

TDK-dolgozat

Jámbor Zsanett

Debreceni Egyetem
Informatikai Kar
Számítógéptudományi Tanszék

Bilineáris leképezés alkalmazása

Témavezető: Dr. Pintér-Husztai Andrea
egyetemi docens,
Dr. Bertók Csanád adjunktus

Készítette: Jámbor Zsanett
programtervező informatikus

Debrecen

Tartalom

1	Bevezetés	1
2	Biztonságos jelszó tárolás.....	3
3	Hash függvény	3
3.1	Hash függvénnyel szembeni támadások	3
3.2	Só.....	5
3.3	Többkörösítés	6
4	Elliptikus görbék	6
4.1	Általánosan az elliptikus görbékről.....	6
4.2	Véges testek feletti elliptikus görbék	7
4.3	Művelet.....	7
4.3.1	Geometriai leírás	7
4.3.2	Algebrai leírás	8
4.4	Elliptikus görbe diszkrét logaritmus probléma (ECDLP)	11
4.5	Bilineáris párosítás	11
4.5.1	Tate párosítás	12
5	Gyakorlatban használt rendszerek	15
5.1	Bcrypt.....	15
5.2	Blowfish	15
5.3	Bcrypt működése.....	16
5.3.1	Tesztelés.....	17
5.4	Argon2.....	17
5.4.1	Argon2 változatok.....	17

5.4.2	Argon2 működése	18
5.4.3	Tesztelés.....	19
6	Tervezett rendszer.....	20
6.1	Az algoritmus	21
6.1.1	Elliptikus görbecsoportba történő hashelés	22
6.1.2	Tulajdonságok.....	24
6.1.3	Egyirányúság	24
6.1.4	Ütközés ellenállóság	24
7	Összehasonlítás.....	25
7.1	RSA	25
7.1.1	Algoritmus	25
II.	Titkosítás	25
III.	Visszafejtés	26
7.1.2	RSA példa	26
7.1.3	RSA algoritmus többkörösítve hasheléshez.....	27
7.2	Az összehasonlítás menete	27
8	Összefoglalás	30
	Köszönetnyilvánítás.....	32
	Saját munka leírása	33
9	Irodalomjegyzék	34

1 Bevezetés

A jelszavak használata régóta fontos szerepet játszik a felhasználó hitelesítési megoldásoknál, kezdetben például örök is jelszó ismeretében engedték át az embereket az ajtón. Manapság a jelszavakat jellemzően különböző weboldalakra való bejelentkezéshez, mobiltelefonoknál, pénzautomatáknál stb. használunk.

Mivel a hitelesítésben kritikus szerepe van, ezért kiemelten fontos a jelszavak megfelelő és biztonságos tárolása, ugyanis ennek hiányában komoly károkat szenvedhetünk, ha titkos vagy személyes információink jogosulatlanok kezébe kerülnek.

Számos hitelesítési rendszer szolgáltatónál tárolt jelszóval kapcsolatos információkra támaszkodik. Miután a felhasználó bejelentkezett, a megadott jelszó hash értéke kiszámításra kerül és összehasonlítják a tárolt hash értékkel. Amennyiben a két hash egyezik, akkor a hitelesítés sikeres.

A felhasználók sokszor gyenge jelszavakat választanak, vagy az alapértelmezett jelszót használják, ezek azonban a támadók számára könnyen kitalálhatóak. A jelszó mellett általában egy úgynevezett só értéket is tárolnak. A só egy rövid (12-48 bit) véletlenszerű adat, amelyet a jelszóhoz kell konkatenálni hashelés előtt.

A támadó, akinek sikerül ellopnia a jelszófájlt vagy az adatbázis, számítást igénylő offline támadást indít, hogy megtalálja a felhasználók jelszavait a szózott hash értékekből. Azaz minden lehetséges jelszót minden lehetséges só értékkel le kell tesztelni.

A tudományos irodalomban számos egy- és kétfaktoros hitelesítési rendszert javasolnak, amelyekben a jelszavak alapvető szerepet játszanak.

Manapság egyre gyakrabban olvashatunk arról, hogy a kiberbűnözők célpontjaivá váltak a jelszavak, a jelszókezelő alkalmazások sebezhetőek.

Tavalyi év végén jelent meg a hír ([19]), hogy közel 13 milliárd felhasználó adata kering az interneten, köztük magyar weboldalak adatai is. A magyar weblapok között a Budapesti Corvinus Egyetem és a Budapesti Metropolitan Egyetem belépési adatai is. Ezek a kiszivárgott adatok felhívhatják a figyelmünket arra, hogy különböző weboldalakhoz más jelszót használjunk. Ugyanis egy rosszakaró különböző weboldalakon is kipróbálhatja a kiszivárgott jelszavakat.

Az elmúlt években többször jelent meg hír arról, hogy jelszókezelő alkalmazásokban fedeztek fel sérülékenységet. 2019-ben például Independent Security Evaluators (ISE)

tanulmányában sérülékenységet fedeztek fel jelszókezelő programokban. A 1Password 4-es verziójában észrevették, hogy szabad szöveges formában a mesterjelszó hozzáférhető a RAM-ban ([17]). Majd 2021-ben egy ellátási lánc elleni támadást szenvedett a Passwordstate. A támadók tömörített fájlt juttattak az áldozatok eszközére, amelyben a káros kód képes volt megszerezni a felhasználók mester jelszavait ([18]).

Vannak olyan hash algoritmusok, amelyeket szándékosan lassúnak terveztek, hogy megnehezítsék az off-line támadásokat. A bcrypt hash séma például konfigurálható egy olyan költséggel, amely exponenciálisan növeli a végrehajtási időt. A Password Hashing Competition győztese az Argon2, amely jelszó hashelésre is használható. Fő jellemzője az time-memory trade-off támadások elleni védelem, ahol a támadó gyors hardveren, kevés memóriával végezheti a munkáját. Bemutatok egy biztonságos jelszó hashelő sémát, amelyet többkörösre terveztünk, így a körök számának növelésével lassítani tudjuk az algoritmust.

A dolgozat a napjainkban egyre inkább használt elliptikus görbe kriptográfia területéhez (ECC - Elliptic Curve Cryptography) kapcsolódó megoldást mutat be, a biztonságos jelszótárolásra ad javaslatot. Elliptikus görbecsoportok feletti bilineáris leképezést alkalmazunk, ahol a jelszavakat bilineáris leképezésekkel tároljuk egyetlen körben. Bemutatom a bilineáris leképezés egyirányúságát. A körök számának növelésével lassítani tudjuk az algoritmust, hogy ellenálló legyen a nyers erő és a szótár támadásokkal szemben. Só használatával a rendszer biztonságos a szivárványtábla támadásokkal szemben. A sémát összehasonlítom a gyakorlatban használt jelszó hashelő algoritmusokkal a biztonság és a hatékonyság szempontjából.

A dolgozat első részében a jelszótárolás háttérét mutatom be, majd rátérek az elliptikus görbék elméletére. Ezt követően a gyakorlatban használt Argon2 és Bcrypt rendszereket ismertetem és kitérek az általunk készített rendszerre, valamint az implementálására.

Végezetül pedig ezen rendszereket hasonlítom össze. Az algoritmust 3 fő szempont alapján vizsgálom, összehasonlítom futási idő, memória használat és a kódméret szempontjából a manapság elterjedt algoritmusokkal. Bemutatom, hogy algoritmusunk megfelelő jelszó hashelő alternatíva.

2 Biztonságos jelszó tárolás

Kezdetben, mint sok más a jelszavakat is nyíltan, titkosítás, hashelés nélkül tárolták. Ez a megoldás nem biztonságos, hiszen, ha jogosulatlanul hozzáfértek a jelszavakat tároló adatbázishoz, akkor minden felhasználó jelszava illetéktelen kezekbe kerülhetett.

Manapság a jelszavakat nem nyíltan, hanem azok hash értékét tárolják le. A jelszótárolást tekintve a hash függvény alkalmazása előnyös, mivel ennek a függvények tulajdonságai (egyirányú, ősképp ellenállóság) biztosítják, hogy a jogosulatlan felek az adatbázis kiszivárgása esetén se férjenek hozzá a jelszavakhoz.

A felhasználók hitelesítésekor a begépzelt jelszónak a hash értéke lesz összehasonlítva a letárolt hash értékkel. Ha a két érték megegyezik, akkor sikeres az autentikáció.

A támadók dolgát nehezíti, ha minél lassabb a hash kivonatot előállító algoritmus. Lassú hash függvények alkalmazásával nehezebb a támadóknak az úgynevezett offline támadásokkal megtalálni azt a jelszót, aminek a hash értéke megegyezik a letárolt kivonat értékével.

3 Hash függvény

Kriptográfiában a hash függvény $H: \{0,1\}^* \rightarrow \{0,1\}^n$ egy adott véges méretű bitsztringet rendel egy tetszőleges véges hosszúságú bemenethez, olyan módon, hogy egy adott jelszó függvényértéke mindig ugyanaz a hash érték lesz. A hash értékét lenyomatnak is szoktuk hívni.

A kriptográfiai hash függvény tulajdonságai:

- Ősképp ellenálló
- Második ősképp ellenálló
- Ütközésmentes

3.1 Hash függvénnyel szembeni támadások

- Ősképp elleni támadások

Megkülönböztetünk első és második ősképp elleni támadást. Az első ősképp elleni támadás alatt azt értjük, ha a támadó célja az, hogy az y lenyomathoz találjon egy olyan x értéket, hogy $H(x)=y$. Az első ősképp ellenállóság a függvény egyirányúságára utal. Az egyirányúságának köszönhetően nem tudjuk hatékonyan a hash értékből visszaszámolni (invertálni), hogy mi volt a bemenet.

Második ősképp elleni támadásról akkor beszélhetünk, ha a cél az, hogy egy adott x érték esetén olyan $x' \neq x$ értéket tudjunk megadni, melyre $H(x) = H(x')$. A második ősképp ellenálló hash függvényeket gyengén ütközésmentes függvényeknek nevezzük.

- Ütközéses támadások

A támadó célja, hogy meg tudjon határozni két különböző üzenetet, melyek hash értéke megegyezik, azaz találjon egy x és x' értéket, ahol $x' \neq x$ és $H(x) = H(x')$. Azokat a hash függvényeket, amelyek ütközés ellenálló tulajdonsággal rendelkeznek erősen ütközésmentes hash függvényeknek hívjuk ([7]).

A két támadás közötti különbség az, hogy az ütközéses támadások esetén a támadónak nem áll rendelkezésére a lenyomat sem és a bemenetek egyike sem.

- Szivárványtábla támadás

A szivárványtábla egy olyan előre kiszámított szótár, amelyben jelszavak és a hozzájuk tartozó hash láncok vannak letárolva.

Minden lánc egy véletlenszerűen generált jelszóval kezdődik, és a következő lehetséges jelszó az előzőből kerül kiszámításra a hash függvény segítségével. A hash függvény tartománya és a jelszavak tartománya általában különbözik, ezért a hash értékeket egy redukciós függvény segítségével kell átalakítani az adott jelszó tartományra.

Az i -edik lánc k hash értékkel a H hash-függvény és a redukciós R függvénnyel a következőképpen írható fel

$$p_{i,1} \xrightarrow{H} c_{i,1} \xrightarrow{R} p_{i,2} \xrightarrow{H} c_{i,2} \xrightarrow{R} p_{i,3} \rightarrow \dots \rightarrow p_{i,k} \xrightarrow{H} c_{i,k}$$

ahol csak az egyes láncok első és utolsó elemeit kell tárolni. Ahhoz, hogy egy jelszót megtaláljunk az adott hash értékhez $\tilde{c}_1$ a következő láncot kell végig számolni:

$$\tilde{c}_1 \xrightarrow{R} \tilde{p}_2 \xrightarrow{H} \tilde{c}_2 \xrightarrow{R} \tilde{p}_3 \rightarrow \dots \rightarrow \tilde{p}_j \xrightarrow{H} \tilde{c}_j$$

Ha a lánc $c_{i,k}$ -ra végződik, ahol $\tilde{c}_j = c_{i,k}$ bizonyos i esetén, akkor

$$h^{-1}(\tilde{c}_1) = p_{i,k+1-j}$$

Így egyszerűen kideríthető, hogy melyik jelszóhoz milyen hash érték tartozik. A táblák létrehozása időigényes, de amint rendelkezésre áll nagyon gyorsan lehet benne keresni. Megfelelő védelmet nyújt az ilyenfajta támadással szemben a jelszavak sózása.

Minden lehetséges jelszóhoz az összes lehetséges só értéket figyelembe kell venni, ezért minden lehetséges sóhoz egy táblázatot kell készíteni, 12 bites sóhoz 4096 különböző táblázat szükséges. A só méretének a növelésével a szivárványtáblák generálása

kivitelezhetetlen. Azonban, ha csak egyetlen sóértéket alkalmazunk minden felhasználói jelszóhoz, akkor a szivárványtábla generálása lehetséges. Éppen ezért fontos, hogy különböző jelszavakhoz különböző sóérték legyen használva.

- Brute force támadás

Jelszó alapú hitelesítés esetén, feltéve, hogy az egyirányú hash függvény inverzének létrehozása technikailag lehetetlen, egy hash tartomány elemének meghatározása egy hash értékhez úgy lehetséges, hogy egy adott tartományban az összes lehetséges bemeneti karakterlánc - egyszerű szöveges jelszó – hash értékét teszteljük, és megpróbáljuk megtalálni a hash értékek egyezését. Ha az egyik bemeneti karakterláncnak ugyanaz a hash értéke, mint a jelszóhoz tároltnak, akkor az is használható a sikeres hitelesítéshez, ha nem is egyezik pontosan az eredeti jelszóval.

- Szótár támadás

A táblákat jelszófájlokból állítják össze, úgy, hogy különböző transzformációkat hajtanak végre a jelszavakon. A jelszavakat megismétlik, kisbetűssé/nagybetűssé alakítják, tükrözik stb. majd a megfelelő algoritmussal hashelik és az eredményt összevetik jelszófájlban tárolt értékekkel. Offline módon használják felhasználók jelszavainak megfejtésére, így a támadónak kötetlen idő áll a rendelkezésére. A támadás ellen megfelelő jelszóválasztással lehet védekezni. Célszerű olyan jelszót választani, ami nem egy értelmes szó, nem lehet transzformációkkal értelmes szóvá alakítani.

Másrészt egy adott hash érték inverzének megtalálása nyers erő támadással vagy szótár támadással gyors és sikeres lehet, a só értéke nem kellőképpen lassítja a támadást. A hash függvények számítását lassítani tudjuk azáltal, hogy többkörösek lesznek az algoritmusok. Minél több a körök száma, annál lassabb a hash számítás.

3.2 Só

Előfordulhat, hogy az adatbázisban két vagy akár több felhasználó is egyforma jelszót adott meg. A hash algoritmusának köszönhetően a kivonatuk is meg fog egyezni. Ezáltal a támadó egyszerre több felhasználó jelszavát is meg tudja szerezni. Itt kerül a képbe a jelszó sózása.

A jelszó sója egy véletlen sztring, amit hozzáfűznek a jelszóhoz, aminek a segítségével az azonos jelszavakhoz képzett kivonat érték különböző lesz.

A sózás offline támadások (szótár, szivárványtábla támadások) esetén megnehezíti a támadó dolgát. Abban az esetben, ha ugyanazt a sót adjuk a jelszavakhoz, csak egy hosszabb jelszót hozunk létre. Ha két felhasználó ugyanazt a jelszót használja, akkor az azonos sóval együtt is ugyanaz lesz a jelszavuk, ezáltal a hash értékük is meg fog egyezni. Azonban, ha a két megegyező jelszóhoz különböző sót használunk, akkor két egyedi jelszót fogunk kapni, eltérő hash értékekkel.

Ha a só minden felhasználónál egyedi, akkor a támadónak megnehezítjük a dolgát, hiszen most már minden egyes felhasználóhoz egy hash táblázatot kell kiszámítania. Következésképpen a só alkalmazása jelentősen lassítja a támadást. A támadó képes lehet feltörni egy felhasználó jelszavát, azonban az összes jelszó feltörése kivitelezhetetlen.

3.3 Többkörösítés

A hash algoritmusok alkalmazásánál bizonyos esetekben fontos, hogy lassú legyen. Lassítani az algoritmuson többféleképp lehet, például minél többször futtatjuk, annál lassabb lesz. Ezáltal a támadó dolgát nehezítjük, ugyanis lassabban tudja kiszámítani a kivonatokat.

4 Elliptikus görbék

Az elliptikus görbe kriptográfia (ECC) a nyilvános kulcsú kriptográfia egy fajtája, amely a véges testek feletti elliptikus görbék matematikai tulajdonságain alapszik és ma egyre szélesebb körben használt.

4.1 Általánosan az elliptikus görbékről

Egy F test feletti E elliptikus görbét az alábbi (Weierstrass) egyenlet definiál:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

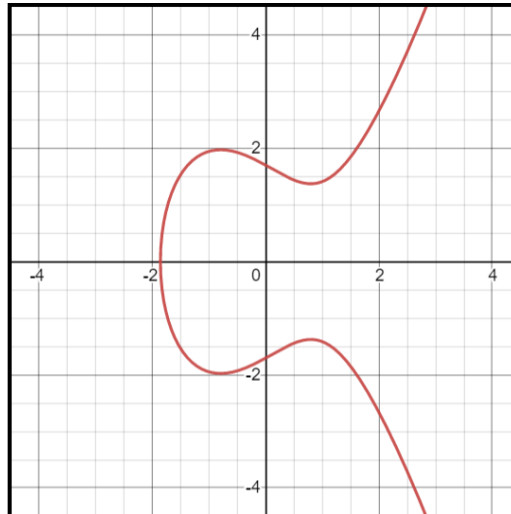
ahol $a_1, a_2, a_3, a_4, a_6 \in F$, $\Delta \neq 0$, ahol Δ az E diszkriminánsa.

Az F test feletti E elliptikus görbe pontjait a fenti egyenletet kielégítő pontok alkotják. Továbbá tartozik a görbéhez egy további pont is, az O végtelen távoli pont, ami rajta van minden az y tengellyel párhuzamos egyenesen, és az x tengelyre vonatkozó tükröképe önmaga.

A fenti egyenlet az F test karakterisztikájától függően felírható rövid Weierstrass egyenletként. Amennyiben a test karakterisztikája nem 2 vagy 3, akkor az alábbi egyenletet írhatjuk fel:

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

ahol $a, b \in F$, $\Delta = -16(4a^3 + 27b^2)$.



1. ábra Elliptikus görbe ([6])

4.2 Véges testek feletti elliptikus görbék

Ha véges testek felett definiálunk elliptikus görbét, akkor a görbére a projektív síkon véges sok pont illeszkedik. Mivel véges sok pont van, így kriptográfiai szempontból egyszerűen tudunk velük számításokat végezni.

Tekintsük az E elliptikus görbét a Z_p véges test felett, ahol p prím. Ekkor a következő egyenlet határozza meg az elliptikus görbét:

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

ekkor, ha az (x, y) koordinátákat behelyettesítjük az egyenletbe és egyenlőséget kapunk $\mod p$, akkor a pont a görbe pontja. O továbbra is egy végtelen távoli pont és $a, b \in Z_p$. A kriptográfiai alkalmazásoknál elvárás továbbá, hogy a görbe ne legyen szinguláris, azaz $4a^3 + 27b^2$ ne legyen 0.

4.3 Művelet

Az elliptikus görbe pontjai Abel-csoportot alkotnak a pontok összeadása műveletre nézve, ahol az egységelem a végtelen távoli pont. A következőkben legyen E a valós számok felett definiált elliptikus görbe és tekintsük a P és Q görbepontokat.

4.3.1 Geometriai leírás

Legyen $P = (x_P, y_P), Q = (x_Q, y_Q), P \neq Q, P \neq -Q, Q \neq -P$ két pont a görbén.

- $R = (x_R, y_R) = P + Q$

Ekkor húzzuk meg a P, Q két pontot összekötő egyenest, jelöljük R' -vel azt a pontot, ahol az elmettszi a görbét. A két pont összege az R' pont x -tengelyre vett tükörképe lesz.

- $P = -Q$ vagy $Q = -P$ esetben $P + (-P) = O, Q + (-Q) = O$
- $P \neq Q$ ebben az esetben Q -n és P -n átmenő egyenes a görbe húrja.
- $R = (x_R, y_R) = 2P$

Ekkor vegyük a pontra illeszkedő érintőt, és jelölje R' azt a pontot, ahol az egyenes elmettszi a görbét. A pont duplázásának az eredménye az R' pont x -tengelyre tükrözött képe lesz.

4.3.2 Algebrai leírás

Vegyünk egy F testet, melynek nem 2 vagy 3 a karakterisztikája.

1. Egységelem: O

$$P + O = O + P = P, \quad \forall P \in E(K)$$

2. Additív inverz: $P = (x_P, -y_P)$

$$P = (x_P, y_P) \in E(K), (x_P, y_P) + (x_P, -y_P) = O$$

3. Pontok összeadása: $R = P + Q = (x_R, y_R)$

$$P = (x_P, y_P) \in E(K), Q = (x_Q, y_Q) \in E(K), P \neq \pm Q$$

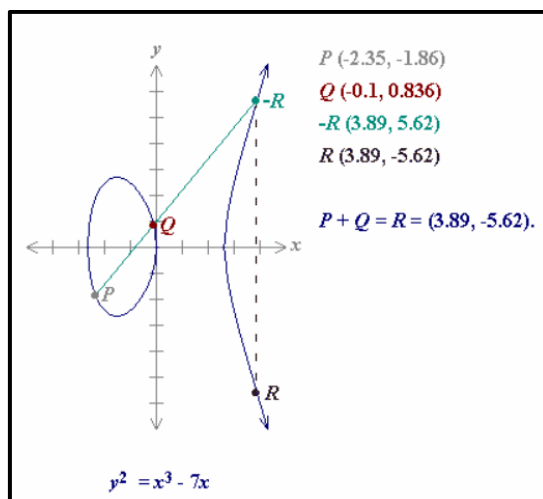
$$x_R = \left(\frac{y_Q - y_P}{x_Q - x_P} \right)^2 - x_P - x_Q$$

$$y_R = \left(\frac{y_Q - y_P}{x_Q - x_P} \right) (x_P - x_R) - y_P$$

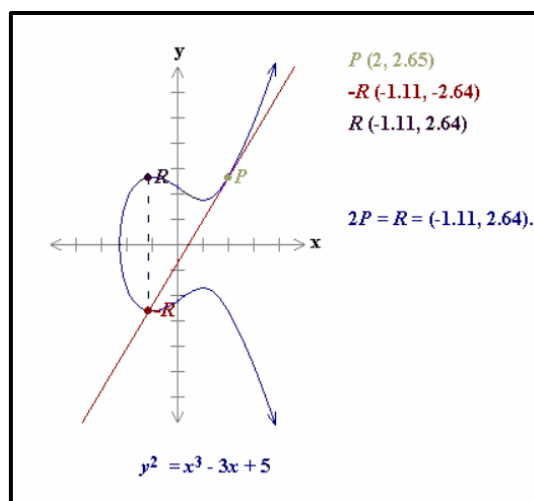
4. Pontok duplázása: $R = 2P = (x_R, y_R)$

$$x_R = \left(\frac{3x_P^2 + a}{2y_P} \right)^2 - 2x_P$$

$$y_R = \left(\frac{3x_P^2 + a}{2y_P} \right) (x_P - x_R) - y_P$$



2. ábra Elliptikus görbe pontok összeadása ([15])



3. ábra Pont duplázása ([15])

Az általam implementált algoritmus több pontján is véges testekben történő algebrai műveletek eredményeire van szükség. Míg az összeadás, kivonás és az „egyszerű” szorzás rendszerint igen gyors, addig a nagy számok gyors hatványozása már komolyabb algoritmust igényel. Az elkészült kódban az ún. Sliding Window algoritmust használjuk erre a célra.

Sliding Window algoritmus

Véges testbeli hatványozásra implementáltam a Sliding Window algoritmust ([4]). Ez a stratégia a nullás szavak átlagos számának növelését célozza meg, miközben viszonylag nagy d ablak méretet használ. Ezáltal kevesebb szorzás műveletet kell elvégezni.

A metódus a kitevőnek, későbbiekben E , veszi a bináris megfelelőjét, majd a legalacsonyabb helyiértékű bittől felosztja a kitevőt NW (nem nulla ablak) és ZW (nulla ablak) részekre egy d ablak méret segítségével. A k ablakok száma nem lehet egyenlő n/d -vel, ahol n az E hossza.

Sliding Window algoritmus

Input: x, E

Output: $y = x^E$

1. Számoljuk ki és mentjük x^w minden $w = 3, 5, 7 \dots 2^d - 1$
 2. E felbontása nulla és nem nulla ablakokra F_i , aminek hossza $L(F_i)$
 3. $y := x^{F_{k-1}}$
 4. **for** $i = k - 2$ **down to** 0
 - 4a. $y := y^{2^{L(F_i)}}$
 - 4b. **if** $F_i \neq 0$ **then** $y := y * x^{F_i}$
 5. **return** y
-

A programban a konstans hosszú nem nulla ablak (CLNW) algoritmust implementáltam. Itt az algoritmus minden lépésben egy nulla ablakot (ZW) vagy nem nulla ablakot (NW) alakít ki.

ZW: Megnézi a bejövő bitet, amennyiben a bit 0, akkor marad ZW, egyébként átmegy NW módra.

NW: Addig marad NW módban, amíg d bitet nem gyűjt be. Utána megnézi a bejövő bitet, ha az 0, akkor átmegy ZW módba, egyébként NW módban kezd.

Ez a technika tetszőleges hosszúságú nulla ablakokat, és d hosszúságú nem nulla ablakokat hoz létre, ugyanis két szomszédos ZW konkatenálható.

1996 = 011111001100				
i	F_i	$L(F_i)$	4a	4b
4	011	3	x^3	x^3
3	111	3	$(x^3)^8 = x^{24}$	$x^{24} * x^7 = x^{31}$
2	0	2	$(x^{31})^2 = x^{62}$	x^{62}
1	011	3	$(x^{62})^8 = x^{496}$	$x^{496} * x^3 = x^{499}$
0	00	2	$(x^{499})^4 = x^{1996}$	x^{1996}

1. táblázat Sliding Window metódus példa

4.4 Elliptikus görbe diszkrét logaritmus probléma (ECDLP)

Az ECDLP egy olyan matematikai probléma, amelyről azt feltételezzük, hogy nehéz megoldani.

Legyen E egy Z_p feletti elliptikus görbe, ahol p prím, jelölje $E(Z_p)$ az elliptikus görbére illeszkedő pontokat. Legyen $G \in E(Z_p)$ pont, rendje n és legyen $A \in \langle G \rangle$. Az elliptikus görbe diszkrét logaritmus probléma A, G ismeretében a meghatározása, ahol $A = aG$. Az a egészt A elem G alapú diszkrét logaritmusának nevezzük.

Sejthető, hogy kellően nagy csoport esetén, a P és az aP pontok ismeretében, nincs hatékony módszer az a érték meghatározására, hisz a támadónak a kiszámításához a diszkrét logaritmus problémát kellene megoldania.

Az ECC-n alapuló kriptográfiában számtalan algoritmus támaszkodik az ECDLP nehézségére.

4.5 Bilineáris párosítás

Az elkészült konstrukció a bilineáris leképezések használatán alapul. Legyen n egy prímszám, G_1 egy additív n rendű csoport, melynek egységeleme O , és legyen G_2 egy n rendű multiplikatív csoport, melynek egységeleme 1 . Az elliptikus görbe alapú kriptográfiánál ugyanolyan rendű, ciklikus csoportokkal dolgozunk, így a bilineáris párosítás alábbi definícióját használjuk.

Definíció: Egy bilineáris párosítás alatt azt az

$$e: G_1 \times G_1 \rightarrow G_2$$

leképezést értjük, amely kielégíti a következő feltételeket:

1. Bilineáris, azaz minden $x, y, z \in G_1$ esetén:

$$e(x + y, z) = e(x, z) * e(y, z)$$

$$e(x, y + z) = e(x, y) * e(x, z)$$

2. Nem elfajuló, azaz létezik olyan $x, y \in G_1$ hogy $e(x, y) \neq 1$.
3. Hatékonyan kiszámítható: Létezik olyan algoritmus, amely polinomiális idő alatt kiszámítja $e(x, y)$ értékét bármely $x, y \in G_1$ esetén.

Ilyen párosítás például a Weil és a Tate párosítás.

4.5.1 Tate párosítás

4.5.1.1 R-torzió

Az elliptikus görbéken értelmezett párosítások során szükségünk lesz az ún. r-torziós csoport fogalmára.

Definíció: egy E elliptikus görbe r-torziós csoportján (jel. $E[r]$) az alábbi halmazt értjük

$$E[r] = \{P \in E \mid r[P] = O\}.$$

Az r-torziós csoport $r + 1$ ciklikus részcsoporthból áll, ám általában nem igaz az, hogy ezen részcsoporthok összes pontja pontja a görbének is. Így a párosítás során szükségünk van $\mathbb{Z}_p$ egy olyan legszűkebb) $\mathbb{Z}_{p^k}$ bővítésére, mely esetén teljesül, hogy $E(\mathbb{Z}_{p^k})$ tartalmazza az egész r-torziós csoportot. Ezt a k pozitív egész számot a *beágyazás fokának* nevezzük és adott r és q esetén $r \mid q^k - 1$ összefüggésből könnyedén meghatározhatjuk, úgy, hogy a legkisebb olyan k szám mely kielégíti ezt az oszthatóságot.

4.5.1.2 Divizorok

Ahhoz, hogy a Tate párosítást megértsük, szükségünk lesz néhány fogalom megismerésére. Látni fogjuk, hogy a divizoroknak fontos jelentősége lesz a párosításoknál, így célszerű a bemutatást ennek definíciójával kezdeni.

Egy E elliptikus görbe D divizora alatt a

$$D = \sum_{P \in E(\overline{\mathbb{Z}_p})} n_P(P)$$

formális összeget értjük, melyre teljesül, hogy csupán véges sok $n_P \in \mathbb{Z} \neq 0$.

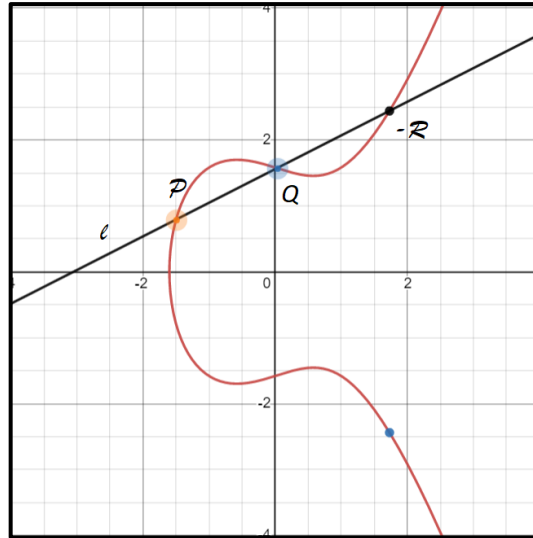
A divizorok halmaza összeadásra nézve csoportot alkot (jel. $Div_{\overline{\mathbb{Z}_p}}(E)$), melynek egységeleme az ún. zéró divizor (azon divizor, melynél az összes n_P együttható 0).

Egy divizornak a foka a benne szereplő együtthatók összege, azaz $Deg(D) = \sum n_P$, továbbá a supportja $supp(D) = \{P \in E(\overline{\mathbb{Z}_p}) : n_P \neq 0\}$.

Divizorokat függvények esetében is tudunk értelmezni. Egy elliptikus görbe pontjain értelmezett f függvény divizora alatt az

$$(f) = \sum_{P \in E(\overline{\mathbb{Z}_p})} ord_P(f)(P)$$

formális összeget értjük, ahol $ord_P(f)$ az f függvény P -ben vett multiplicitása. A függvény divizora így jól jellemzi E és f metszéspontjait és ezen pontok multiplicitását.



4. ábra Divizor példa

Az ábra segítségével láthatjuk, hogy pl. az l egyenes divizora nem más, mint az egyenes és a görbe metszéspontjainak „összege” megfelelő multiplicitással:

$$(l) = 1P + 1Q + 1(-R) - 3O$$

Amennyiben E egy D divizora esetén létezik olyan f függvény, hogy $(f) = D$, úgy D -t ún. fődivizornak nevezzük. Minden fődivizor egyben zéró divizor is.

Amennyiben létezik olyan f függvény, amelyre $D_1 = D_2 + (f)$, akkor azt mondjuk, hogy a két divizor ekvivalens és $D_1 \sim D_2$ -vel jelöljük.

Tate párosítás

A továbbiakban jelöljük $\#S$ -el az S halmaz (csoport, részcsoporthat.) elemszámát, illetve egy G csoport esetén G^* -al a csoport multiplikatív részcsoporthatját. Ezen jelölések mellett a Tate párosítást az alábbiak szerint definiálhatjuk.

Definíció: Rögzített q prím és r pozitív egész esetén legyen $P \in E(Z_{p^k})[r]$ egy pont az r -torziós csoportból. Ekkor létezik olyan f függvény, melyre teljesül, hogy $(f) = r(P) - r(O)$. Válasszuk ki továbbá az $E(Z_{p^k})/rE(Z_{p^k})$ ekvivalencia osztály egy tetszőleges $Q \in E(Z_{p^k})$ reprezentánsát és egy olyan Z_{p^k} felett értelmezett D_Q zéró divizort, mely egyrészt ekvivalens a $(Q) - (O)$ divizorral, másrészt $\text{supp}(D) \cap \text{supp}(f) = \emptyset$. Ekkor Tate párosítás alatt azon

$$t_r : E(Z_{p^k})[r] \times E(Z_{p^k})/rE(Z_{p^k}) \rightarrow Z_{p^k}^*/(Z_{p^k}^*)^r$$

leképezést értjük mely az alábbiak szerint van definiálva:

$$t_r(P, Q) = f(D_Q)$$

Tekintsük meg egy konkrét példán keresztül ([5]). Legyenek $q = 5$, $E(Z_p) : y^2 = x^3 - 3$, $r = 3$ (így a korábbi összefüggés alapján $k = 2$), a választott $E[r]$ -beli P pontunk pedig legyen a $(3, 2)$ pont. Legyen továbbá $Q = (i + 1, 4i + 2) \in E(F_{q^k})$. Látható, hogy az $f : y + 2x + 2 = 0$ függvénynek E -re vett divizora a $3(P) - 3(O)$, így a párosítás kiszámításához egy olyan $D_Q \sim (Q) - (O)$ divizorra van szükségünk, melyre teljesül, hogy $\{P, O\} \notin \text{supp}(D_Q)$

Válasszunk egy tetszőleges R pontot (legyen most $R = (2i, i + 2)$), és legyen $D_Q = (Q + R) - R$, ahol $(Q + R) = (3i + 1, 2)$.

Ekkor a Tate párosítás kiszámolható az alábbiak szerint:

$$t_r(P, Q) = f(D_Q) = \frac{f(Q + R)}{f(R)} = \frac{2 + 2 * (3i + 1) + 2}{(i + 2) + 2 * 2i + 2} = 4i + 4$$

A fő probléma a Tate párosítás kapcsán, hogy mivel R -et véletlenszerűen választottuk, így a számolás végén kapott eredmény (a fenti példában ez $4i + 4$) nem egy egyedi elem, hanem egy ekvivalencia osztály egy reprezentánsa. Emiatt maga a párosítás kriptográfiai célokra nem használható, hiszen ott biztosítanunk kell, hogy különböző elemek esetén a párosítás mindig egyedi értéket adjon vissza, ne csupán egy „halmaz” egy „tetszőleges” elemét.

A fenti probléma orvoslására találták ki az ún. redukált Tate párosítást.

Redukált Tate párosítás

Definíció: A Tate párosításnál használt jelölések megtartása mellett jelölje μ_r Z_{p^k} r -edik egységgyökeinek csoportját. Ekkor redukált Tate párosítás alatt azt a

$$T_r : E(Z_{p^k})[r] \times E(Z_{p^k})/rE(Z_{p^k}) \rightarrow \mu_r$$

leképezést értjük, melynek definíciója

$$T_r(P, Q) = t_r(P, Q)^{\#Z_{p^k}/r} = f_{r,P}(D_Q)^{\frac{p^k-1}{r}}$$

A fenti definíció jobb oldalán található $\frac{p^k-1}{r}$ -edik hatványozásnak köszönhetően ez a leképezés nem egy ekvivalencia osztály egy (véletlenszerű) reprezentánsát adja vissza, hanem egy egyedi egységgyököt, így ez a redukált Tate párosítás alkalmazható kriptográfiai célokra.

5 Gyakorlatban használt rendszerek

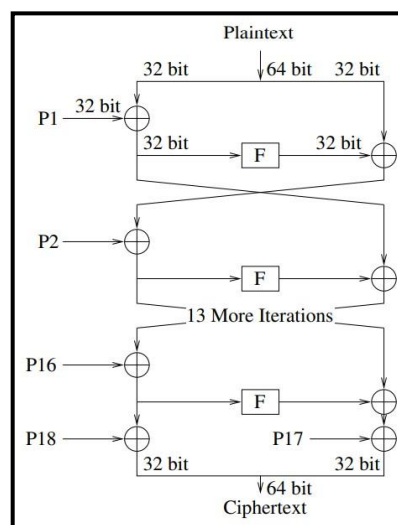
5.1 Bcrypt

A Bcrypt ([11]) egy Blowfish alapú hash függvény jelszótároláshoz, melyet Niels Provos és David Mazières tervezett 1999-ben. Az OpenBSD alapértelmezett hash algoritmusaként válhatott ismertté. A Bcrypt 128 bites sót használ, kimenete pedig 192 bites hash érték. Fontos jellemzője, hogy a hash számításának költsége állítható, így gyengébb jelszavak esetén csökkenti az offline támadások hatékonyságát, így növelve a biztonságot.

5.2 Blowfish

A Blowfish-t Bruce Schneier tervezte 1993-ban, ami egy szimmetrikus kulcsú blokktitkosító. Általános célú algoritmusként lett megtervezve, amelyet a DES alternatívájának szántak.

64 bites blokktitkosító, 16 körös Feistel struktúrát alkalmaz. Működése során 18 darab (P1-P18) 32 bites körkulcsot használ. A titkosítás első lépése, hogy a 64 bites bemenő inputot kettő felé vágja, amelyeken a titkosítás zajlik több körben. A kulcsok mérete 32 bit és 448 bit közöttiek lehetnek. Emellett 4 darab S-boxot alkalmaz, amik 256 darab 32 bites szóból állnak.



```

EksBlowfishSetup (cost, salt, key)
  state ← InitState ()
  state ← ExpandKey (state, salt, key)
  repeat (2cost)
    state ← ExpandKey (state, 0, salt)
    state ← ExpandKey (state, 0, key)
  return state

```

6. ábra EksBlowfishSetup algoritmus ([11])

Az eksblowfish egy olyan algoritmus, amely ugyanúgy titkosít, mint a Blowfish. Azonban különbözően transzformálja a kulcsokat körkulcsokká, illetve az S-boxok előállításában is különbözik. Három bemeneti paraméterrel rendelkezik, szüksége van a költségre, egy 128 bites sóra és a kulcsra. A költség paraméterrel szabályozhatjuk, hogy milyen drága legyen a kulcs előállítása. A kulcs paraméter egy titkos kulcs, ami legfeljebb 56 byte hosszú jelszó lehet.

Az InitState lépésben a P_i számjegyeivel inicializálja az S-boxokat és a körkulcsokat. Az ExpandKey metódus a P-tömbökön és az S-boxokon dolgozik a só és a kulcs a segítségével XOR műveleteket végrehajtva. Többkörösítést az ExpandKey(state, 0, salt) és az ExpandKey(state, 0, key) függvények biztosítják, amelyek a P-tömbökön és az S-boxokon végzett műveletekkel lassítják az algoritmust.

5.3 Bcrypt működése

```

bcrypt (cost, salt, pwd)
  state ← EksBlowfishSetup (cost, salt, key)
  ctext ← "OrpheanBeholderScryDoubt"
  repeat (64)
    ctext ← EncryptECB (state, ctext)
  return Concatenate (cost, salt, ctext)

```

7. ábra Bcrypt algoritmus ([11])

A Bcrypt két fázisban működik. A költséges kulcsgeneráló algoritmus lefutása után elkezdődik a titkosítás. Az „OrpheanBeholderScryDoubt” 192 bites sztring titkosítást ismételi 64 alkalommal Electronic Code Book (ECB) módban. Az ismétlés végén konkatenáljuk a költséget, a sót, illetve az előállított titkosított üzenetet.

Tekintsünk egy Bcrypt futást. Az alábbi táblázatban látszik, hogy milyen végeredményt kapunk egy jelszó kivonatolása után.

Jelszó	1234
Költség	$2^4 = 16$
Futási idő	416261799
Hash	\$2a\$04\$TIE8xwoTOTGrGJp9yxEEO.2qmeBuCQEWnT1tpOSLwoDCCSjrqFita

2. táblázat Bcrypt példa

A hash sztring négy részből tevődik össze, „\$” jelekkel elválasztva. Az első rész az algoritmust azonosítja, a második részben a költség jelenik meg. A harmadik rész a 128 bites só, majd az utolsó 192 bit a hash érték base-64 kódolásban.

5.3.1 Tesztelés

Költség	Körök száma	Idő (Nanoszekundum) Kulcsok, S-boxok számítása	Idő (Nanoszekundum) Teljes hash számítás
4	16	1432000	2589000
5	32	3390100	4399700
6	64	5328100	6370700
7	128	14634900	23421700
8	256	23826100	25586400
9	512	44562900	45759200
10	1024	88752900	89709500

3. táblázat Bcrypt teszt futtatás

Látható a fenti táblázatban, hogy a körök számának a duplázásával közel lineárisan nő a hash számításához szükséges idő is.

5.4 Argon2

Az Argon2 algoritmust ([1]) 2015 júliusában a Password Hashing Competition verseny győztesének választották. Az Argon2 tervezői Alex Biryukov, Daniel Dinu és Dmitry Khovratovich a Luxemburgi Egyetemről.

5.4.1 Argon2 változatok

- **Argon2i**
 - Biztonságos a side-channel támadás ellen
 - Adatfüggetlen memória-hozzáférés
 - Kriptovaluták és Proof of Work (PoW) alkalmazások számára ideális
- **Argon2d**
 - Biztonságos GPU cracking ellen
 - Adatfüggetlen memória-hozzáférés
 - Jelszó hash készítéshez
- **Argon2id**
 - Biztonságos time-memory tradeoff ellen

5.4.2 Argon2 működése

Paraméterek:

- Memória mérete (m): Az algoritmus által használt memória. Amennyiben ez az érték magas, akkor a támadó számára a hash számítás költségesebbé válik.
- Iterációk száma (t): A memória fölötti iterációk száma, 1 és $2^{32}-1$ között. A végrehajtási idő lineárisan korrelál ezzel a paraméterrel. Ez lehetővé teszi az egy hash kiszámításához szükséges számítási költségek növelését.
- Párhuzamosság (p): A használandó szálak száma, értéke 1 és $2^{24}-1$ között lehet.
- Só hossza: Az Argon2 szerzői javasolják, hogy ez a paraméter legyen 128 bites, de szerintük korlátozott tárhely esetén 64 bitre lecsökkenthető.
- Kulcs hossza (azaz a Hash hossz): Ez a paraméter a tervezett felhasználástól függ, értéke 8 és $2^{32}-1$ byte között lehet. Az Argon2 algoritmus szerzői szerint a legtöbb alkalmazáshoz 128 bites értéknek elegendőnek kell lennie.

Az Argon2 egy belső G tömörítési algoritmust használ két 1024 bájtos bemenettel és egy 1024 bájtos kimenettel, valamint egy belső H hash függvényt. A H egy Blake2b hash függvény, a G pedig annak belső permutációján alapul. Az Argon2 működési módja nagyon egyszerű, ha nem használunk párhuzamosságot, ekkor a G algoritmust m-szer iteráljuk.

5.4.3 Tesztelés

Argon2i						
Szálak száma	1			2		
Memória	131072	262144	524288	131072	262144	524288
Iteráció	Idő	Idő	Idő	Idő	Idő	Idő
2	875743200	1059662300	1827941500	361103700	846235200	1414412200
3	942086400	1075460000	2430936600	516697300	795753800	1393714600
4	775274300	1466300300	2743776600	597711100	915937900	1815080300
5	821227000	1754462000	3804135500	520962900	1172931400	2183486700

4. táblázat Argon2i teszt futtatás eredménye

Argon2d						
Szálak száma	1			2		
Memória	131072	262144	524288	131072	262144	524288
Iteráció	Idő	Idő	Idő	Idő	Idő	Idő
2	666674700	1172667600	2151705300	565867800	990334900	1272213600
3	791439100	1098616700	2679042900	500668100	965157600	1415338400
4	780146100	1377080100	3522017000	611348400	949428800	1880702600
5	958596800	1671582701	3803867900	503414300	1171236700	2188742700

5. táblázat Argon2d teszt futtatás eredménye

Argon2id						
Szálak száma	1			2		
Memória	131072	262144	524288	131072	262144	524288
Iteráció	Idő	Idő	Idő	Idő	Idő	Idő
2	638043300	1238200500	1997100000	438096600	826644600	1519782300
3	825073400	1250740400	2121949000	489450000	906163800	1432253900
4	766652700	1493119700	2812854800	626704300	983229200	1852801200
5	904269300	1836465300	3509406000	623857100	1191244000	2280329500

6. táblázat Argon2id teszt futtatás eredménye

6 Tervezett rendszer

A dolgozat célja az volt, hogy tervezzünk egy hatékony jelszótároló algoritmust. Az implementált algoritmus bilineáris leképezés segítségével hasheli a jelszavakat.

Implementáláskor megvizsgáltuk a bilineáris leképezés jelszó tárolásra való alkalmazásának lehetőségét. A tervezés során figyelembe vettük a Password Hashing Competition (PHC) verseny elvárásait is.

A Password Hashing Competition ([13]) nevű nyílt versenyt 2013-ban hirdették meg, abból a célból, hogy egy vagy több jelszó hashelő algoritmust fogadjanak el ajánlott szabványként. Végül 2015-ben az Argon2 algoritmust választották végső győztesként.

A Password Hashing Competition versenyen az alábbi elvárásokat vették figyelembe:

- Képes legyen legalább 0 és 128 bájt közötti jelszavakat kezelni, függetlenül a kódolástól.
- A só hossza legalább 16 bájt legyen.
- A kimenet hossza legalább 32 bájt legyen.
- Egy vagy több paraméter legyen állítható, az idő (t_cost) és/vagy a memóriaigény (m_cost).

Ezeknek a kritériumoknak az algoritmusunk megfelel. Bármilyen hosszú jelszót képes leképezni 512 bit hosszúra, hiszen a legelső lépés során a prímtestbe képezzük a megadott jelszót. A só, amit használunk egy pont, ami meghaladja a 16 bájtot. Végül a kapott kimenet szintén megfelel a kritériumoknak. Esetünkben az állítható paraméter az idő, amit a körök számával tudunk szabályozni.

Mindezek mellett egy olyan rendszert terveztünk, amely illeszkedik a manapság legelterjedtebb jelszó alapú hitelesítő protokollokhoz. Tehát a felhasználó megadja a jelszót és szerver oldalon a szózott hash értéke tárolódik le. A szózásnak köszönhetően egy egyedi hash értéket fognak kapni a felhasználók.

Továbbá az volt a célunk, hogy egy olyan lassú hash algoritmust tervezzünk, ami többkörös, azaz bármikor lehet az igényeknek megfelelően lassítani, a körök számának módosításával.

Megvizsgáltuk az algoritmus memória felhasználását is és összevetettük a gyakorlatban használt algoritmusokéval, illetve a kód méretét is összehasonlítottuk. Az eredmények azt mutatják, hogy mindkettő értéke kevesebb lett a vizsgált algoritmusokhoz képest.

6.1 Az algoritmus

Egy olyan többkörös algoritmust szeretnénk volna, amely megállja a helyét a manapság népszerű algoritmusok mellett. Ellenben a mi rendszerünk bilineáris leképezésen alapul.

A javasolt algoritmus hash értéket generál egy adott jelszóhoz és sóhoz. Az algoritmus két fő részből tevődik össze. Az első fő része, egy elliptikus görbecsoportba történő hashelő algoritmus. Ahhoz, hogy jelszóból görbepontot tudjunk előállítani kerestünk egy megfelelő algoritmust, amit a ([10]) cikkben leírtak alapján implementáltam. Erre az algoritmusra szükségünk van minden további körben is, ugyanis a többkörösítés miatt szükség volt arra, hogy elliptikus görbe pontot tudjunk előállítani a bilineáris párosítás eredményeként kapott multiplikatív csoportbeli értékhez is.

A másik fő része az algoritmusnak a bilineáris leképezés használata. A bilineáris leképezés bemenete az első körben a jelszóból előállított görbe pont és a só, ami szintén egy görbe pont. A további körökben a bemenet az előző kör eredményeként kapott multiplikatív csoportbeli értékhez rendelt görbe pont és a jelszóhoz tartozó só.

Az általunk használt elliptikus görbe a következő: $y^2 = x^3 + x$, ami egy szuperszinguláris görbe. A használt prím:

$q=7313295762564678553220399414112155363840682896273128302543102778210584118$
 $101444624864132462285921835023839111762785054210425140241018649354445745491$
 039387 . A rendje: $730750818665451459101842416358141509827966402561$.

Az általam implementált algoritmus bemenete a jelszó. A jelszóhoz tartozó só jelen leírás szerint egy elliptikus görbe csoport eleme. A kódban a már bemutatott redukált Tate párosítást használjuk, aminek a bemenete két görbe pont. A pontok közül egyik a só, a másik a hashToCurve metódus alapján előállított görbepont. A kódban a párosításhoz kapcsolódó kódot egy előző szakdolgozó, Nyakacska Lilla készítette ([12]), amit később továbbfejlesztett Kovács Szabolcs egy cikkhez ([9]), végül ezt használtam fel. A bilineáris párosítás további implementációja a ([3]) cikkben szerepel.

Algoritmus

Input: password

Output: (PswStore, S)

1. $E(Fp)$ elliptikus görbe inicializálása
 2. S inicializálása
 3. $PswStore = Convert(password)$
 4. **for** $i = 0$ **up to** körök száma
 - 4a. $R = hashToCurve(PswStore)$
 - 4b. $PswStore = TateParositas(p, S)$
 - 4c. $PswStore = Convert(PswStore)$
- return** (PswStore, S)
-

6.1.1 Elliptikus görbecsoportba történő hashelés

Rögzített q prím és $n > 0$, a, b egész számok mellett tekintsük az F_{q^n} feletti $E_{a,b}: Y^2 = X^3 + aX + b$ elliptikus görbét.

Ahhoz, hogy a hashelés működjön szükség van arra, hogy egyrészt a görbe szuperszinguláris legyen, másrészt pedig teljesüljön, hogy $q > 3$ és $q^n \equiv 2 \pmod{3}$. Szükséges továbbá az is, hogy az $F_q: x \rightarrow x^3$ hozzárendelés legyen bijekció. Amennyiben ezen feltételek teljesülnek, úgy lehetőségünk nyílik a ([10]) cikkben bemutatott parametrizáció és konstrukció használatára.

SAGE matematikai programcsomag segítségével ([14]) ellenőriztük, hogy az általunk használt görbe szuperszinguláris. A mi esetünkben $a=1, b=0$, míg n értéke 1 volt, így a $q \equiv 2 \pmod{3}$ feltétel is teljesül, továbbá amennyiben egy csoport rendje és egy k pozitív egész relatív prímek, úgy az $x \rightarrow x^k$ leképezés bijekció, így a fenti kongruencia miatt az utolsó feltétel is teljesül, hisz F_q^* rendje $q-1$, ami relatív prím a 3-hoz. A cikk szerzője által ismertett parametrizáció és a görbére való hashelés a következő elven történik:

A szerző először definiálja az $f_{a,b}: F_{q^n} \rightarrow E_{a,b}$ leképezést úgy, hogy tetszőleges F_{q^n} -beli u ponthoz egy görbén lévő (x, y) pontpárt rendel az alábbiak szerint:

$$x = (v^2 - b - \frac{u^6}{27})^{\frac{1}{3}} + \frac{u^2}{3}$$
$$y = ux + v$$

ahol

$$v = \frac{3a - u^4}{6u}.$$

Abban az esetben, ha $u = 0$, akkor $f_{a,b}(0) = 0$, azaz a végtelen távoli pont.

A fenti algoritmusból látszik, hogy az egyetlen nehézséget a köbgyökvonás okozhatja. Mivel a fentiek alapján az $x \rightarrow x^3$ hozzárendelés bijekció, ezért a köbgyökvonást helyettesíteni tudjuk a következő képlet segítségével: $x^{\frac{2p-1}{3}}$. Az alábbi lemma alapján látható, hogy az új parametrizáció tényleg egy görbepontot ad vissza.

Lemma: Legyen adott F_{q^n} , ahol $q^n = 2 \bmod 3$, $q > 3$. Minden $u \in F_{q^n}$ esetén $f_{a,b}(u)$ egy pont lesz az $E_{a,b}(F_{q^n})$: $Y^2 = X^3 + aX + b$ görbén.

Bizonyítás: Tekintsük az $u \neq 0$, $(x, y) = f_{a,b}(u)$ esetet. Az x definíciójából:

$$\left(x - \frac{u^2}{3}\right)^3 = v^2 - b - \frac{u^6}{27}$$

ezt az egyenletet, ha megfelelően átrendezzük megkapjuk, hogy

$x^3 + ax + b = u^2x^2 + 2uvx + v^2 = (ux + v)^2$, ami $y = ux + v$ választással megfelel az $x^3 + ax + b = y^2$ egyenletnek.

Nekünk az algoritmusban (a következő alfejezetben bemutatandó eredmény miatt) a hash függvény kiválasztásánál az SHA256 algoritmusra esett a választásunk. Így a felhasznált konstrukció, ami az előzőleg bemutatott f függvény segítségével elliptikus görbére hashel egyszerűen annyiban módosítja a parametrizációt, hogy az u pont helyére annak hashelt képét illeszti, tehát tetszőleges $h : \{0, 1\}^* \rightarrow F_q$ hash függvény esetén a fent ismertetett f leképezés helyett a szerző a

$$H(m) = f(h(m))$$

függvénnyel dolgozik.

A fentiek alapján mi az alábbi algoritmust implementáltam ($a = 1$, $b = 0$, $n = 1$ választással):

hash to curve függvény
<i>Input:</i> $u \in F_q$
<i>Output:</i> (x, y)
1. $v = \frac{3a - h(u)^4}{6h(u)}$
2. $x = (v^2 - b - \frac{h(u)^6}{27})^{\frac{1}{3}} + \frac{h(u)^2}{3}$
3. $y = ux + v$
4. return (x, y)

6.1.2 Tulajdonságok

Password-Hashing Status ([8]) című cikk szerint elemeztük az algoritmust, hogy a tulajdonságait össze tudjuk vetni a már biztonságos jelszó hashelő algoritmusokéval. Hiszen számos biztonsági kérdés van, amelyeket figyelembe kell venni a tervezési fázisban.

Egy jelszó hash-elő algoritmusnak kriptográfiai szempontból biztonságosnak kell lennie és a következőket kell teljesítenie:

- Öskép ellenálló
- Második öskép ellenálló
- Ütközésmentes

Először is vizsgáltuk, hogy ezek a tulajdonságok mennyire jellemzőek a kész algoritmus esetében.

6.1.3 Egyirányúság

Kriptográfiai szempontokból igen fontos, hogy konstrukcióban ismertetett leképezés egyirányú-e. A korábban hivatkozott ([10]) cikkben az alábbi lemma bizonyítja, hogy megfelelő hash függvény választásával a válasz egyértelmű igen.

Lemma: Ha $h(t, \epsilon)$ egy egyirányú hash függvény, akkor $H(t, \epsilon')$ is egyirányú függvény, ahol $\epsilon' = L^2 \epsilon$, $L^2 = 4^2$. Ha tehát L polinomiális a biztonsági paraméterrel és h egyirányú, akkor H egyirányú.

A lemmában szereplő L az f függvény lehetséges előképeinek a maximális darabszáma (tetszőleges elem behelyettesítése esetén), azaz azt mondja meg, hogy hány különböző F_q -beli elem van, ami ugyanazt a görbe pontot adja vissza az f függvény által. A cikkben belátták, hogy L maximum 4 lehet.

Ismét hangsúlyozzuk, hogy h egy általunk választott hash függvény, ami az előzőleg ismertetett f függvény bemenete lesz. A fenti lemma alapján így a $H = f(h)$ függvény egy egyirányú hash függvény feltéve, hogy h is az. Ez indokolja a dolgozatban az SHA256 választását.

6.1.4 Ütközés ellenállóság

Ha a h hash függvény ütközésellenállóságából nem következik H ütközésellenállósága, melyre a [10] cikk szerzői példát adtak. A tervezés során azt találtuk, hogy a one-way password file-oknak elegendő, ha öskép ellenállóak, amit az algoritmusunknál be is tudunk látni.

7 Összehasonlítás

Az elkészült konstrukciót összehasonlítottuk több ismert jelszó hashelő algoritmussal.

Megnéztük, hogy a használatban lévő algoritmusok, amik szintén többkörös hashelő algoritmusok ugyanannyi kör esetében milyen időt produkálnak.

Ebben a fejezetben megvizsgáljuk a Bcrypt, Argon2 és a saját konstrukció közti időbeli különbséget ugyanannyi kör esetében, valamint hasonlítottuk egy másik aszimmetrikus algoritmushoz is az RSA-hoz.

Habár az RSA algoritmus egy aszimmetrikus titkosító algoritmus, elképzelhető, hogy jelszó hashelésre is használható többkörösítve. Éppen ezért előbb nézzük meg az általános RSA algoritmust és hogy mi hogyan használtuk fel jelszó hasheléshez.

7.1 RSA

Az RSA algoritmus egy aszimmetrikus titkosító eljárás, amely az egyik leggyakrabban használt napjainkban. Nevét fejlesztői kezdőbetűiből kapta, 1976-ban fejlesztette ki Ron Rivest, Adi Shamir és Leonard Adleman.

Az RSA algoritmus egy nyílt és egy titkos kulcsot használ. Ahol a nyílt kulcs publikus, segítségével titkosítjuk az üzenetet, míg a titkos kulcsot az üzenetek visszafejtésére használjuk.

7.1.1 Algoritmus

I. Kulcsok generálása

- Vegyünk két különböző véletlen, nagy prímszámot, p -t és q -t
- Legyen $n = p * q$
- Kiszámoljuk az Euler-féle φ függvény értékét n -re: $\varphi(n) = (p - 1) * (q - 1)$
- Vegyünk egy olyan páratlan e egész számot, amely relatív prím $\varphi(n)$ -hez és $1 < e < \varphi(n)$
- Keressünk egy olyan d számot, amelyre teljesül, hogy $e * d \equiv 1 \pmod{\varphi(n)}$
- $PK = (e, n)$
- $SK = d$

II. Titkosítás

Az M üzenet titkosítása a nyilvános kulcs segítségével a következő képlet szerint történik:

$$C = Enc_{PK}(M) = M^e \pmod{n}$$

III. Visszafejtés

A C titkos üzenet visszafejtése a d titkos kulcs segítségével a következő képlet szerint történik:

$$M = Dec_{SK}(C) = C^d \pmod{n}$$

Láthatjuk, hogy az RSA algoritmus erőssége a moduláris számelméletben és a prímszámelmélet egyes tételeiben rejlik. Az eljárás során használnunk kell az Euklideszi algoritmust, illetve a Kibővített Euklideszi algoritmust.

A Moduláris gyorshatványozást (FME) az egészek hatványának meghatározására alkalmazzuk. Az FME segítségével viszonylag kevés művelet elvégzésével kiszámíthatjuk $a^b \pmod{m}$ értéket.

Miller-Rabin prímteszt (MR) segítségével gyorsan kideríthetjük, hogy egy adott szám prímszám-e, ha átmegy a teszten nagy valószínűséggel prímszám. Ez a gyakorlatban éppen elegendő.

Visszafejtéshez használatba kerül a Kínai maradéktétel algoritmus (CRT), amivel növelni lehet a visszafejtés hatékonyságát.

7.1.2 RSA példa

I. Kulcs generálás

- $p = 11, q = 23$
- $n = 11 * 23 = 253$
- $\varphi(n) = 10 * 22 = 220$
- $1 < e < 220, (e, 220) = 1$ válasszuk $e = 3$
- $3 * d \equiv 1 \pmod{220}$

EEA (3, 220)					
k	0	1	2	3(=N)	4
r_k	3	220	3	1	0
q_k	-	0	73	3	
x_k	1	0	1	73	

$$x = (-1)^N * x_N = (-1)^3 * 73 = -73$$

$$d = -73 + \varphi(n) = -73 + 220 = 147$$

- $PK = (253, 3)$

- $SK = 147$
- II. Titkosítás
- $$M = 165$$
- $$C = M^e \pmod{n} = 165^3 \pmod{253} = 110$$
- III. Visszafejtés

$$M = C^d \pmod{n} = 110^{147} \pmod{253} = 165$$

7.1.3 RSA algoritmus többkörösítve hasheléshez

Az algoritmusban használt p és q prímek mérete 2048 bit. A jelszó, amit titkosítunk az „1234” lesz.

A többkörösítést úgy oldottuk meg, hogy a jelszót többször titkosítottuk ugyanazon e értékkel.

```
int[] iterations = {16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144};
for (int iteration : iterations) {
    m = new BigInteger( val: "1234");
    for (int j = 0; j < iteration; j++) {
        m = titkosit(m, e, n);
    }
}
```

8. ábra RSA algoritmus többkörösítve

Körök számát a Bcrypt és az általunk konstruált hashelő algoritmusához igazítottuk. Végezetül az iterációkat lemértük és a kapott eredményt dolgoztuk fel.

7.2 Az összehasonlítás menete

Az összehasonlítás során az algoritmusokat futtatni kezdtem különböző körökkel, illetve feljegyeztem a futási időket. Az algoritmusok bemenetként az „1234” jelszót kapták meg, amit hasheltek. Először összehasonlítottuk a Bcrypt, RSA és a saját algoritmusunk futási idejét, amit most szeretnék bemutatni.

Azt láthatjuk az alábbi táblázatban, hogy ugyanannyi körrel futtatva az algoritmusok hány másodpercig futnak. Az elkészült táblázatban szekundumban adtam meg az értékeket. Első ránézésre észrevehető, hogy az általunk készített konstrukció sokkal lassabb, mint azok,

amelyekhez hasonlítottuk őket. Egy jelszó hashelő algoritmus során fontos, hogy minél lassabb legyen, ekkor a támadók a szótártáblákat lassabban tudják feldolgozni.

Bcrypt esetében a körök számát kettő hatványaként adhatjuk meg, ahol a legkisebb költség 4, azaz a körök száma $2^4 = 16$, így a Bcrypt algoritmus miatt választottuk ezeket a kör számokat.

Körök száma	Bcrypt	Saját	RSA
16	0,0030458	0,5346845	0,009764
32	0,0037977	0,8726219	0,0090346
64	0,0069453	1,7379774	0,0251674
128	0,0130193	1,5386831	0,0334561
256	0,023243	3,5953085	0,0638214
512	0,0431535	5,3515215	0,1371731
1024	0,087049	10,3966082	0,2013071
2048	0,167253	20,9222832	0,452279
4096	0,3439718	46,5067361	0,7515071
8192	0,6667411	86,7408044	1,3365767
16384	1,3645529	173,550175	2,0155614
32768	2,6975281	341,9936824	3,9575559
65536	5,5396111	666,9726737	7,8926018
131072	10,9934645	1326,925216	15,8081499

7. táblázat Algoritmusok összehasonlítása

Láthatjuk, hogy abban az esetben, ha 1 másodperc alatt szeretnénk tartani a hash számítást, akkor a Bcrypt esetében 8192 kört szükséges futtatni. Összehasonlítva egy aszimmetrikus rendszer hatékonyságával azt láthatjuk, hogy az RSA számításakor 4096 kör szükséges, míg az elkészült algoritmusnál 32 kör elegendő ehhez.

Az Argon2 algoritmus esetében a körök számán kívül egyéb paramétereket is lehet állítani, ezért kipróbáltam néhány jelszó hasheléshez szükséges ajánlást és azoknak az idejét jegyeztem fel.

Típus:	Argon2id	Argon2i	Argon2id	Argon2id	Argon2d
Memória mérete (KiB):	1048576	65536	4096	32 768	4096
Iterációk száma:	2	2	3	1	128
Párhuzamosság:	8	4	1	1	1
Só hossza:	16	24	16	16	16
Hash hossza:	32	24	32	32	32
Idő (szekundum):	2,0267844	0,2813827	0,1232007	0,2651719	0,6441149

8. táblázat Argon2 összehasonlítás

Az utolsó oszlopban lévő beállítás az Argon2 referencia C implementációjában szerepel, ami elérhető githubon ([2]). A referencia C implementáció szerint az iterációk alapértelmezett száma 3, de növeltem addig, hogy 1 másodperc alatt tartsuk a hash számítást a többi paraméter megtartásával.

Láthatjuk, hogy az algoritmusokat különféle paraméterezésekkel tudjuk lassítani, amennyire csak szeretnénk. A mi algoritmusunk megfelelő alternatíva, hiszen megfelelően lehet szabályozni a futási időt.

Egy másik szempont, ami alapján hasonlítottuk az elkészült algoritmust a gyakorlatban használt módszerekhez, az a memória használat. Mivel az általunk készített kód Python nyelven íródott, ezért a memory-profiler modult használtuk az elemzéshez.

A memory-profiler egy olyan Python modul, ami egy folyamat memóriafogyasztásának megfigyelésére alkalmas, valamint a Python programok memóriafogyasztásának soronkénti elemzését végzi el. Fontos megjegyezni, hogy maga a memory-profiler jelentős mennyiségű memóriát fogyaszt.

Összehasonlítottuk a saját kódunkat, a Pythonban implementált Bcrypt és Argon2 algoritmusok futtatásával, miközben 1 másodperc alatt hashelik a megadott jelszót.

Elsőként a Bcrypt algoritmus memória fogyasztását néztük meg, aminek az „1234” jelszót adtam meg hasheléshez, illetve a körök számát 2^{13} -ra állítottam be.

```

Line #   Mem usage   Increment   Occurrences   Line Contents
=====
18  19.5078125000 MiB  19.5078125000 MiB      1  @profile(precision=10, stream = fp)
19                                     def bcryptTest():
20  19.5703125000 MiB   0.0625000000 MiB      1      salt = bcrypt.gensalt(rounds=13)
21  19.5781250000 MiB   0.0078125000 MiB      1      bcrypt.hashpw(b'1234', salt)

```

9. ábra Bcrypt memória igénye

Az Argon2 algoritmust a már bemutatott C implementáció szerint paramétereztem fel, kivéve a körök számát, amit 128-nak állítottam be.

Line #	Mem usage	Increment	Occurences	Line Contents
=====				
22	19.5195312500 MiB	19.5195312500 MiB	1	@profile(precision=10, stream = fp)
23				def argonTest():
24	19.5273437500 MiB	0.0078125000 MiB	2	argon2Hasher = argon2.PasswordHasher(
25	19.5195312500 MiB	0.0000000000 MiB	1	type=argon2.Type.D, time_cost=128,
26	19.5195312500 MiB	0.0000000000 MiB	1	memory_cost=2 ** 12,parallelism=1,
27	19.5195312500 MiB	0.0000000000 MiB	1	hash_len=32, salt_len=16)
28	19.6093750000 MiB	0.0820312500 MiB	1	argon2Hasher.hash("1234")

10. ábra Argon2 memória igénye

Az általunk implementált megoldásnál a körök számát 32-re állítottam, és így futtattam le a memória elemzést.

Line #	Mem usage	Increment	Occurences	Line Contents
=====				
768	21.3398437500 MiB	21.3398437500 MiB	1	@profile(precision=10, stream = fp)
769				def pwHashTest():
770	21.3398437500 MiB	0.0000000000 MiB	1	toHash = '1234'
771	21.3671875000 MiB	0.0078125000 MiB	33	for i in range(0, 32):
772	21.3671875000 MiB	0.0195312500 MiB	32	generatedPoint = ec.hashToCurve(toHash)
773	21.3671875000 MiB	0.0000000000 MiB	64	toHash = TatePairing\
774	21.3671875000 MiB	0.0000000000 MiB	32	.compute(TatePairing, P, generatedPoint, ec)\
775				.toString()

11. ábra Saját algoritmus memória igénye

Az eredményeknél az első oszlop a vizsgált kód sorszáma, a következő oszlop (Mem usage) a Python-interpreter memóriahasználata az adott sor végrehajtása után. A harmadik oszlop (Increment) az aktuális sor memóriájának különbségét mutatja az előző sorhoz képest. Az utolsó oszlop (Line Contents) pedig a profilozott kódot mutatja meg, ami az adott sorban található.

Végül megvizsgáltuk még a kódunk méretét, és az ismereteink alapján összevettettük a Bcrypt és Argon2 kód méretével. A rendelkezésünkre álló adatok alapján a Bcrypt kb. 20-30 kB és az Argon2 80-110 kB. A mi kódunk mérete körülbelül a Bcrypt és az Argon2 kód mérete között helyezhető el.

8 Összefoglalás

A dolgozat középpontjában a biztonságos jelszótárolás szerepel. Mivel manapság nagyon fontos a jelszavak megfelelő tárolása és egyre több támadó próbálja feltörni a jelszavakat így egy igazán fontos területről van szó.

Kidolgoztunk és implementáltam egy megfelelő alternatívát a ma elterjedt jelszó hashelő algoritmusok mellé. A dolgozat során egy bilineáris párosításon alapuló algoritmust mutattam be, amely jelszavak hashelésére szolgál.

Implementáltam egy algoritmust, amely egy többkörös jelszó hashelő algoritmus. A jelszó hashelő algoritmus fontos tulajdonsága, hogy minél lassabb legyen, a mi algoritmusunkat a körök számának állításával tudjuk az igényeknek megfelelően lassítani.

Az általunk javasolt algoritmus két fő részből áll. Az első fő része, egy elliptikus görbecsoportba történő hashelő algoritmus. A másik fő része az algoritmusnak a bilineáris leképezés használata.

Az ajánlott algoritmus tulajdonságait vizsgáltam, hogy a Password Hashing Competition formai tulajdonságainak eleget tesznek-e. Megmutatom, hogy az általunk készített konstrukció egyirányú. Sózást használunk az algoritmusban, azaz az azonos jelszavak is egyediek lesznek a szózásnak köszönhetően.

Megvizsgáltuk az algoritmus memória fogyasztását és a kód méretét is, illetve összevetettük a gyakorlatban használt algoritmusokéval. A vizsgálatok azt mutatják, hogy az algoritmus memória fogyasztása kevesebb, mint az Argon2 és Bcrypt algoritmusoké. A kód mérete pedig a két összevetett algoritmus kódmérete között helyezkedik el.

Végezetül egy külön fejezetben elemeztem, hogy az implementált algoritmus mennyivel lassabb, mint a Bcrypt és az Argon2 algoritmus. Továbbá összehasonlítottuk egy aszimmetrikus rendszer hatékonyságával is, az RSA algoritmust módosítottuk megfelelően, hogy többkörösíteni tudjuk.

Összességében elmondhatjuk, hogy az általunk készített algoritmus megfelelő alternatíva lehet a manapság használt rendszerek helyett. Sikerült implementálni egy lassú, többkörös jelszó hash algoritmust, amely rendelkezik a hash függvények egyirányú tulajdonságával. A kód méretét tekintve átlagos méretű, nem terveztük, hogy korlátozott erőforrású eszközön futtassuk, így a mérete megfelelő.

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőimnek, Dr. Pintér-Husztai Andreának és Dr. Bertók Csanádnak, akik elláttak a dolgozatom elkészítése alatt tanácsaikkal, ötleteikkel. Mindezek mellett szeretném megköszönni azt az időt, amit rászántak a dolgozatom elkészítésére, a megbeszélésekre és a kérdéseim megválaszolására. Hálás vagyok a folyamatos támogatásukért.

Továbbá szeretném megköszönni Oláh Norbertnek és Kovács Szabolcsnak, hogy a kriptográfiai ismereteim bővítésében részt vettek, illetve segítettek a dolgozat elkészítéséhez szükséges Python ismeretek megszerzésében.

Saját munka leírása

- Feldolgoztam, értelmeztem majd implementáltam véges testbeli hatványozáshoz a Sliding Window metódust.
- Feldolgoztam, értelmeztem majd implementáltam az Elliptikus görbecsoportba történő hashelő algoritmust.
- Implementáltam az ajánlott többkörös jelszó hashelő algoritmust.
- Feldolgoztam és értelmeztem az Argon2 és Bcrypt algoritmusok működését, hogy összehasonlíthassam a mi algoritmusunkkal.
- Lemértem az Argon2, Bcrypt, és a saját algoritmusunk futási idejét különböző paraméter beállítások mellett.
- Átalakítottam az RSA algoritmust többkörössé és mértem a futási idejét.
- Lemértem és összehasonlítottam az Argon2, Bcrypt és a saját algoritmusunk memória fogyasztását.

Lezárás dátuma: 2022.06

9 Irodalomjegyzék

- [1] Alex Biryukov, Daniel Dinu és Dmitry Khovratovich, „Argon2: the memory-hard function for password hashing and other,” October 1, 2015.
- [2] Argon2, 2015. [Online]. Available: <https://github.com/P-H-C/phc-winner-argon2/>.
- [3] Bagossy Attila, Pethő Attila és Vécsi Ádám, „Cross-platform Identity-based Cryptography using WebAssembly,” *Infocommunications Journal* 11(4), pp. 31-38, 2019.
- [4] C.K.Koç, „Analysis of Sliding Window Techniques,” *Computers and Mathematics with Applications*, %1. kötet30, %1. szám10, p. 17–24, 1995.
- [5] Costello Craig, Pairings for beginners, 2012.
- [6] Elliptic Curve Points, „www.desmos.com,” [Online]. Available: <https://www.desmos.com/calculator/ialhd71we3>. [Hozzáférés dátuma: 10 2021].
- [7] Folláth János, Huszti Andrea és Pethő Attila, *Informatikai biztonság és kriptográfia*, 2010.
- [8] Hatzivasilis George, „Password-Hashing Status,” *Cryptography*, pp. 1(2), 10, 27 June 2017.
- [9] Huszti Andrea, Kovács Szabolcs és Oláh Norbert, „Hybrid anonymous message broadcast for VANETs,” *In 2021 17th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*(52687), pp. 03-108, 2021 október.
- [10] Icart Thomas, How to Hash into Elliptic Curves, 2009.
- [11] Niels Provos és David Mazières, „A Future-Adaptable Password Scheme,” in *Proceedings of 1999 USENIX Annual Technical Conference: 81-91*, 1999.
- [12] Nyakacska Lilla, *Elliptikus görbék és bilineáris párosítások (Diplomamunka, Debreceni Egyetem, Debrecen, Hungary)*, Letöltve: <https://dea.lib.unideb.hu/dea/handle/2437/232828>, 2016.
- [13] Password Hashing Competition, „www.password-hashing.net,” [Online]. Available: <https://www.password-hashing.net/>.
- [14] SageMath, „www.sagemath.org,” [Online]. Available: <https://www.sagemath.org/>. [Hozzáférés dátuma: 10 2021].

- [15] [www.certicom.com](https://www.certicom.com/content/certicom/en/21-elliptic-curve-addition-a-geometric-approach.html). [Online]. Available: <https://www.certicom.com/content/certicom/en/21-elliptic-curve-addition-a-geometric-approach.html>. [Hozzáférés dátuma: 10 2021].
- [16] [www.cybernews.com](https://cybernews.com/news/largest-compilation-of-emails-and-passwords-leaked-free/), 12 02 2021. [Online]. Available: <https://cybernews.com/news/largest-compilation-of-emails-and-passwords-leaked-free/>. [Hozzáférés dátuma: 10 2021].
- [17] [www.nki.gov.hu](https://nki.gov.hu/it-biztonsag/hirek/serulekenysegeket-fedeztek-fel-tobb-nepszeru-jelszokezelo-program-kapcsan/). [Online]. Available: <https://nki.gov.hu/it-biztonsag/hirek/serulekenysegeket-fedeztek-fel-tobb-nepszeru-jelszokezelo-program-kapcsan/>. [Hozzáférés dátuma: 10 2021].
- [18] [www.nki.gov.hu](https://nki.gov.hu/it-biztonsag/hirek/ellatasi-lanc-elleni-tamadas-tortent-egy-prominens-jelszokezelo-alkalmazasnal/). [Online]. Available: <https://nki.gov.hu/it-biztonsag/hirek/ellatasi-lanc-elleni-tamadas-tortent-egy-prominens-jelszokezelo-alkalmazasnal/>. [Hozzáférés dátuma: 10 2021].
- [19] [www.pcworld.hu](https://pcworld.hu/pcwlite/13-milliard-felhasznaloi-adat-kering-a-neten-magyarok-is-vannak-koztuk-287196.html), 21 11 2020. [Online]. Available: <https://pcworld.hu/pcwlite/13-milliard-felhasznaloi-adat-kering-a-neten-magyarok-is-vannak-koztuk-287196.html>. [Hozzáférés dátuma: 10 2021].