יש את התכונה שסדר העקומה (כמות הנקודות שעליה) שווה בדיוק למודולוס p, אז היא נקראת עקומה אנומלית (Anomalous Curve) והיא פגיעה למתקפה בשם Smart's Attack. במתקפה זו משתמשים במספרים p-אדיים (p-adic number). מספר כזה ניתן להציג כסכום של חזקות (חיוביות ושליליות) של p עם מקדמים כלשהם. באופן פורמלי מספר s כזה הוא טור מהצורה:

$$s = \sum_{i=-k}^{\infty} a_i p^i = a_{-k}p^{-k} + \cdots + a_0 + a_1 p + a_2 p^2 + \cdots$$

כשהמקדמים הם מספרים שלמים בטווח $0 \leq a_i < p$, והסכום יכול להיות אינסופי בכיוון החזקות החיוביות של p. במספרים כאלה "מסתכלים" על ספרות המספר מימין לשמאל במקום משמאל לימין, ולכן טור שכזה יכול להתכנס לאיזשהו ערך. מספרים כאלה שייכים למערכת מספרים אחרת מזו שאנחנו מכירים, ומתנהגים באופן שונה מאוד מהכללים המתמטיים ה"רגילים". אפשר לכתוב מאמר נפרד שלם רק על הנושא הזה, ולמי שמתעניין בו צירפתי ברפרנסים בסוף המאמר קישור לסרטון שמציג אותו בצורה יחסית ברורה.

בכל מקרה, במתקפה זו יוצרים מהעקומה הנתונה עקומה חדשה שמוגדרת להיות מעל המספרים ה-p-אדיים. בהינתן שתי נקודות G ו-$Q = mG$ על העקומה המקורית, ממפים להן נקודות מתאימות על העקומה החדשה. מהקואורדינטות של הנקודות שהתקבלו ניתן לחשב בקלות את m.

הקוד הבא מבצע את המתקפה:

```
def lift(P, E, p):
    # lift point P from old curve to a new curve
    Px, Py = map(ZZ, P.xy())
    for point in E.lift_x(Px, all=True):
        # take the matching one of the 2 points corresponding to this x on the p-adic curve
        _, y = map(ZZ, point.xy())
        if y % p == Py:
            return point
```

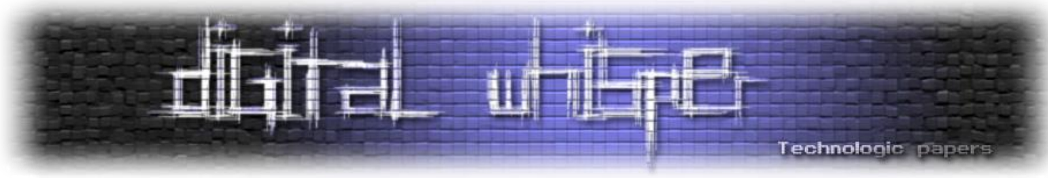
```
p = 82880337306360052550952380657384418102169134986290141696988204552000561657747
a = 26413685284385555604181540288021678971301314378522544469879270355650843743231
b = 10017655579196313780863100027113686719855502076415017585743221280232958057095
E = EllipticCurve(GF(p), [a, b])
G = E(37991937053350834320678619330546903567320901767090609881924528835279022654346,
      28947208718252880061735762506756351277969075978732800286053352115837132331595)
assert E.order() == p

private_key =
28153370716511608040616395150859085058202177279382452583684367923334520519740
P = private_key * G

# Lift the points to some new curve over p-adic numbers
E_adic = EllipticCurve(Qp(p), [a+p*13, b+p*37])
G = p * lift(G, E_adic, p)
P = p * lift(P, E_adic, p)

# Calculate discrete log
Gx, Gy = G.xy()
Px, Py = P.xy()
found_key = int(GF(p)((Px / Py) / (Gx / Gy)))
assert found_key == private_key
print("success!")

from Crypto.Util.number import long_to_bytes
print("The private key is:", long_to_bytes(found_key).decode())
```

בקטע קוד זה ראשית מגדירים פונקציית lift שמקבלת נקודה על העקומה המקורית, ומתאימה לה נקודה מתאימה על עקומת היעד. לאחר מכן מגדירים עקומה וגנרטור שלה, ומוודאים שהסדר של העקומה הוא אכן p. בוחרים מפתח פרטי ומחשבים את המפתח הציבורי המתאים לו. ואז מבצעים את המתקפה. מגדירים עקומה חדשה מעל המספרים ה-p-אדיים, ומתאימים לנקודות G ו-P המקוריות נקודות מתאימות בעקומה החדשה בעזרת הפונקציה lift והכפלתן ב-p.

לכל נקודה חדשה מחשבים את היחס בין קואורדינטת ה-x שלה לקואורדינטת ה-y שלה. המנה של שני הערכים האלה היא פתרון ה-ECDLP בנקודות המקוריות. מה שמודפס למסך הוא:

```
success!
The private key is: >>>>> Extraordinarily Nice <<<<<
```

הסיבה שהחישוב הזה עובד קשורה בעובדה שכמות הנקודות על העקומה היא בדיוק p. תכונה זו מאפשרת לנו לבצע מספר מיפויים שהאחרון בהם ממפה בין נקודות בעקומה מעל מספרים p-אדיים, למספרים מודולו p^2. המיפוי הזה מקיים את התכונה שיחס בין זוג המספרים המתאימים לשתי הנקודות המקוריות, הוא בדיוק תוצאת הלוגריתם של שתי הנקודות. את כל המיפויים האלה נשאיר כקופסא שחורה, אך בסוף המאמר הוספתי רפרנסים להסברים המתמטיים הרלוונטים.



מתקפות ידועות על ECDSA

לא מחשבים hash של ההודעה לפני שחותמים עליה

ראינו שבתהליך החתימה על הודעה ראשית מחשבים את ה-hash של ההודעה, מהתוצאה לוקחים את הביטים העליונים ומשתמשים בהם בחישוב החתימה. נניח שבמימוש כלשהו של חתימה מדלגים על חישוב ה-hash של ההודעה, ובמקום לקחת את הביטים העליונים של ה-hash לוקחים את הביטים העליונים של ההודעה כמו שהיא. במימוש שכזה, החלק היחיד בהודעה שמשפיע על החתימה שלה הוא תחילת ההודעה. במילים אחרות, אם יש לנו הודעה וחתימה שלה, נוכל להשאיר את תחילת ההודעה קבועה ולשנות את ההמשך שלה, והחתימה תישאר חוקית. זו מתקפה ממש פשוטה.

נניח למשל שתכתבו לבנק שלכם את ההודעה הבאה ותחתמו עליה מבלי לעשות עליה hash:

"נא להעביר מהחשבון שלי אלף ש"ח לאפיק קסטיאל כדי שימשיך לתחזק את Digital Whisper האהוב"

הבנק יאמת בהצלחה שההודעה הזו נחתמה כראוי, ויבצע את הפעולה. תוקף ...כלשהו... יוכל ליצור את ההודעה הבאה:

"נא להעביר מהחשבון שלי אלף ש"ח לאפיק קסטיאל ומיליון ש"ח לאלי קסקי"

ולהשתמש בחתימה שהרגע יצרתם. החתימה תהיה חוקית גם להודעה זו, והבנק יבצע את הפעולות. לא טוב. (בעצם תלוי למי).

הקוד הבא מדגים את המתקפה:

```
from ecdsa import SigningKey, NIST256p

signing_key = SigningKey.generate(NIST256p)
verifying_key = signing_key.verifying_key

class MyHash:
    def __init__(self, data):
        self.data = data

    def digest(self):
        return self.data

# Sign the message and verify the signature
message = "Please transfer 1,000$ to Digital Whisper"
signature = signing_key.sign(message.encode(), hashfunc=MyHash)
assert verifying_key.verify(signature, message.encode(), hashfunc=MyHash)

# Construct an evil message and verify the original message's signature is
valid for it as well
evil_message = "Please transfer 1,000$ to Digital Whisper and 1,000,000$ to
Eli"
assert verifying_key.verify(signature, evil_message.encode(), hashfunc=MyHash)
print("success!")
```

בקטע קוד זה משתמשים בספריה ecdsa ובעקומה ידועה. בקוד מוגדרת מחלקה שאמורה לממש פונקציית hash אך לא מבצעת זאת, ומשאירה את ההודעה כמו שהיא. לכן בחתימה על הודעה משתמשים רק בביטים הראשונים מתוך ההודעה המקורית, במקום מתוך ה-hash שלה. בהמשך הקוד מתבצעת חתימה על ההודעה ואימות שלה. לאחר מכן מוגדרת הודעה זדונית, והקוד מוודא שהחתימה על ההודעה המקורית מתאימה גם להודעה הזדונית.

בתרחיש כזה אמנם לא השגנו את המפתח הפרטי כדי ליצור חתימות חדשות משלנו, אבל בהינתן חתימה אחת אנחנו יכולים לחתום על כמה הודעות שנרצה, בתנאי שהן מתחילות באותו ערך.

שימוש באותו ערך k יותר מפעם אחת בחתימות שונות

כחלק מתהליך החתימה על הודעה, המשתמש נדרש להגריל ערך k בצורה אקראית ולהשתמש בו כדי לחתום על ההודעה. חשוב מאוד להשתמש בערכי k שונים בחתימות שונות. אחרת - בהינתן שתי הודעות חתומות, שבהן המשתמש השתמש באותו ערך k במקום להגריל אותו מחדש, תוקף יכול לחשב את המפתח הפרטי של המשתמש.

כזכור, בחתימה המשתמש שולח באופן ציבורי את $r = x_1 \pmod n$ ואת $s = k^{-1}(z + rd_A)$. נניח שמשתמש כלשהו חתם על שתי הודעות שונות שמתאימות ל- z_1 ו-z_2, ושלח באופן ציבורי שני זוגות ערכים r, s_1 ו- r, s_2, כלומר השתמש באותו ערך k בשתי החתימות האלה. נשים לב שמתקיים:

$$s_1 - s_2 = k^{-1}(z_1 + rd_A) - k^{-1}(z_2 + rd_A) = k^{-1}(z_1 + rd_A - z_2 - rd_A) = k^{-1}(z_1 - z_2)$$

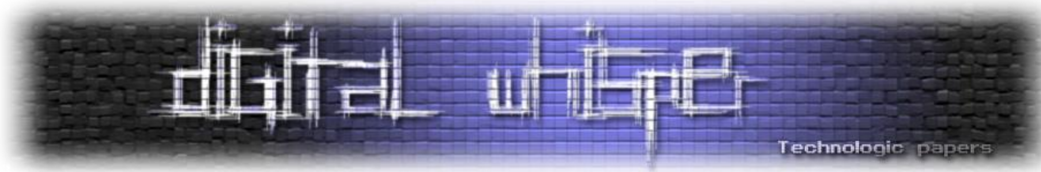
מכאן, התוקף יכול למצוא את הערך של k על ידי החישוב:

$$k = \frac{z_1 - z_2}{s_1 - s_2}$$

לאחר שהתוקף מצא את k, הוא יכול לחשב את המפתח הפרטי של המשתמש מתוך אחת החתימות. נשים לב שמתקיים:

$$r^{-1}(ks - z) = r^{-1}(kk^{-1}(z + rd_A) - z) = r^{-1}(z + rd_A - z) = r^{-1}rd_A = d_A$$

בהינתן ערכי r, s ו-z של הודעה והחתימה שלה, והערך של k שהתוקף מצא, התוקף יחשב את $d_A = r^{-1}(ks - z)$. מנקודה זו, התוקף יכול לחתום על כל הודעה שהוא רוצה, בשם המשתמש שאת המפתח הפרטי שלו הוא השיג.



הקוד הבא מבצע את המתקפה:

```
from ecdsa.ecdsa import curve_256, generator_256, Public_key, Private_key
from Crypto.Util.number import bytes_to_long,  long_to_bytes
from hashlib import sha256
import random

# Select a curve and generator
curve = curve_256
generator = generator_256
n = generator.order()

# Create private key and public keys
secret_key = 67435291307740909279281011696174811547823O9
public_key = Public_key(generator, generator * secret_key)
private_key = Private_key(public_key, secret_key)

# Sign 2 messages using the same k
k = random.randrange(curve.p())
message1 = "Life is like a box of chocolates."
message2 = "You never know what you're gonna get."
z1 = bytes_to_long(sha256(message1.encode()).digest())
z2 = bytes_to_long(sha256(message2.encode()).digest())

signature1 = private_key.sign(z1, k)
signature2 = private_key.sign(z2, k)

# Given the two messages and their signatures, find k
found_k = (z1 - z2) * inverse_mod(signature1.s - signature2.s, n) % n
assert k == found_k

# Given k and one of the messages, find the private key
found_key = inverse_mod(signature1.r, n) * (found_k * signature1.s - z1) % n
assert found_key == secret_key
print("success!")
print("The secret is:", long_to_bytes(found_key).decode())
```

בקטע קוד זה משתמשים בספריה ecdsa ובעקומה ידועה. מגדירים מפתח פרטי ומשתמשים בו כדי לחתום על שתי הודעות. הערך של k אמנם מוגרל בצורה אקראית, אך הוא נשאר זהה עבור שתי החתימות. בהינתן שתי ההודעות והחתימות שלהן, הקוד מבצע את החישוב שראינו כדי למצוא את k.
לבסוף, משתמשים בערך k שנמצא כדי לחשב את המפתח הפרטי כפי שראינו. מה שמודפס למסך הוא:

```
success!
The secret is: Mistakes were made
```

מעניין לציין שמתקפה זו יושמה בפועל בשנת 2010, כשחברת סוני מימשה בצורה לא בטוחה את מנגנון החתימה שלה על התוכנה של קונסולת פלייסטיישן 3. סוני השתמשה בערך סטטי של k עבור החתימות שלה, מה שאיפשר לתוקפים להשיג את המפתח הפרטי של סוני בעזרת החישוב הנ"ל. הדבר הוביל לכך שניתן היה לחתום על כל קוד שהוא, ולגרום לפלייסטיישן 3 להסכים להריץ אותו. בהמשך השתמשו ביכולת זו כדי להתקין משחקים פיראטיים ולא-רשמיים על הקונסולה.

בחירת k בצורה לא בטיחותית

אם המשתמש בוחר k בצורה לא מספיק אקראית, אז אפשר למצוא את המפתח הפרטי. למשל, אם לתוקף ידוע ש-k נמצא בטווח מאוד מצומצם של ערכים, או שידועים לתוקף חלק מהבתים של k, אז ניתן על ידי Brute Force פשוט למצוא את המפתח הפרטי של המשתמש על ידי הודעה חתומה אחת. התוקף יריץ את החישוב שראינו במתקפה הקודמת עבור ערכי k השונים, עד שיגיע לערך הנכון וישיג ממנו את המפתח הפרטי.

כדי להתגבר על בעיה זו, לפעמים המשתמשים מגרילים איזשהו ערך, מחשבים את ה-hash שלו עם פונקציית hash כלשהי, ומשתמשים בתוצאה בתור k. השיטה הזאת עלולה לגרום לבעיות. נניח למשל שהאורך של סדר הגנרטור, n, הוא 256 ביט, ופונקציית ה-hash שנבחרה היא SHA-1. הפלט של הפונקציה הזאת הוא מספר באורך של 160 ביט. בחישובים מודולו n, ידוע שהערך של k מכיל בתחילתו 96 אפסים, כלומר k הוא מספר יחסית קטן. במצב כזה אומרים שערכי k הם מוטים (Biased), ובהינתן מספר הודעות שנחתמו עם אותו המפתח הפרטי, ניתן למצוא את המפתח הפרטי.

המתקפה מתבססת על מבנה אלגברי שנקרא סריג (Lattice). באופן לא רשמי, אפשר לחשוב על סריג כעל קבוצה של וקטורים במרחב m-מימדי, שאפשר לבטא אותם בתור צירוף לינארי של וקטורי "בסיס" עם מקדמים שלמים. במתמטית, אם $\{b_1, \ldots, b_d\}$ הם וקטורי הבסיס מעל $\mathbb{R}^m$ אז הסריג המתאים להם הוא $\mathcal{L} = \{\sum_{i=1}^{d} a_i b_i \,|a_i \in \mathbb{Z}\}$. במבנה זה קיימת הבעיה: בהינתן בסיס של סריג, צריך למצוא את הוקטור הקצר ביותר שנמצא בסריג. בהקשר זה, באופן לא רשמי, "וקטור קצר" זה וקטור שאיבריו קרובים לאפס ככל הניתן. בעיה זו נקראת Shortest Vector Problem (SVP), והיא נחשבת NP-קשה. קיימים אלגוריתמים שפותרים בעיה דומה אך קלה יותר - מציאת וקטור קצר כלשהו, כלומר וקטור יחסית "קרוב" לוקטור הקצר ביותר בסריג. בעיה זו נקראת Closest Vector Problem (CVP), ואחד האלגוריתמים שפותרים אותה נקרא אלגוריתם Lenstra-Lenstra-Lovász (LLL). במתקפה זו נשתמש באלגוריתם הזה כקופסא שחורה.

בהינתן d הודעות חתומות, ניתן לבנות סריג שמכיל את הוקטור $(k_1, \ldots, k_d)$, כאשר כל אחד מאיברי הוקטור הוא ערך k שמתאים לחתימה אחת. האלגוריתם LLL ימצא לנו קירוב לוקטור הקצר ביותר בסריג הזה. מכיוון שידוע שערכי k הם קטנים, בהסתברות גבוהה הוקטור הקצר שהאלגוריתם ימצא יכיל לפחות איבר k אחד נכון. ברגע שמצאנו k אחד נכון אז ניתן למצוא את המפתח הפרטי כפי שראינו במתקפה הקודמת.

כדי לבנות את הסריג הזה, צריך להגדיר את וקטורי הבסיס שלו. צירפתי ברפרנסים בסוף המאמר לינק למאמר שמסביר איך וקטורי הבסיס האלה מוגדרים. טכנית, ניתן לייצג את וקטורי הבסיס של הסריג כמטריצה, כך שכל שורה בה מורכבת מאיברי וקטור בסיס אחד. כדי לשפר את הדיוק של אלגוריתם LLL, מומלץ להוסיף למטריצה זו שתי עמודות שמכילות מידע על הגודל הצפוי של ערכי k ועל היחס בין k ל-n. השיפור הזה גם מוסבר ברפרנס שצירפתי.

הקוד הבא מבצע את המתקפה:

```
from ecdsa.ecdsa import curve_256, generator_256, Public_key, Private_key
from Crypto.Util.number import bytes_to_long, long_to_bytes
from hashlib import sha1
import random


def build_matrix(signatures, bias, q):
    # M matrix should be:
    """
    [
            B 0   m'1 m'2 m'2 ... m'n
            0 B/q r'1 r'2 r'3 ... r'n
            0 0
            0 0        q * I
            0 0
    ]
    where:
        m' = s^-1 * m
        r' = s^-1 * r
    """

    # Construct the first 2 rows of M:
    row1 = [bias, 0]
    row2 = [0, bias / q]
    for m, r, s in signatures:
        row1.append((inverse_mod(s, q) * m) % q)
        row2.append((inverse_mod(s, q) * r) % q)
    top_rows = Matrix(QQ, [row1, row2])

    # Construct the q*I block along with 2 columns of zeros
    zero_cols = zero_matrix(QQ, len(signatures), 2)
    qI = q * identity_matrix(QQ, len(signatures))
    bottom_rows = block_matrix([[zero_cols, qI]])

    # Combine all rows into one matrix
    M = top_rows.stack(bottom_rows)
    return M


def find_private_key(L, signatures, public_key):
    # Check if any valid k was found in L
    generator = public_key.generator
    q = generator.order()
    for row in L.rows():
        for i in range(len(signatures)):
            m,r,s = signatures[i]
            # Skip the first two vector components we used to improve LLL
            possible_k = row[i+2]
            # LLL might have swapped the sign of the found short vectors
            for k in [possible_k, -possible_k]:
                d = inverse_mod(r,q)*(k*s-m) % q
                if d*generator == public_key.point:
                    return d


# Select a curve and generator
curve = curve_256
generator = generator_256
q = int(generator_256.order())

# Create private key and public key
secret_key = 17930562343097730778621250068433837260292627646807278516
```

```
public_key = Public_key(generator, generator * secret_key)
private_key = Private_key(public_key, secret_key)

# Sign some messages
messages_to_sign = [
    "And then I go and spoil it all",
    "By saying somethin' stupid like",
    "I love you"
]

signatures = []
for message in messages_to_sign:
    message_hash = bytes_to_long(sha1(message.encode()).digest())
    k = bytes_to_long(sha1(long_to_bytes(random.randrange(q))).digest())
    signature = private_key.sign(message_hash, k)
    signatures.append((message_hash, signature.r, signature.s))

# Given the messages and their signatures, retrieve the private key

# Build the matrix out of the signatures
# We know that k < 2^160 because it is the result of sha1
bias = 2^160
M = build_matrix(signatures, bias, q)

# Calculate the closest short vector
L = M.LLL()

# Find the private key!
found_key = find_private_key(L, signatures, public_key)
assert found_key == secret_key
print("success!")
print("The secret is:", long_to_bytes(found_key).decode())
```

בקטע קוד זה בוחרים עקומה סטנדרטית ומפתח פרטי, ומחשבים לו מפתח ציבורי מתאים. יוצרים 3 הודעות וחותמים עליהן עם 3 ערכי k אקראיים שהם תוצאה של פונקציית SHA-1. לאחר מכן יוצרים את המטריצה המתאימה לבסיס הסריג כפי שמוסבר במאמר, ומפעילים עליה את אלגוריתם LLL. לבסוף, עוברים על שורות המטריצה המתקבלת ובודקים אם ערך נכון של k כלשהו נמצא באחת מהן.

הבדיקה מתבצעת על ידי חישוב המפתח הפרטי מתוך k, כפי שראינו במתקפה הקודמת, ובדיקה אם המפתח שהתקבל אכן נכון. לבסוף מוודאים שהמפתח הפרטי שנמצא הוא אכן נכון. מה שמודפס למסך הוא:

```
success!
The secret is: I am Jack's broken heart
```

סיבוכיות המתקפה הזו היא כסיבוכיות אלגוריתם LLL, שהיא $O(d^6 \log^3 B)$ כאשר B מציין את אורך ההטיה (Bias) של k (2^{160} במקרה שלנו), ו-d מציין את מספר ההודעות החתומות (3 במקרה שלנו). נשאלת השאלה מהו המספר המינימלי של הודעות חתומות שאנחנו נדרשים להשתמש בהן כדי שנוכל להריץ את המתקפה. התשובה לכך היא $d = O\left(\frac{\log n}{\log n - \log B}\right)$ כאשר n הוא סדר הגנרטור ו-B הוא ההטיה (Bias). הסבר לכך מופיע במאמר שברפרנס השני שצירפתי לנושא זה בסוף המאמר.

בפועל ניתן להריץ וריאציה של מתקפה זו גם במקרים שבו ידועים הביטים העליונים של k, או סתם ביטים מסוימים מתוך k. ניתן להריץ את המתקפה אפילו אם ידוע ערך של רק ביט אחד, או אפילו אם ידוע רק את ערכו של ביט אחד בהסתברות גדולה מ-50%! אך כמובן שבמקרים אלה צריך הרבה יותר הודעות חתומות כדי לבצע את המתקפה.

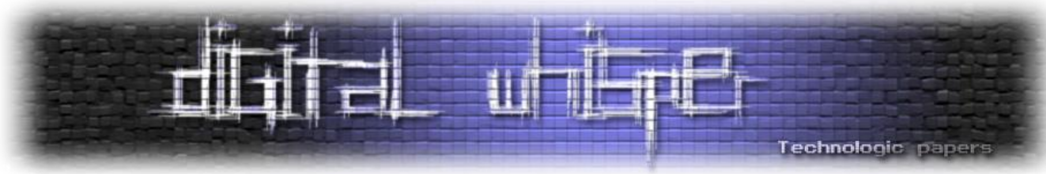
חוסר בדיקה שהגנרטור תקין

ראינו שבתהליך אימות חתימה, הצד החותם שולח לצד המאמת את זוג הערכים r ו-s. בדפדפנים שמממשים את פרוטוקול HTTPS למשל, נהוג לשלוח את זוג הערכים האלה ב-Certificate שיכול לפעמים להכיל גם נתונים על העקומה שהצד החותם השתמש בה. הצד המאמת צריך לוודא שנתוני העקומה שנמצאים ב-Certificate אכן מתאימים לעקומה שהוסכם עליה קודם לכן. אם הוא לא עושה זאת, עלולות להיווצר בעיות.

נניח שבעקומה מסוימת לאליס יש מפתח פרטי d_A ומפתח ציבורי P_A שמתאים לו, כלומר מתקיים $P_A = d_A G$ עבור הגנרטור G שבעקומה הזו. בעזרת המפתח הפרטי d_A, אליס יכולה לחתום על ההודעות שלה כפי שראינו בהגדרת הפרוטוקול ECDSA. נניח שהצד המאמת את החתימה מקבל מהמשתמש גם את הגנרטור G, ולא מוודא שהגנרטור שהתקבל מהמשתמש הוא אכן הגנרטור המוסכם. תוקף יכול לשלוח כגנרטור את הנקודה שהיא המפתח הציבורי של אליס, $G' = P_A$. התוקף יבחר כמפתח פרטי "מזויף" את הערך $d'_A = 1$, ומכאן ברור שמתקיים $P_A = d'_A G'$. כלומר התוקף יכול "להוכיח" שהוא בעל המפתח הפרטי שמתאים למפתח הציבורי של אליס. כך תוקף יכול ליצור כל הודעה שירצה, ולחשב עבורה זוג ערכי r ו-s באופן רגיל עם d'_A, והחתימה שתתקבל תאומת בהצלחה.

אינטואיטיבית, בתהליך החתימה, הצד החותם מוכיח שהוא אכן ה"בעלים" של המפתח הציבורי, שהוא בעצם נקודת "יעד" על העקומה. זה כי הוא היחיד שיודע כמה צעדים לקחת מנקודת ההתחלה כדי להגיע לנקודת היעד. אם הצד המאמת לא בודק שנקודת ההתחלה שהתקבלה מהמשתמש היא אכן נקודת ההתחלה האמיתית, אז תוקף יכול להחליט שנקודת ההתחלה שלו היא נקודת היעד, ושכמות הצעדים שצריך לקחת ממנה היא אפס. כל שאר חלקי אימות החתימה נשארים זהים, והחתימה תאומת בהצלחה. המתקפה הזו נקראת Curveball.

באופן כללי ניתן להכליל את המתקפה עם ערכים נוספים. התוקף יבחר ערך כלשהו x, ויחשב את $G' = xP_A$. המפתח הפרטי המזויף יהיה $d'_A = x^{-1} \ (mod\ n)$. אז ברור שמתקיים $d'_A G' = x^{-1} x P_A =$ P_A.



הקוד הבא מדגים את המתקפה:

```
from ecdsa.ecdsa import generator_256
from Crypto.Util.number import bytes_to_long
from hashlib import sha256
import random

def hash_message(message):
    return bytes_to_long(sha256(message.encode()).digest())

def verify(public_key, G, message, r, s):
        n = G.order()
        if r < 1 or r > n - 1 or s < 1 or s > n-1:
            return False
        hash = hash_message(message)
        u1 = (hash * inverse_mod(s, n)) % n
        u2 = (r * inverse_mod(s, n)) % n
        P = u1 * G + u2 * public_key
        return P.x() % n == r

def sign(private_key, G, message):
    n = G.order()
    k = random.randrange(n)
    hash = hash_message(message)

    r = (k * G).x() % n
    s = inverse_mod(k, n) * (hash + r * private_key) % n
    return r, s

# Create private and public keys
G = generator_256
n = G.order()
private_key = random.randrange(n)
public_key = private_key * G

# Sign a message and verify it
message = "Let me be the one that shines with you"
r, s = sign(private_key, G, message)
assert verify(public_key, G, message, r, s)

# Create a fake private key and generator that match the original public key
x = random.randrange(n)
fake_G = x * public_key
fake_private_key = inverse_mod(x, n)
assert fake_private_key != private_key
assert fake_G != G

# Sign an evil message and verify it using the same public key
evil_message = "Where did I go wrong?"
r, s = sign(fake_private_key, fake_G, evil_message)
assert verify(public_key, fake_G, evil_message, r, s)
```

בקטע קוד זה בוחרים גנרטור ידוע, מפתח פרטי ומפתח ציבורי. חותמים על הודעה ומוודאים שהיא מאומתת בהצלחה. לאחר מכן יוצרים מפתח פרטי מזויף וגנרטור מזויף, כך ששניהם מתאימים למפתח הציבורי המקורי. מתבצעת חתימה על הודעה אחרת עם המפתח המזויף, ולבסוף מתבצע אימות מוצלח של החתימה המזויפת בעזרת המפתח הציבורי המקורי. הבעיה בקוד זה היא שאלגוריתם האימות לא מוודא שהגנרטור G תואם למפתח הציבורי. אמנם במתקפה זו לא מצאנו את המפתח הפרטי של המשתמש, אך תוקף יכול לנצל את המימוש השגוי לאימות חתימה, וליצור חתימה שתאומת בהצלחה. עם זאת, לא תוקף לא יכול ליצור חתימות "אמיתיות" שאכן יאומתו בהצלחה במימוש תקין של אימות חתימה.

מעניין לציין שזו חולשה אמיתית שהייתה קיימת בארכיטקטורת CryptoAPI של Windows. בפונקציה שנועדה לאמת חתימה של Certificate, לא הייתה בדיקה מספיקה של הפרמטרים של העקומה במידה והם נכללו ב-Certificate עצמו. בפרט, לא הייתה בדיקה שהגנרטור הוא אכן הגנרטור שמתאים למפתח הציבורי. תוקף יכל ליצור Certificate-ים מזויפים שנחשבים מהימנים (Trusted) כי הם נראים כאילו הם נחתמו על ידי Certificate Authority מהימן. הדבר התבצע על ידי הוספת שדות זדוניים של עקומה ל-Certificate, ובחירת הגנרטור בצורה שתיארתי. החולשה התגלתה על ידי ארגון ה-NSA, תוקנה בשנת 2020 וקיבלה את המספר CVE-2020-0601.

סיכום המתקפות הידועות על ECDH

סוג הבעיה	הבעיה	המתקפה	איך המתקפה עובדת	סיבוכיות המתקפה
בחירת עקומה עם גנרטור לא בטוח	סדר הגנרטור n הוא קטן מדי	Baby-Step Giant-Step	Meet In The Middle	$O(\sqrt{n})$
	סדר הגנרטור n הוא מספר חלק	Pohlig-Hellman	פירוק n לגורמים ראשוניים, תקיפת כל אחד מהם בנפרד, ואיחוד לתוצאה עם משפט השאריות הסיני	$O(\sqrt{p_{max}})$ כאשר p_{max} הוא הגורם הראשוני הגדול ביותר בפירוק של n
בחירת עקומה עם גנרטור לא בטוח + בחירת מפתח בצורה לא בטוחה	סדר הגנרטור n הוא מספר כמעט-חלק והמפתח קטן יחסי	Pohlig-Hellman משופר	פירוק n לגורמים ראשוניים, הסרה של גורמים גדולים מדי, תקיפת כל אחד מהם בנפרד, ואיחוד לתוצאה עם משפט השאריות הסיני	
מימוש שגוי של ECDH	חוסר בדיקה שהמפתח הציבורי שהתקבל מהמשתמש נמצא על העקומה	Invalid Curve Attack	שליחת נקודות מסדר קטן על עקומות זדוניות בתור מפתח ציבורי, תקיפת כל אחת מהן בנפרד, ואיחוד לתוצאה עם משפט השאריות הסיני	$O(n_{max})$ כאשר n_{max} הוא הסדר הגדול ביותר מבין הסדרים של הנקודות הזדוניות
בחירת פרמטרים בעקומה בצורה לא בטוחה	העקומה סינגולרית	המרת בעיית ECDLP לבעיית DLP	מיפוי נקודות למספרים באופן שממיר חיבור בין נקודות לכפל בין מספרים	$O(\sqrt{p_{max}})$ כאשר p_{max} הוא הגורם הראשוני הגדול ביותר בפירוק של $p-1$
	העקומה סופר-סינגולרית			$e^{O\left((\log p^k)^{1/3}(\log\log p^k)^{2/3}\right)}$ כאשר k הוא ה-Embedding Degree ביחס לגנרטור
	העקומה אנומלית	Smart's Attack	סדרת מיפויים מנקודות בעקומה לנקודות בעקומה מעל מספרים p-אדיים, ובחזרה למספרים שלמים	$O(1)$

סיכום המתקפות הידועות על ECDSA

סוג הבעיה	הבעיה	המתקפה	איך המתקפה עובדת	סיבוכיות המתקפה
מימוש שגוי של חתימה ואימות חתימה	אי חישוב hash של ההודעה לפני החתימה עליה	בהינתן הודעה חתומה, יצירת הודעות נוספות שמתאימות לאותה החתימה	שומרים את תחילת ההודעה קבועה ומשנים את ההמשך שלה	$O(1)$
שימוש שגוי באלגוריתם החתימה	שימוש באותו ערך k בחתימות שונות	מציאת המפתח הפרטי של המשתמש	מציאת הערך של k, וחישוב המפתח הפרטי של המשתמש ממנו	$O(1)$
	הגרלת ערך k בצורה לא בטוחה	בהינתן מספר הודעות חתומות, מציאת המפתח הפרטי של המשתמש	המרת הבעיה למציאת וקטור קצר בסריג, מציאת הערך של k כלשהו, וחישוב המפתח הפרטי של המשתמש ממנו	$O(d^6 \log^3 B)$ כאשר B היא ההטיה (Bias) של k, ו-d הוא מספר ההודעות החתומות
מימוש שגוי של אימות חתימה	חוסר בדיקה שהגנרטור תקין	יצירת חתימות מזויפות שיאומתו בהצלחה (Curveball)	בחירת גנרטור ומפתח פרטי מזויפים שמתאימים למפתח הציבורי של משתמש אחר	$O(1)$

התגוננות מפני המתקפות

יש לשים לב שב-ECDH, שני הצדדים צריכים להסכים ביניהם על העקומה בתחילת הפרוטוקול. אם משתמש מדבר עם תוקף כלשהו, והתוקף הוא זה שמספק את פרמטרי העקומה, אז התוקף יכול לספק פרמטרים לא בטיחותיים. כתוצאה מכך התוקף יכול להשיג את המפתח הפרטי של המשתמש. אם המשתמש תמיד משתמש באותו מפתח פרטי, אז התוקף יכול לפענח את כל השיחות בין המשתמש הזה לכל משתמש אחר. לכן חשוב מאוד לא לאפשר למשתמשים זרים לספק את פרמטרי העקומה במידה ולא ניתן לסמוך עליהם. בנוסף, צריך לוודא שכל נקודה שהתקבלה ממשתמש זר אכן נמצאת על העקומה שהוסכם עליה. וכמובן, כדאי לוודא שהעקומה עצמה שנבחרה לא פגיעה לאחת מהמתקפות הידועות שראינו. כמו כן, עדיף להשתמש במפתח פרטי חדש בכל שימוש בפרוטוקול ECDH.

באופן דומה, ב-ECDSA צריך להקפיד על מימוש נכון של אלגוריתם החתימה ואימות. לא לדלג על hash של המסר, על הגרלה אקראית ובטוחה של ערך k בכל פעם שמשתמשים בפרוטוקול מחדש, וכמובן באימות החתימה, אם הגנרטור מתקבל מהמשתמש - לוודא שהוא אכן זה שהוסכם עליו קודם לכן.

סיכום

במאמר זה למדנו מה הן עקומות אליפטיות, מה הפעולות הבסיסיות שניתן לעשות בהן, ומה השימושים שיש להן בהקשר קריפטוגרפי. בעיקר ראינו איך אפשר לשבור את המערכות האלה כשמשתמשים בהן בצורה לא בטוחה או כשמממשים אותן לא נכון.

מוסר השכל שחשוב לקחת ממאמר זה - לא להמציא בעצמנו ערכים לעקומות אליפטיות. עדיף להשתמש בעקומות סטנדרטיות ו"מוכחות" שנחשבות ליעילות ובטוחות, ושמומחי קריפטוגרפיה בדקו אותן. עקומות שכאלה נקראות עקומות NIST, על שם הארגון National Institute of Standards and Technology שממליץ עליהן. המשפט המפורסם הבא מסכם את העניין:

“Don’t roll your own cryptography”

חשוב לקחת בחשבון שיכול להיות שגם עצות אלה לא מספיקות. אולי קיימות היום מתקפות נוספות שפותחו על ידי ארגוני אבטחה ברחבי העולם ונשמרות בסוד. אולי קיימות פרצות אבטחה בעקומות NIST סטנדרטיות שנחשבות כ"בטוחות", עם טריקים מתמטיים לא ידועים.

בכל מקרה, אני מקווה שהצלחתי להעביר את הנושא הזה בצורה ברורה ואינטואיטיבית עד כמה שאפשר. נראה שהעולם הולך לכיוון Elliptic Curve Cryptography משיקולי יעילות ואבטחה, ואני מחכה לראות איזה עוד מתקפות יתגלו בעתיד.

מקווה שנהניתם מהקריאה!

על המחבר

אני אלי קסקי, בן 30, אוהב לכתוב ולפתור אתגרי CTF, במיוחד בקטגוריות Reverse Engineering וקריפטוגרפיה, ואוהב לעשות דברים מגניבים עם קוד.

לפניות בכל נושא מוזמנים ליצור איתי קשר ב-elikaski94@gmail.com או ב-LinkedIn.

רפרנסים

- במאמר זה השתמשתי בתרשימי גרפים מתוך הספר Understanding Cryptography מאת Christof Paar:
  https://gnanavelrec.wordpress.com/wp-content/uploads/2019/06/2.understanding-cryptography-by-christof-paar-.pdf
- אתר שממחיש איך נראות עקומות אליפטיות קריפטוגרפיות:
  https://graui.de/code/elliptic2/
- הסברים מפורטים על פעולות החיבור והכפל בעקומות אליפטיות:
  https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication
- הרצאה על מבוא לעקומות אליפטיות וחיבור בין נקודות - מאת Christof Paar:
  https://www.youtube.com/watch?v=vnpZXJL6QCQ
- הרצאה על גנרטורים, ECDLP, קושי של בעיות, ECDH, Double And Add - מאת Christof Paar:
  https://www.youtube.com/watch?v=zTt4gvuQ6sY
- הסבר על מדד Security Level של אלגוריתמי הצפנה שונים:
  https://en.wikipedia.org/wiki/Security_level
- הסבר על ECDH:
  https://en.wikipedia.org/wiki/Elliptic-curve_Diffie%E2%80%93Hellman
- הסבר על ECDSA:
  https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm
- הסבר על חתימות עם ElGamal:
  https://en.wikipedia.org/wiki/ElGamal_signature_scheme
- הסבר על משפט השאריות הסיני:
  https://en.wikipedia.org/wiki/Chinese_remainder_theorem
- הסבר על הקשר בין הדיסקרימיננטה של עקומה סינגולרית לכך שיש בה שורש כפול:
  https://www.quora.com/For-an-elliptic-curve-in-the-form-Y-2-X-3+AX+B-why-is-4A-3+27B-2-neq-0-the-condition-for-non-singularity

- דוגמא במספרים קטנים של המיפוי בין נקודות למספרים בעקומות סינגולריות:

https://crypto.stackexchange.com/questions/61302/how-to-solve-this-ecdlp/61434#61434

- הסברים על מספרים p-אדיים:

https://en.wikipedia.org/wiki/P-adic_number

https://www.youtube.com/watch?v=3gyHKCDq1YA

- הסבר על המתמטיקה מאחורי מתקפת MOV:

https://risencrypto.github.io/WeilMOV/

- הסברים על המתמטיקה מאחורי מתקפת Smart's Attack (היא די מסובכת, ראו הוזהרתם):

https://wstein.org/edu/2010/414/projects/novotney.pdf

http://www.monnerat.info/publications/anomalous.pdf

- הסבר על המתקפה מבוססת סריג ואלגוריתם LLL:

https://forum.vac.dev/t/lattice-attacks-on-ecdsa/136

המתקפה מתבססת על חלק 4 במאמר מאת Joachim Breitner ו-Nadia Heninger:

https://eprint.iacr.org/2019/023.pdf

- הסבר על בעיית CVP:

https://en.wikipedia.org/wiki/Lattice_problem#Closest_vector_problem_(CVP)

- הסבר על אלגוריתם LLL:

https://en.wikipedia.org/wiki/Lenstra%E2%80%93Lenstra%E2%80%93Lov%C3%A1sz_lattice_basis_reduction_algorithm