Elliptikus görbéken alapuló nyilvános kulcsú kriptográfia elemzése chipkártyás és PC-s környezetben*

Berta István Zsolt - Mann Zoltán Ádám

2002. augusztus 15.

Tartalomjegyzék

*Ez a dolgozat a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karán a 2000. évi Tudományos Diákköri Konferencián 1. helyezést ért el; a 2001. évi Országos Tudományos Diákköri Konferencián különdíjban részesült.

1. Bevezetés

A matematikusok már évszázadok óta foglalkoztak az elliptikus görbékkel, pusztán az elmélet esztétikai szépsége miatt. Az 1980-as években fordult először a kriptográfusok figyelme is az elliptikus görbék felé, mivel ezek segítségével rengeteg véges csoportot lehet gyártani, melyekben a diszkrét logaritmus probléma segítségével erős kriptográfia hozható létre [Miller1985]. Az azóta eltelt mintegy 15 év tapasztalatai alátámasztani látszanak ezt az elgondolást, de mivel nincs semmilyen elméleti bizonyíték a módszer biztonságára, így sokan úgy tartják, hogy ez a technológia még mindig nem érett meg az ipari és egyéb alkalmazásra. Ezért minél több gyakorlati tapasztalatra van szükség arról, hogy az ECC (Elliptic Curve Cryptography) valójában mennyire biztonságos. Vizsgálódásainkkal mi is hozzá szeretnénk járulni e gyakorlati tapasztalatok gyűjtéséhez.

Sietni kell azonban a tapasztalatszerzéssel. Alapvetően két fajta veszéllyel kell szembenéznünk az aszimmetrikus, nyilvános kulcsú kriptográfia terén. Az első veszély abból fakad, hogy lényegében egyetlen nyilvános kulcsú titkosítási eljárás terjedt el széles körben, az RSA. Ennek biztonságára vonatkozóan sincsen elméleti garanciánk, mégis lényegében a teljes PKI (Public Key Infrastructure, Nyilvános kulcsú infrastruktúra) erre épül. Ha egy napon valaki találna egy hatékony algoritmust az RSA törésére, ennek beláthatatlan következményei lennének. Nagyon jó lenne tehát, ha lennének más nyilvános kulcsú kriptográfiai módszereink is. Erre pedig jelenleg az ECC a legesélyesebb.

A másik veszély a törésre felhasználható számítási kapacitás ugrásszerű növekedése. Ez egyrészt az olcsó és nagy teljesítményű számítógépek széleskörű elterjedésének köszönhető, másrészt annak, hogy ezek a számítógépek egyre nagyobb mértékben hálózatba vannak kötve. Ennek megfelelően az utóbbi évek leglátványosabb törései nem szuperszámítógépek, hanem párhuzamosan működő munkaállomások segítségével történtek. [Crack1999] Ez a jövőben várhatóan egyre inkább így lesz, mivel a nagy teljesítményű munkaállomások az idő jelentős részében nincsenek használatban, így nagy számítási kapacitás marad kihasználatlanul. Ennek megfelelően egyre fontosabbá válik ezek kihasználása, akár párhuzamos algoritmusok alkalmazásával, akár virtuális szuperszámítógéppé összefogva az egyes munkaállomásokat (Metacomputing).

Szintén viszonylag új szempont a kriptográfiában, hogy olyan algoritmusokra van szükség, amelyek a chipkártyák adta szűk erőforrás-keretek között is működőképesek. Érzékeltetésül: egy tipikus chipkártya órajelfrekvenciája kb. 3-5 MHz, írható-olvasható memóriája 4-32 KByte. Ez erősen korlátozza a nyilvános kulcsú kriptográfia alkalmazhatóságát ezekben az eszközökben.

Az ECC annyiban jelenthet megoldást ezekre a problémákra, hogy jelenlegi ismereteink szerint ugyanazt a biztonsági szintet lényegesen kisebb kulcsokkal tudja elérni, mint az RSA. Ez abból

fakad, hogy az ECC alapját képező ECDLP (Elliptic Curve Discrete Logarithm Problem) törésére az általános esetben nem ismeretes exponenciálisnál hatékonyabb algoritmus, míg az RSA törésére igen (bár persze ez sem polinomiális). Ebből nem csak az következik, hogy ugyanazon biztonsági szint eléréséhez az ECC-nek kisebb kulcsokra van szüksége, hanem az is, hogy ahogy a számítási kapacitás növekedésével a kulcsméreteket növelni kell, ez a különbség tovább fog nőni. A kisebb kulcsméret lehetővé teszi a chipkártyás implementációt is. Mint látni fogjuk azonban, nem gyértelmű, hogy a kisebb kulcsok következtében a műveletigény is csökken.

A dolgozatban először áttekintjük az elliptikus görbék matematikai alapjait és kriptográfiai alkalmazhatóságuk elméletét. Ezután megismerkedünk azzal a chipkártyás környzettel, amelyikben az ECC-t implementáltuk. Ezt követően bemutatjuk az implementáció során alkalmazott algoritmusokat, majd kitérünk a (PC-s és chipkártyás) megvalósítás részleteire. Végül eredményeinket, tapasztalatainkat gondos elemzésnek vetjük alá, ennek kapcsán azonosítjuk az ECC hatékonyság és biztonság szempontjából kritikus részeit, és a törési lehetőségeket is számba vesszük.

2. Elméleti alapok

A Fermat-sejtés bizonyításának kapcsán nemrég az érdeklődés középpontjába kerültek az elliptikus görbék, ugyanis a több évszázados sejtés bizonyításához meglepő módon ezek jelentették a kulcsot. [Ronyai1995] Azonban az elliptikus görbék tanulmányozása már igen nagy múltra tekint vissza. Mint a matematika oly sok egyéb fejezeténél, itt is elmondható, hogy már Euler is foglalkozott a témával, és olyan mély eredményeket ért el, melyek igazi jelentőségére csak jóval később derült fény, amikor az algebrai topológia és a komplex függvénytan eljutott a megfelelő fejlettségi szintre. Az alábbiakban a szerteágazó elméletből csak azokat az eredményeket ismertetjük, amelyek a további kriptográfiai vizsgálódásokhoz szükségesek, a rövidség érdekében ezeket is bizonyítás nélkül. A bizonyításokhoz és az elmélet mélyebb összefüggéseihez bőséges szakirodalom áll rendelkezésre, pl. [Husemoller1987]

2.1. Elliptikus görbék elmélete

Elliptikus görbét az általános esetben az alábbi egyenlet definiál:

$$y^2 + axy + by = x^3 + cx^2 + dx + e \tag{1}$$

Itt az a_i együtthatók egy adott K testből valók, csakúgy, mint az x és y változók. Ilyenkor az (1) egyenlet megoldásai az E elliptikus görbe K test fölötti pontjait szolgáltatják. Ezek halmazát $E(K)$ jelöli. (Ha L a K test bővítése, akkor tekinthetjük ugyanezen egyenlet L fölötti megoldásait is, így az $E(L)$ elliptikus görbéhez jutunk.) A legszemléletesebb a $K = R$ (valós) eset, így először ezzel foglalkozunk. Ezután megvizsgáljuk a $K = C$ (komplex) esetet is, mivel a valós esetben szereplő, misztikusnak tűnő definíciók itt világosabbá válnak. A $K = Q$ (racionális) esettel csak érintőlegesen foglalkozunk, mivel ez kriptográfiai szempontból nem jelentős. Végül rátérünk a számunkra legfontosabbra, a véges testek esetére.

2.1.1. Valós elliptikus görbék

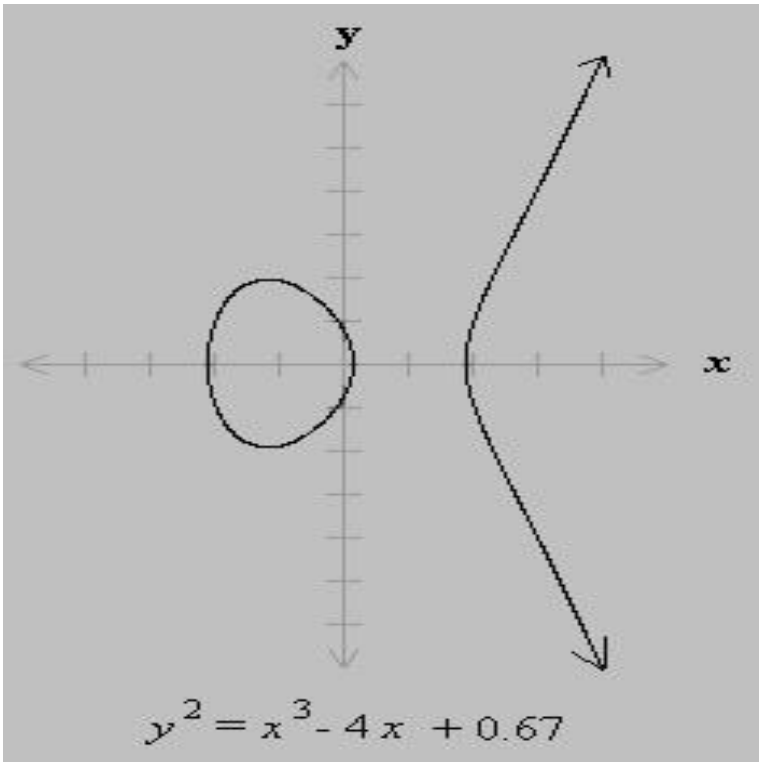
A valós esetben az (1) egyenlet bal oldalán szereplő xy-os és y-os tag lineáris helyettesítéssel eliminálható. Ezt ki lehet számolni, de abból is látszik, hogy az xy-os tag a görbe forgatásának, az y-os tag pedig y irányú eltolásnak felel meg az $x - y$ síkon. Vagyis egy forgatással és egy eltolással ezektől meg lehet szabadulni. Ezután a görbe egyenlete a következő alakot ölti:

$$y^2 = f(x) \tag{2}$$

ahol $f(x)$ egy harmadfokú polinom. Sőt, azt is fel lehet tételezni, hogy

$$f(x) = x^3 + ax + b$$

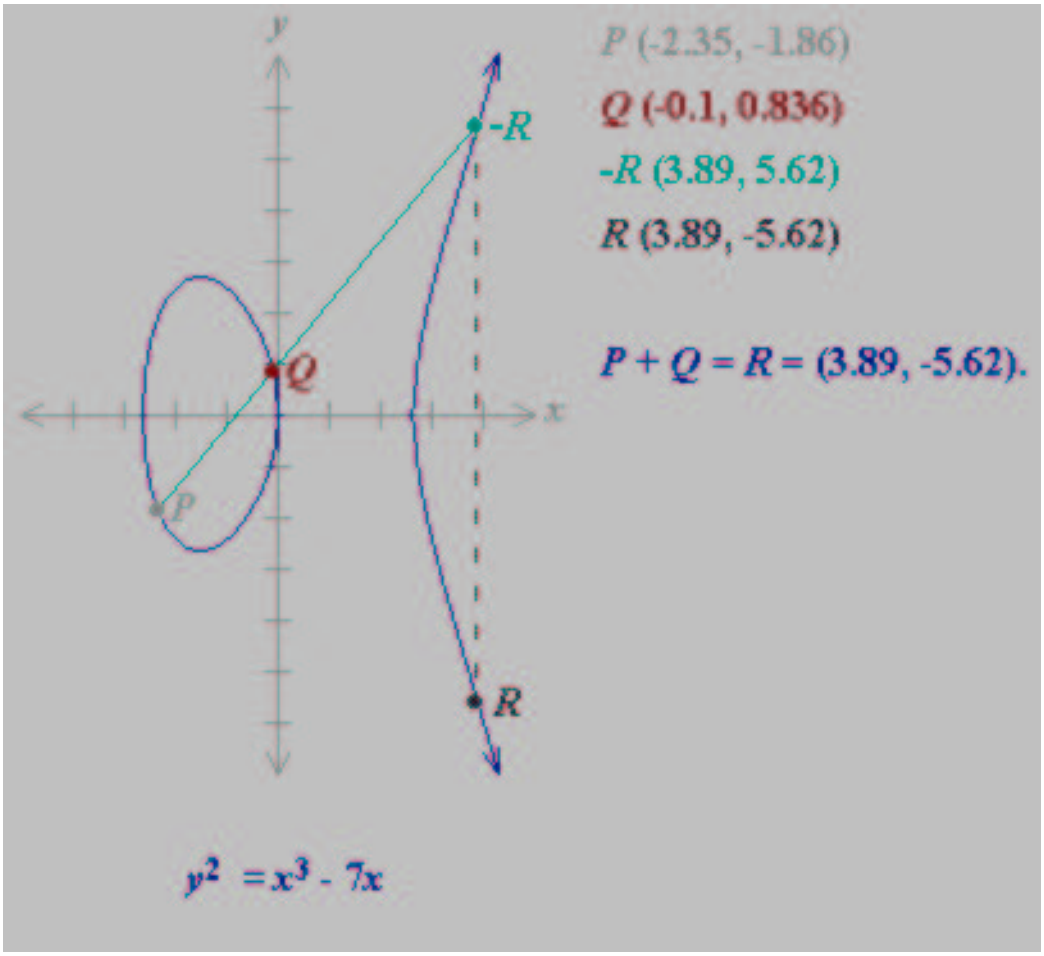
alakú. Azt is meg szokták követelni, hogy ennek a polinomnak ne legyen többszörös gyöke. (Ez tulajdonképpen annak felel meg, hogy ha az egyenletet $F(x, y) = 0$ alakra hozzuk, az F függvény legyen sima, vagyis akárhányszor differenciálható.) Egy ilyen egyenlettel definiált görbe grafikonja látható a következő ábrán (e fejezet ábrái a Certicom [Certicom] honlapjáról származnak).



(Ez a görbe két különálló komponensből áll, de ez nincs mindig így.)

Az elliptikus görbék kriptgráfiai alkalmazhatósága a rajtuk értelmezhető csoport-struktúrából fakad. Ehhez először is definiálni kell egy kétváltozós műveletet a görbe pontjai között. Ezt összeadásnak fogjuk nevezni, összhangban azzal, hogy kommutatív csoportok esetén általában az additív írásmód a szokásos. Az összeadást a következőképpen szeretnénk definiálni. Legyen adva a görbén két pont: P és Q. Tekintsük a rajtuk átmenő egyenest. Ennek az egyenesnek általában pontosan három közös pontja van a görbével; a harmadik közös pontot jelölje $-R$. Tükrözzük $-R$-

et az y tengelyre. Az így kapott pont szintén rajta van a görbén, jelölje ezt R. Ekkor definiáljuk az összeget így: $P + Q = R$. Erre mutat egy példát a következő ábra:



Számos probléma van azonban ezzel a definícióval. Például mi van akkor, ha a P és Q pontok egymásnak az x tengelyre vonatkozó tükörképei? Ebben az esetben ugyanis a rajtuk átmenő egyenes nem metszi többször a görbét. E probléma kiküszöbölésére ki kell egészítenünk a görbét egy új ponttal, melyet O-val jelölünk, és úgy gondolhatunk rá, mint a sík y irányú végtelen távoli pontjára. Mint később kiderül, ez a pont lesz a csoport neutrális eleme.

Az O pont precíz definíciójához projektív koordinátákra kell áttérni. A projektív sík pontjai az (x, y, z) valós hármasok ekvivalenciaosztályai, ahol az (x_1, y_1, z_1) és (x_2, y_2, z_2) pontokat akkor tekintjük ekvivalensnek, ha van olyan, 0-tól különböző λ valós szám, amire $x_1 = \lambda x_2$, $y_1 = \lambda y_2$, $z_1 = \lambda z_2$. Ha $z \neq 0$, akkor az (x, y, z) osztályt reprezentáló $(\frac{x}{z}, \frac{y}{z}, 1)$ projektív pont megfeleltethető az $(\frac{x}{z}, \frac{y}{z})$ „normál" pontnak, így a „normál" (affin) sík beágyazható a projektív síkba. Ezen kívül azonban a projektív síkon vannak $z = 0$ tulajdonságú pontok is, amikre úgy gondolhatunk, mint az affin sík végtelen távoli pontjaira. (Ezek alkotják együtt a végtelen távoli egyenest.) Ha most arra vagyunk kíváncsiak, hogy az elliptikus görbénk hogy néz ki a projektív síkon, akkor a görbe egyenletében x helyére $\frac{x}{z}$-t, y helyére pedig $\frac{y}{z}$-t kell írnunk, majd a nevezőkben szereplő z-ket eliminálnunk. A következőt kapjuk:

$$y^2z = x^3 + axz^2 + bz^3$$

Nyilvánvaló, hogy ha $z \neq 0$, akkor feltehető, hogy $z = 1$, és ilyenkor visszakapjuk az eredeti egyenletet. Ha viszont $z = 0$, akkor az adódik, hogy $x = 0$, viszont y tetszőleges lehet. Ennek a projektív síkon a $(0, 1, 0)$ által reprezentált ekvivalenciaosztály felel meg. Ez pedig éppen az affin sík y irányú végtelen távoli pontja. Éppen ezt jelöltük O-val. Látható tehát, hogy az O pontnak a görbéhez való hozzávétele mégsem olyan erőltetett, hiszen a projektív síkon valóban hozzátartozik még egy ilyen tulajdonságú pont a görbéhez.

E kitérő után térjünk vissza az összeadás definíciójához. Most már tudunk válaszolni arra a kérdésre, hogy mi legyen két olyan pont összege, melyek egymásnak az x tengelyre vett tükörképei. Ugyanis az O pont lesz a két ponton átmenő egyenesnek a görbével vett harmadik közös pontja. Ha még azt feltesszük, hogy az O pontnak az x tengelyre vett tükörképe önmaga, akkor azt kapjuk, hogy a fenti szabály alkalmazásával a két pont összege éppen az O pont. Ezek szerint - tudván, hogy az O lesz a csoport nulleleme - két ilyen pont egymás inverze. Ez indokolja azt is, hogy az összeadási szabály fenti leírásánál miért használtuk az R és $-R$ pontokra ezt a jelölést: e két pont ugyanis tényleg egymás inverze.

Külön meg kell még vizsgálni azt az esetet, amikor a két összeadandó pont megegyezik. Ilyenkor összekötő egyenesükön értsük a görbének az adott pontban vett érintőjét. (Ilyen nyilván van, hiszen a görbe sima.) Hasonlóképp az is előfordulhat, hogy a P és Q pontokat összekötő egyenes nem metszi el egy harmadik pontban is a görbét, hanem például a P pontban érinti azt. Ilyenkor a $-R$ pont megegyezik a P-vel.

Két pont összegét lehet tisztán algebrai úton, képletekkel is definiálni. Tegyük fel, hogy a két összeadandó pont koordinátái $P(x_1, y_1)$ illetve $Q(x_2, y_2)$. Az összeg koordinátái ekkor:

$$x_3 = s^2 - x_1 - x_2$$
$$y_3 = s(x_1 - x_3) - y1$$

ahol

$$s = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1}, \; ha \, P \neq Q \\ \frac{3x_1^2 + a}{2y_1}, \; ha \, P = Q \end{cases}$$

éppen az összekötő egyenes meredeksége. A képletekből jól látszik, hogy ha a két pont egymás inverze, akkor 0-val kéne osztani; ez felel meg annak, hogy a végtelen távoli pontot kapjuk eredményül.

Több szempontból is nagy jelentőségű az a tény, hogy az összeg koordinátáit egy képlet segítségével ki tudtuk fejezni az összeadandók koordinátáiból. Először is, a képleteket a geometriai szemlélet segítségével lehet ugyan levezetni, de innentől fogva már ezek a képletek helyettesítik a geometriai szemléletet, hiszen ezekben már minden benne van, ami a csoport struktúrára jellemző. Ez fontos, hiszen kriptográfiai rendszerünket így nem geometriai, hanem algebrai eszközökkel tudjuk definiálni. Még sokkal lényegesebbé válik ez a szempont, amikor áttérünk a véges testek feletti elliptikus görbék tanulmányozására, hiszen azoknál már nem tudunk a geometriai szemléletre

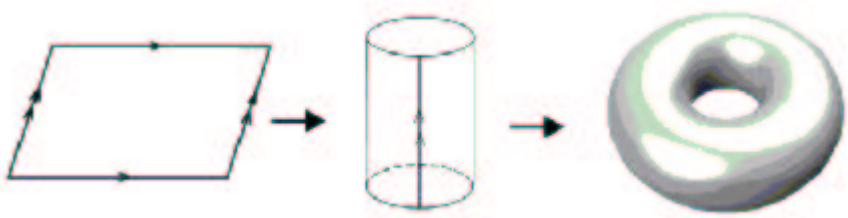
támaszkodni. Továbbá e képletek azt is bizonyítják, hogy a görbe két pontján átmenő egyenes tényleg átmegy a görbének egy harmadik pontján is. (Sőt, a képletek levezetése során az is látszik, hogy pontosan három közös pont lesz.) Végezetül, mivel a képletekben csak a négy alapművelet szerepel, így ebből az is következik, hogy ha P és Q koordinátái valamint a görbe a paramétere benne vannak egy bizonyos K testben, akkor az összeg koordinátái is K-ból valóak.

Hiányzik még annak a bizonyítása, hogy az elliptikus görbe pontjai a fent definiált művelettel kommutatív csoportot alkotnak. A műveletre vonatkozó zártságról már szóltunk, csakúgy, mint a nullelemről és az inverzről. A kommutativitás nyilvánvaló. Már csak az asszociativitás hiányzik, ennek bizonyítása azonban lényegesen nehezebb, így ezt itt nem részletezzük. A következő pontban azonban látni fogjuk, hogy ez min is múlik.

2.1.2. Komplex elliptikus görbék

A valós elliptikus görbék tárgyalásánál semmilyen módon nem tudtuk megindokolni, hogy miért úgy érdemes definiálni a műveletet, ahogy azt definiáltuk. Mint ebben a pontban látni fogjuk, ennek igen mély okai vannak. Mivel egy komplex elliptikus görbét csak a négy-dimenziós tér két-dimenziós felületeként tudnánk ábrázolni, így erről lemondunk, és más módon közelítjük meg a problémát.

Tekintsünk egy rácsot a komplex síkon. Ezen egy $\Lambda = \{av+bw : a,b \in Z\}$ halmazt értünk, ahol v és w adott komplex számok, melyek nem egymás valósszorosai (vagyis mint valós kétdimenziós vektorok, kifeszítik az egész síkot). Ha például $v = 1$ és $w = i$, akkor éppen a Gauss-egészeket kapjuk. Egy komplex számokon értelmezett f függvényt Λ szerint periodikusnak mondunk, ha $z_1 - z_2 \in \Lambda$ esetén mindig $f(z_1) = f(z_2)$ is teljesül. Egy ilyen tulajdonságú függvényt elég a v és w által kifeszített parallelogrammán megadni, mivel onnan egyértelműen kiterjeszthető az egész síkra. Ráadásul e parallelogramma szemközti oldalain lévő megfelelő pontokban a függvény értéke megegyezik. Ennek alapján tekinthetjük úgy is, hogy a függvény C/Λ faktortéren van értelmezve. A faktortérre gondolhatunk úgy, mintha a parallelogramma szemközti oldalpárjait összeragasztottuk volna. Így egy tóruszt kapunk: [Crutchley1999]



Az elmélet nagyon szép, azonban egyáltalán nem nyilvánvaló, hogy vannak ilyen tulajdonságú, nem konstans függvények. Szerencsére vannak: ilyen a rácshoz tartozó Weierstrass-féle $\wp$ függvény és annak $\wp'$ deriváltja. Ezeknek a függvényeknek a rácspontokban pólusai vannak, de egyébként az egész síkon regulárisak. A faktorizálásnál (tehát amikor áttérünk a tóruszra) a rácspontok egyetlen pontba mennek át. Ráadásul e függvények eleget tesznek a következő függvény-egyenletnek:

$$\wp'(z)^2 = 4\wp(z)^3 - g_2\wp(z) - g_3$$

ami egy komplex elliptikus görbe egyenletének tekinthető (g_2 és g_3 a rácstól függő állandók). Ebből tehát az derül ki, hogy egy komplex elliptikus görbe $(\wp(z), \wp'(z))$ szerinti paraméterezése egy megfeleltetést ad a görbe pontjai és a tórusz között. (Ennek során a tórusznak az a pontja, ahol $\wp$-nek pólusa van, a görbe végtelen távoli pontjának felel meg.) Egy igen lényeges eredmény, az uniformizációs tétel, értelmében minden komplex elliptikus görbéhez van egy olyan rács és $\wp$ függvény, melyre ez a megfeleltetés bijektív, sőt, a pontok „közelségét" is megtartja. Ennek alapján végre választ kaphatunk arra a kérdésre is, hogy miért úgy kellett definiálni az összeadást az elliptikus görbe pontjai között, ahogy definiáltuk. A válasz az, hogy a tóruszon, ami a komplex számok additív csoportjának faktora, adva van egy additív csoport-struktúra, és ezt akarjuk átvinni az elliptikus görbére. Pontosabban úgy kell definiálni az összeadást az elliptikus görbén, hogy a fenti leképezés Abel-csoportok közötti izomorfizmus legyen. A $\wp$ függvényre teljesül az alábbi összefüggés:

$$\begin{vmatrix} \wp(z_1) & \wp'(z_1) & 1 \\ \wp(z_2) & \wp'(z_2) & 1 \\ \wp(z_1+z_2) & -\wp'(z_1+z_2) & 1 \end{vmatrix} = 0$$

Ebből következően a $(\wp(z_1), \wp'(z_1))$, $(\wp(z_2), \wp'(z_2))$ és $(\wp(z_1+z_2), -\wp'(z_1+z_2))$ pontok egy egyenesen vannak. (Itt a harmadik pont éppen az első kettő összegének az inverze.) Vagyis úgy kell definiálni az összeadást az elliptikus görbe pontjai között, hogy egy egyenesre eső három pont összege O legyen, és tényleg ezt tettük a korábbiakban. Ez volt az az összefüggés, amit már Euler is ismert.

2.1.3. Racionális elliptikus görbék

A racionális számtest felett is éppúgy értelmezhetünk elliptikus görbéket, mint a valós vagy a komplex számtest felett, és a pontok közötti összeadást is ugyanazon képletek definiálják, mint az előző esetekben. Mivel a racionális elliptikus görbék sem az általános elmélet megértését nem szolgálják, sem pedig kriptográfiai céljainkat, így csak egészen röviden érintjük a témát. Hogy egyáltalán foglalkozunk vele, az azért van, mert számelméleti szempontból viszont ezek bizonyultak a leghasznosabbaknak, így egyetlen elliptikus görbékről szóló leírás sem lehet teljes a racionális eset ismertetése nélkül.

Az egyik legfontosabb, immár klasszikusnak számító eredmény a racionális elliptikus görbék elméletében a Mordell-tétel (1921). Ennek értelmében a racionális számtest felett definiált elliptikus görbe, mint additív Abel-csoport, végesen generált. Vagyis a csoport két kisebb csoport direkt szorzatára bontható, az egyikben vannak a véges rendű elemek, a másik pedig véges sok (r darab) végtelen rendű elem által generált. Ezt az r számot nevezzük a görbe rangjának. (Az egész csoport pontosan akkor véges, ha $r = 0$.)

A TSW (Taniyama-Shimura-Weil) sejtés, melyből a Fermat-sejtés is következik, szintén racionális elliptikus görbékkel kapcsolatos, és lényegében azt állítja, hogy minden racionális együtthatós elliptikus görbe paraméterezhető úgynevezett moduláris függvényekkel. Erről részletesebben lásd [Ronyai1995]

2.1.4. Véges testek feletti elliptikus görbék

Végre elérkeztünk a kriptográfiai szempontból igazán lényeges részhez, a véges testek esetéhez. A végtelen testek azért nem voltak megfelelőek, mert számítógépekkel eredendően csak véges pontossággal tudunk számokat tudunk tárolni, ami például valós számok használata esetén számos problémához vezethet. Ennek elkerülésére dolgozunk inkább véges test felett.

A véges testek elemszáma mindig prímhatvány, a továbbiakban ezt $q = p^r$ jelöli. [Katona1997] Bizonyítható, hogy q elemszámú testből lényegében csak egy van; ennek jele $GF(q)$. Egy test karakterisztikáján azt a legkisebb c pozitív egész számot értjük, amire teljesül, hogy minden testelemet c-szer saját magával összeadva 0-t kapunk. Egy p^r elemszámú test esetén $c = p$. Véges testek multiplikatív csoportja (a $GF(q) \setminus \{0\}$ halmaz) mindig ciklikus, egy generátorelemét g jelöli.

Ezután a kis bevezető után rátérhetünk a véges testek fölötti elliptikus görbék vizsgálatára. Az a módszer, ahogy a valós esetben az (1) egyenletből 3 tagot eliminálunk, véges testek fölött nem mindig alkalmazható. Pontosabban, csak akkor alkalmazható, ha a test karakterisztikája nem 2 vagy 3. A 2-karakterisztikájú eset a legbonyolultabb: ilyenkor választhatunk, hogy az xy-os és x^2-es tagot ejtjük ki, vagy pedig az y-os és az x-es tagot. (Azonban mindezen tagokat egyszerre nem tudjuk kiejteni.) Az előbbi esetet szuperszinguláris, az utóbbit nem-szuperszinguláris nevezzük. A nem-szuperszinguláris esetben azt is fel lehet tételezni, hogy az xy-os tag együtthatója 1. 3-as karakterisztika esetén ki tudjuk ejteni az xy-os és az y-os tagot, de a jobb oldalt már nem tudjuk tovább egyszerűsíteni. Összefoglalva tehát az egyenletek a következők.

$p = 2$, szuperszinguláris eset:

$$y^2 + ay = x^3 + bx + c$$

$p = 2$, nem-szuperszinguláris eset:

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

$p = 3$:

$$y^2 = x^3 + ax^2 + bx + c$$

$p > 3$:

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

Mivel az összeadásra vonatkozó képleteket is a (2) egyenletből vezettük le, így $p = 2$ illetve 3 esetén ezek sem érvényesek. Kanyarodjunk vissza ezért az általánosabb (1) egyenlethez. Ebből is levezethetjük a megfelelő képleteket, melyek azonban korábbi formuláinknál jóval bonyolultabbak lesznek, ezért ezeket itt nem ismertetjük. Azonban azt vegyük észre, hogy mivel a görbe el van tolva és forgatva, így az (x, y) pont inverze nem az $(x, -y)$ pont lesz, hanem (az (1) egyenlet jelöléseivel) $(x, -ax - b - y)$.

A véges testek fölötti elliptikus görbék elméletében fontos szerepet játszik annak meghatározása, hogy a görbének hány pontja van. Jelölje ezt N. Az affin (tehát nem végtelen távoli) pontok tekintetében persze legfeljebb q^2 pont jöhet szóba, de N ennél lényegesen kevesebb. Ugyanis x egy konkrét értéke mellett a görbe egyenlete egy másodfokú egyenlet y-ra. Ezért $N \leq 2q$. Azonban

nem minden másodfokú egyenlet oldható meg a véges testek fölött, így N valójában még ennél is kevesebb. Hasse 1934-es tétele értelmében $N \sim q$, pontosabban N és $q+1$ eltérése $2\sqrt{q}$-val becsülhető fölülről. (Ebből következően könnyen lehet egy adott görbén véletlenszerűen pontokat keresni: veszünk egy tetszőleges x-et, és ehhez körülbelül 50% eséllyel tudunk megfelelő y-t találni. Érdekes azonban, hogy általánosságban nem ismeretes determinisztikus polinomidejű algoritmus egy adott görbén pontok megtalálására. Pontosabban, csak olyan algoritmusok ismeretesek, amikről csak akkor bizonyítható, hogy általánosságban determinisztikusan polinomidejűek, ha feltesszük, hogy az általánosított Riemann-hipotézis igaz.)

Az előző eredménynek az általánosítása mély összefüggést mutat a Riemann-sejtéssel, ezért szokták az elliptikus görbékre vonatkozó Riemann-sejtésnek is nevezni. Az $E(GF(q))$ elliptikus görbét tekinthetjük $GF(q^s)$ felett is. E görbe pontjainak számát jelölje N_s. (Tehát $N_1 = N$.) E számokból a következő formális hatványsort készítjük el (zeta-függvény):

$$Z(E(GF(q)), T) = e^{\sum N_s T^s / s}$$

Egy szintén Hasse-tól származó tétel szerint ez egy racionális törtfüggvény:

$$Z(E(GF(q))) = \frac{1 - AT + qT^2}{(1-T)(1-qT)}$$

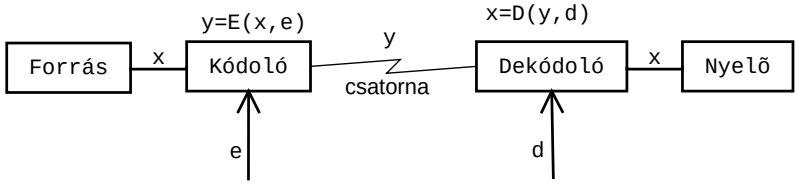
Ennek értéke kizárólag az A értékén keresztül függ az adott görbétől. Ráadásul a következő teljesül: $N = q + 1 - A$, vagyis N ismeretében A, és ezen keresztül az összes N_s meghatározható.

2.2. Nyilvános kulcsú kriptográfia

Ebben a fejezetben röviden áttekintjük a nyilvános kulcsú kriptográfia jellemzőit, előnyeit, hátrányait, néhány megvalósítási lehetőségét. Vagyis mindazokat a szempontokat, amiket a következő fejezetben, az elliptikus görbéken alapuló nyilvános kulcsú kriptográfiai rendszerek tárgyalásánál, figyelembe kell majd vennünk.

2.2.1. A nyilvános kulcsú rendszerek legfontosabb tulajdonságai

Egy kriptográfiai rendszer modelljét mutatja az alábbi ábra:



A kriptográfiai rendszert aszerint nevezzük szimmetrikusnak vagy aszimmetrikusnak, hogy az e és d kódoló illetve dekódoló kulcs egymásból könnyen kalkulálható (például megegyeznek) vagy sem. Annak érdekében, hogy egy aszimmetrikus rendszerben az egyik kulcsból a másikat ne lehessen kitalálni, rendszerint sokkal nagyobb kulcsokat alkalmaznak, mint szimmetrikus rendszerek esetében. Ennek tulajdonképpen csak akkor van értelme, ha az egyik kulcs titkos, a másik pedig

nyilvános, mivel ekkor tényleg elvárás, hogy a nyilvános kulcsból ne lehessen kitalálni a titkos kulcsot. Ezért az aszimmetrikus kriptográfiai rendszerek egybeesnek a nyilvános kulcsú kriptográfiai rendszerekkel.

A nyilvános kulcsú rendszerekben alkalmazott nagyméretű kulcsok miatt ezek általában sokkal lassabbak és erőforrás-igényesebbek, mint szimmetrikus társaik. (Érzékeltetésül: egy 128 bites szimmetrikus eljárás biztonságosabbnak tekinthető, mint az 1024 bites RSA.) Az, hogy mégis alkalmazunk nyilvános kulcsú kriptográfiát, abból fakad, hogy bizonyos dolgokat csak ezekkel tudunk megoldani. Ilyen például a digitális aláírás és a letagadhatatlanság kérdése. Egy szimmetrikus kriptográfiai rendszerben semmilyen módon nem ellenőrizhető, hogy az üzenet tényleg attól jött, akitől gondoljuk. Hasonlóan nem is lehet rábizonyítani valakire, hogy egy adott üzenetet ő küldött.

Nyilvános kulcsú rendszerekben azonban, ha valaki egy üzenetet a saját titkos kulcsával elkódol, akkor ezt bárki visszakódolhatja az illető nyilvános kulcsával, és így meggyőződhet arról, hogy az üzenet valóban az illetőtől jött-e. Ráadásul adott esetben rá is lehet bizonyítani az illetőre, hogy köldött egy ilyen üzenetet. Látható tehát, hogy azokban a gyakorlati alkalmazásokban, ahol szükség van autentikációra és letagadhatatlanságra (pl. elektronikus kereskedelem, elektronikus levelezés, elektronikus szavazás stb.) elengedhetetlen a nyilvános kulcsú kriptográfia alkalmazása.

A nyilvános kulcsú kriptográfia másik jelentős alkalmazási területe a kulcscsere. Ez tulajdonképpen a szimmetrikus és az aszimmetrikus kriptográfia szimbiózisa: mivel a szimmetrikus kriptográfia gyorsabb, ezért ezt használjuk az üzenetek titkosítására. Azonban a protokoll elején gondoskodni kell arról, hogy mindkét félnél ugyanaz a titkos kulcs legyen. Ezt a kulcsot nyilvános csatornán nyilván nem lehet átvinni, ha pedig ezt is titkosítani kell, az ördögi körhöz vezethetne. Szerencsére erre megoldást kínál a nyilvános kulcsú kriptográfia, ugyanis ehhez nem kell közös titok. Így aszimmetrikus rejtjelezést használva megegyezhetnek a felek a kulcsról, majd áttérnek a hatékonyabb szimmetrikus rejtjelezésre. (Ezen az elven működik például a biztonságos Internet-alapú kommunikációt szolgáló, népszerű SSL rendszer. [Hornak])

2.2.2. Konkrét nyilvános kulcsú rendszerek

A nyilvános kulcsú kriptográfia születésének az 1976-os év tekinthető. Ekkor javasolta ugyanis W. Diffie és M. Hellman a róluk elnevezett aszimmetrikus kriptográfiai eljárást. Ennek biztonsága az úgynevezett diszkrét logaritmus problémán (DLP) alapul. Ennek lényege, hogy egyes véges, de nagy méretű csoportokban (pl. a modulo p maradékosztályok multiplikatív csoportjában) g és a ismeretében g^a kiszámítása egyszerű, ám g és g^a ismeretében a meghatározása nagyon nehéz. Ennek megfelelően a Diffie-Hellman rendszerben, ha egy szereplő titkos kulcsa a, akkor nyilvános g^a. (g az egész rendszerben ismert állandó, a csoport generátoreleme.)

1978-ban jelent meg az RSA (kitalálóiról Rivest-Shamir-Adleman). Ennek alapját az egészek faktorizációs problémája (IFP, Integer Factorisation Problem) jelenti. Ennek lényege, hogy ha p és q két nagy prímszám, akkor ezek ismeretében könnyű meghatározni az $m = pq$ és $\varphi(m) = (p-1)(q-1)$ számokat, azonban pusztán m-ből $\varphi(m)$ meghatározása nagyon nehéz, mivel ehhez lényegében m prímfelbontását kell előállítani. Az RSA esetében, ha a felhasználó nyilvános kulcsa egy e szám, mely relatív prím $\varphi(m)$-hez, akkor titkos kulcsa egy olyan d szám, amelyre $de \equiv 1 \bmod \varphi(m)$.

Az azóta született majdnem minden nyilvános kulcsú kriptográfiai rendszer vagy a DLP, vagy pedig az IFP nagy számításigényére épít. (A már említett Diffie-Hellman rendszer mellett a DLP-

re épül még például a Massey-Omura és az ElGamal rendszer is. Tulajdonképpen az elliptikus görbéken alapuló kriptográfia is ide sorolható, bár erről megoszlanak a vélemények.) Nagyméretű kulcsok esetén ugyanis a problématér mindkét esetben olyan nagy lesz, hogy lehetetlenné válik az esetek végignézése (exponenciálisan sok eset van). Azonban két további szempontot is figyelembe kell venni. Az első az, hogy mindkét problémára találtak olyan megoldó algoritmusokat, amik nem polinomidejűek ugyan, de azért jóval hatékonyabbak az összes eset végignézésénél. Vagyis léteznek ún. szubexponenciális megoldó algoritmusok. Például a DLP esetén különböző, az index-kalkuluson alapuló szita-módszerekkel (pl. kvadratikus szita) lehet eredményt elérni. A másik lényeges pont az, hogy egy konkrét kriptográfiai rendszer készítésekor nem csak biztonsági, hanem hatékonysági, gazdaságossági stb. szempontokat is figyelembe kell venni. Ez korlátozza a felhasználható kulcsméretet.

Megemlítjük még, hogy a nyilvános kulcsú kriptográfiára vonatkozóan vannak szabványok is. Legismertebb talán az IEEE P1363, de ezen kívül idevágó munkát végez az ANSI X9F1 munkacsoportja is, valamint az ISO és az IETF. Ezekben a szabványokban egyre nagyobb szerephez jut az elliptikus görbéken alapuló nyilvános kulcsú kriptográfia is.

2.3. Elliptikus görbék kriptográfiai alkalmazása

2.3.1. ECDLP

1985-ben V. Miller és N. Koblitz egymástól függetlenül javasolták az elliptikus görbék pontjain értelmezett diszkrét logaritmus probléma (ECDLP) kriptográfiai alkalmazását. [Miller1985], [Koblitz1987] Ennek lényege a következő. Adva van egy E elliptikus görbe egy véges test felett, tovább egy $P \in E$ pont. Legyen N a P pont rendje, vagyis a legkisebb olyan pozitív egész szám, amire $NP = O$. (NP jelentése: a P pontot N-szer összeadjuk saját magával.) Legyen ezen kívül adva még egy Q pont. A feladat: találni egy olyan $0 < k < N$ természetes számot, amire $Q = kP$, feltéve, hogy ilyen szám létezik.

Ez valóban megegyezik a diszkrét logaritmus problémával, csupán a jelölés más. Ha ugyanis a csoport-operációt nem összeadásnak, hanem szorzásnak tekintenénk, akkor a feladat egy olyan k szám keresése lenne, melyre $Q = P^k$, amit így is jelölhetnénk: $k = \log_P Q$.

Persze sokkal kellemesebb a modulo p maradékosztályokkal számolni, mint egy véges test feletti elliptikus görbe pontjaival. Joggal merül fel tehát a kérdés: miért éppen egy ilyen, viszonylag nehezen kezelhető csoportban kell definiálni a diszkrét logaritmus problémát? A válasz az, hogy természetesen a diszkrét logaritmus problémának tetszőleges csoportban van értelme, azonban az elliptikus görbék segítségével olyan csoportot sikerült definiálni, amiben a diszkrét logaritmus probléma megoldására nem ismeretes exponenciálisnál alacsonyabb időigényű algoritmus. Pontosabban, az elliptikus görbék egyes eseteire ismeretesek szubexponenciális törő algoritmusok, ezért nem mindegy, kriptográfiai rendszerünkben milyen görbét vagy görbéket használunk. A legnagyobb ilyen jellegű eredmény 1993-ból származik. Menezes, Okamoto és Vanstone adott egy módszert, melynek segítségével a szuperszinguláris görbéken értelmezett ECDLP viszonylag hatékonyan oldható meg. (Erről, és más törési lehetőségekről részletesen is írunk még a biztonság elemzésénél.)

Azonban az elliptikus görbék túlnyomórészt nem szuperszingulárisak, így továbbra is nyitott kérdés, hogy van-e az ECDLP-re általánosságban használható szub-exponenciális algoritmus. Je-

lenleg úgy tűnik, hogy nincs, ezért az ECC rendszerek viszonylag kis kulcsmérettel nagy biztonságot tudnak elérni. Ezzel együtt persze, mint azt a későbbiekben látni fogjuk, a bonyolult számítások nagyban rontják a módszer hatékonyságát.

Megemlítjük még, hogy nem csupán elliptikus görbékkel lehet olyan csoportot definiálni, ahol a diszkrét logaritmus probléma nehéz. Az elliptikus görbék csoport-konstrukciója általánosítható (görbék Jacobi-felületén értelmezett csoport-struktúraként); tulajdonképpen csoportok egy sorozatának első elemével állunk szemben. Az elliptikus görbék után következnek a hiperelliptikus görbék. Azonban jelenleg úgy tűnik, ezek nem biztonságosabbak annyival, amennyivel bonyolultabb a kezelésük.

2.3.2. A véges test

A gyakorlati megvalósítás szempontjából nagyon lényeges kérdés, hogy mi az a véges test, ami fölött az elliptikus görbét értelmezzük. Rendszerint vagy egy $GF(p)$ prímrendű testet szoktak használni, vagy pedig egy $GF(2^r)$ 2-karakterisztikájú testet. Az előbbi nagy előnye, hogy egyszerűen lehet benne számolni, hiszen a szokásos módon lehet a műveleteket végezni, csupán arre kell figyelni, hogy az eredmény ismét 0 és $p-1$ között legyen. Az utóbbi viszont jobban illeszkedik a számítástechnikában alkalmazott 2-es számrendszerhez, így számos művelet gyorsabban végezhető el.

$GF(2^r)$ elemeinek reprezentálására több különböző módszer is használatos. Ezek közül a legszélesebb körben a polinom-alapú ábrázolás terjedt el. Ennek lényege, hogy tekintünk egy r-edfokú irreducibilis polinomot $GF(2)$ felett; jelöljük ezt f-fel. Ezután $GF(2^r)$ elemeire úgy tekintünk, mint legfeljebb $r-1$ fokú, $GF(2)$ feletti polinomokra. Az összeadás a polinomoknál megszokott módon történik, csak éppen arra kell figyelni, hogy a 2-karakterisztika miatt egy tetszőleges számot (vagy egy tetszőleges polinomot) önmagához adva 0-t kapunk. (Emiatt például a kivonás megegyezik az összeadással.) Két testelemet úgy kell összeszorozni, hogy polinomként összeszorozzuk őket, majd a kapott polinomot modulo f vesszük. Vagyis elosztjuk maradékosan f-fel, és a maradék (amely szintén egy legfeljebb $r-1$ fokú polinom) lesz a szorzás eredménye. Bebizonyítható, hogy így tényleg testet kapunk, és mivel a 2^r elemszámú test lényegében egyértelmű, így kész vagyunk $GF(2^r)$ konstrukciójával. Megjegyzés: a polinomokat gyakran az együtthatóikból képezett r hosszúságú 0-1 sorozatokkal adják meg.

Egy másik lehetőség, hogy $GF(2^r)$-et, mint $GF(2)$ bővítését és ebből adódóan fölötte értelmezett vektorteret tekintjük. A test elemeit tehát ismét r hosszú 0-1 sorozatokkal adjuk meg, azonban ezeknek most más a szemantikájuk. Az összeadást véletlenül most is ugyanúgy, tagonként kell végezni, mint az előző esetben. A szorzást azonban már nem feltétlenül: ez attól függ, hogy a szorzást hogyan értelmezzük a vektortér bázisán. Ez a „szorzótábla" tulajdonképpen egy $r \times r \times r$-es kocka, aminek (i,j,k) eleme az i. báziselem és a j. báziselem szorzatának a k. komponensét adja.

A szorzást jelentősen lehet gyorsítani, ha ez a szorzótábla viszonylag ritka, vagyis kevés benne az egyes. Erre szolgál az úgynevezett optimális normál bázis, aminek az alkalmazásával a lehetséges r^3 pozícióból csupán kb. $2r^2$ lesz egyes. Vannak olyan konstrukciók, melyek optimális vagy közel optimális normál bázist szolgáltatnak [Mullin1988], azonban ezek sajnos csak bizonyos speciális r értékek esetén működnek. Ráadásul úgy tűnik, a rendszer biztonságát nagyban növeli, ha r prím, és ilyenkor pont nem tudunk optimális normál bázist létrehozni.

2.3.3. Konkrét protokollok

Elméleti kirándulásunk befejezéseként nézzünk néhány példát arra, hogy hogyan lehet az ECDLP-t konkrét kriptográfiai prokollokban használni.

Vegyük például a Diffie-Hellman protokoll elliptikus görbéken alapuló változatát. Tegyük fel, hogy E egy $GF(q)$ feletti elliptikus görbe, és P ennek egy pontja. E és P mindenki által ismert. Minden szereplő választ magának egy titkos k kulcsot, mely egy pozitív egész szám. Ezek után mindenki kiszámítja kP-t, és ezt közzéteszi, mint nyilvános kulcsot. Tegyük fel, hogy két szereplő, A és B üzeneteket akarnak váltani valamilyen szimmetrikus kriptográfiai rendszerrel, és ehhez szükségük van egy közös kulcsra, melyet csak ők ketten ismernek. Ha titkos kulcsaik k_A és k_B, akkor k_Ak_BP pont megfelel. Ugyanis A ezt úgy tudja meghatározni, hogy B nyilvános kulcsát, k_BP-t, megszorozza k_A-val, hasonlóan **B** úgy határozza meg, hogy A nyilvános kulcsát, k_AP-t megszorozza k_B-vel. Viszont k_A és k_B ismerete nélkül senki más nem tudja kiszámítani ezt, hacsak nem oldja meg az ECDLP-t.

Nézzük most az ElGamal protokoll elliptikus görbéken alapuló változatát. A kiindulás ugyanaz, mint az előbb, de most tegyük fel, hogy B akar küldeni egy M üzenetet A-nak. Feltesszük, hogy valamilyen módon az üzenet az E görbe pontjaként van elkódolva. Ekkor B választ egy tetszőleges l természetes számot, és elküldi A-nak az lP és $M + l(k_AP)$ pontokat. Ebből l és k_A ismerete nélkül nem lehet visszanyerni az M üzenetet. Viszont A a kapott első pontot megszorozza k_A-val, és ezt kivonva a második pontból épp M-et kapja.

Természetesen a fenti rendszerek biztonságához egy elég nagy méretű test feletti görbe kell, melynek elég sok pontja van, továbbá a kiválasztott P pont rendjének is nagynak kell lennie.

3. Smart Card-os alapok

3.1. Általában

3.1.1. Bevezetés a smart cardok világába

A chipkártyák (smart card) mindennapjaink részévé váltak: bankkártyák, telefonkártyák, igazolványok formájában lépten-nyomon találkozunk velük. Azonban ezek az alkalmazások a modern chipkártyák technikai lehetőségeinek csupán egy töredékét hasznosítják.

3.1.2. Történelem

A mai chipkártyák ősei az 1950-es évek óta (és mind a mai napig) széles körben alkalmazott mágneskártyák. Az ezeken elhelyezett mágnescsíkon néhány száz byte adatot lehet tárolni, mágneses úton olvasni, esetleg írni. Az 1970-es években merült fel mikroelektronikai áramkörök alkalmazásának lehetősége. A francia Bull cég 1979-ben készítette el első mikroprocesszorral is rendelkező kártyáját, melyen azonban a processzor és a memória még külön chipen helyezkedett el. Ez nem bizonyult kellően megbízható megoldásnak. A technikai fejlődés azonban csak az 1980-as években tette lehetővé az összes áramkör egyetlen chipre való integrálását. Franciaországban 1986 óta használnak chipkártyákat az utcai telefonokhoz.

A chipkártyák számos előnnyel rendelkeznek a mágneskártyákkal szemben. Egyrészt kevésbé érzékenyek mágneses zavarokra, másrészt pedig biztonságosabbak, hiszen a kártya processzora

kontrollálja az olvasási és írási műveleteket. Ezáltal az adatokhoz közvetlenül nem, hanem csak közvetetten, előre definiált műveleteken keresztül lehet hozzáférni.

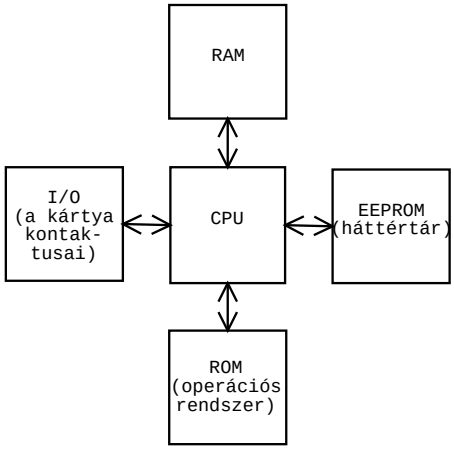
Természetesen a fejlődés azóta sem állt meg. Az intelligens kártyák egyre nagyobb memóriával és egyre nagyobb teljesítményű processzorral rendelkeznek. Ez lehetővé teszi, hogy a kártya ne csak végrehajtsa a kívülről érkező parancsokat, hanem önálló alkalmazások fussanak rajta (esetleg több is), a kártya operációs rendszere fölött. Az ezzel a tulajdonsággal rendelkező kártyák a programozható chipkártyák, melyek számos új előnyt hordoznak. Először is lényegesen nagyobb rugalmasságot: csupán egyféle kártyát kell legyártani, mely ezután mindig újabb és újabb funkciókkal látható el újabb szoftver feltöltésével. Ebből adódóan a kártyákat elég a gyártási folyamat után specializálni, tehát kevesebb kártyatípusra van szükség, amelyek viszont így nagyobb példányszámban gyárthatók. Ez egyrészt a smart card technológia árának csökkenéséhez vezet, másrészt pedig ahhoz, hogy kisebb cégek is képesek kártyákat kiadni.

A programozható chipkártyák másik nagy előnye, hogy megfelelő operációs rendszer támogatás esetén, a kártyán egyszerre több alkalmazás is lehet. Ez megnyitja annak lehetőségét, hogy személyenként egyetlen kártya ellásson minden azonosítási, hitelesítési, adattárolási funkciót. Természetesen ez komoly problémákat vet fel, hiszen garantálni kell azt, hogy a rendszer minden szereplője pontosan annyi információt tudjon kinyerni a kártyából, amennyire jogosult. Végezetül pedig a programozható chipkártyák lehetőséget biztosítanak fejlett kriptográfiai algoritmusok és protokollok megvalósítására, amelyek nagymértékben növelhetik a rendszer biztonságát.

Fontos azonban említést tenni a programozható chipkártyák korlátairól is. A kártya fizikai biztonsága érdekében a teljes hardvert egyetlen chipben kell megvalósítani, hiszen különben a kártya buszrendszerére kapcsolódva értékes információkat lehetne nyerni. Így a processzor és a memória összteljesítményét korlátozza, hogy a termelődő hő elvezetése nincs megoldva. A másik kritikus faktor a kártya előállítási költsége. Ez már így is többszöröse az egyszerűbb kártyák költségének, bár nagyobb példányszám esetén nyilván drasztikusan csökkenthető. Minden esetre ez is gátat szabhat a teljesítmény növelésének.

3.1.3. Programozható chipkártyák felépítése

Mint az alábbi ábrán látható, egy programozható chipkártya belső felépítése (blokkvázlata) nagyban hasonlít a hagyományos Neumann-architektúrájú számítógépek felépítéséhez. Szintén egy CPU vezérel, s hozzákapcsolódik a memóriaegységekhez és a perifériákhoz.



A fő különbség az, hogy míg a Neumann-architektúrában mindezek egyetlen központi buszra kapcsolódnak, egy chipkártya esetén nincsen ilyen busz. Itt a CPU közvetlen kapcsolatban áll a másik négy fő egységgel, s azok egymással nincsenek összekötve.

A chipkártyák tervezésénél a legfőbb szempont a biztonság. Nem engedhető meg, hogy egy I/O periféria közvetlenül meghajtsa a buszt, s így valamilyen módon utasításokat küldhessen a memóriaegységeknek. [Zoreda1994]

Egy asztali PC-hez képest jeletősen lecsökkentették az I/O felületet. A rendszerben egyetlen, jól definiált I/O periféria van, a kártyán lévő kontaktusok. Ezzel is azt kívánják biztosítani, hogy a kártya működésébe minél kevésbé tudjanak beavatkozni a külső folyamatok.

3.1.4. Biztonságos mikroszámítógép

Mint látjuk, a programozható chipkártya rendelkezik processzorral, memóriával és háttértárral. A ROM-ban található a kártyán futó operációs rendszer (ez természetesen nem cserélhető le), az EEPROM-ban pedig a kártyán lévő adatok és programok helyezkednek el. Az EEPROM - mint nem illékony memória - szolgál a kártya háttértáraként. A RAM-ban helyezkednek el a dinamikus változók és a stack.

Egy programozható chipkártya rendelkezik minden olyan hozzávalóval, ami egy számítógéphez szükséges. Így amikor róluk szólunk, nem pusztán passzív adattároló eszközökről beszélünk, hanem aktív elemekről, amelyek adataikról dönteni képesek, és akár fel is dolgozhatják őket. Egy chipkártya valójában nem más, mint egy biztonságos mikroszámítógép.

3.1.5. Szereplők bemutatása

Egy chipkártya alapú rendszer tipikus szereplői a kártya gyártója (card manufacturer), kibocsátója (card issuer) és birtokosa (card holder). Ezek közül az első kettő egybeeshet. További szereplő lehet a terminálok tulajdonosa (terminal owner); ez például a telefonkártya szolgáltatás esetén rendszerint a telefontársaság (amely egyben a kártya kibocsátója is), de például bankkártyák esetén lehetőség van más bank termináljainak, vagy elektronikus kártyaelfogadóhelyek (Electronic Point of Sale) használatára. Tovább bonyolíthatja a helyzetet, hogy a kártya birtokosa nem feltétlenül egyezik meg a kártyán lévő adatok tulajdonosával (data owner); jó példa erre a hitelkártya, ahol a kártyán lévő adat tulajdonosa a bank, hiszen csak ő tudja ezen adatokat megváltoztatni. Programozható kártyák esetén nem egyezik meg a kártya gyártója a kártyán futó programok készítőivel.

Mindez azért rendkívül fontos, mert a szereplők számával arányosan nő a támadási lehetőségek száma is. (Támadás alatt itt természetesen adatok elleni támadást értünk, aminek célja lehet például egy szolgáltatás illetéktelen használata, közvetlen anyagi haszon, titkos információk megszerzése stb.) Ha csupán egyetlen szereplő volna, akkor kommunikáció híján a rendszer a lehető legbiztonságosabb lenne. Azonban, amint áttérünk egy kétszereplős modellre, azonnal megjelenik a lehetőség a két fél közti csatorna megtámadására, vagy arra, hogy az egyik fél támadást intézzen a másik ellen. Hasonlóan a funkciók minden további bontása újabb támadási lehetőségeket kínál.

A rendszer elleni támadások aszerint csoportosíthatók, hogy a támadó mely komponensen, illetve mely komponensek közötti csatornán intéz támadást a rendszer ellen. A támadó meg is egyezhet valamelyik szereplővel. Például lehet a támadó a kártya birtokosa (telefonkártya végtele-

nítése), vagy akár a szoftvergyártó is (kiskapuk elhelyezése), másrészt az is előfordulhat, hogy egy külső támadó a terminálokat veszi célba (hamis bankautomaták).

Jól látható tehát, hogy a programozható chipkártyák használatának ára van: új szereplők jelennek meg, ezzel új támadási lehetőségeket biztosítva. Ezt egyrészt figyelembe kell venni chipkártya alapú rendszerek tervezésénél, másrészt a chipkártyák által nyújtott kriptográfiai szolgáltatásokat is ennek fényében kell értékelni.

3.1.6. Egyéb felhasználási területek

A fenti példák (telefonkártyák, bankkártyák stb.) mellett számos más rendszer is használ chipkártyákat, például a mobil távközlésben használatosak a SIM-kártyák. Ígéretes a chipkártyák internetes alkalmazásának lehetősége is, amelynél a következő funkciók körvonalazódnak:

* Login információk (felhasználói azonosító, jelszó) tárolása kártyán;
* Login dinamikus jelszó segítségével (pl challange and response elven);
* Erőforrás-hozzáférési jogosultságok kezelése;
* Elektronikus levelek titkosítása;
* Üzenetek hitelesítése és kriptográfiai ellenőrzőösszeggel való védelme;
* Digitális pénz az elektronikus kereskedelemben.

A csak Internet-hozzáférésre szolgáló, gyakran nyilvános számítógépek integritásának megóvása további kihívásokat jelent. Gondolunk itt első sorban az Internet kávézók és teleházak PC-ire és NC-ire (Network Computer). Ezen gépek állandó támadási lehetőségeknek vannak kitéve (amibe véletlen rongálás, például egy vírus letöltése is beletartozik), így semmiképp sem tekinthetők megbízhatónak. Ilyen esetekben olyan autentikációs protokoll alkalmazása látszik megfelelőnek, melynek során nemcsak a számítógép ellenőrzi a kártya tulajdonosának jogosultságait, hanem a gép is azonosítja magát a kártya felé. Sőt, elképzelhető az is, hogy a nyilvános gépek adminisztrátora bizonyos időközönként egy chipkártyával ellenőzi a gépek, illetve a rajtuk lévő szoftver integritását. Ebben az esetben a kártya tekinthető a megbízható félnek. Az integritás ellenőrzésére a számítógép memóriaképének egy egyirányú függvénnyel képzett tömörítvényét kell a kártyára tölteni, illetve ellenőrizni. (Egyirányú függvény alatt olyan függvényt értünk, melynek inverzét csak próbálgatással lehet meghatározni, ez pedig elég nagy adattér esetén reménytelenül hosszú időt vesz igénybe.)

Probléma marad azonban az, hogy a kártya - saját megjelenítő eszköz hiányában - csak a számítógépen keresztül tud jelzést adni a felhasználónak, illetve adminisztrátornak. Márpedig ha a számítógépet nem tekintjük megbízhatónak, a kapott eredményt sem tekinthetjük annak. Nagymértékben növelné a chipkártyák biztonságát, ha legalább egy LED segítségével jelezhetnék a felhasználónak, ha valami problémát észlelnek.

3.1.7. Chipkártyák programozása

A chipkártyák jellemző paraméterei (tipikusan 3-8 MHz órajelű 8 bites processzor, 8-32KB memória) az 1970-es évek számítógépeire emlékeztetnek. Ezt súlyosbítja az a körülmény, hogy a kártyák nem tudnak közvetlenül a külvilággal kapcsolatot tartani, csupán a kártyaolvasón keresztül. Ez nagyban nehezíti a hibakeresést. További probléma a rendszer lassúsága és a programírás folyamatának körülményessége (az elkészült programot gyakran több lépésben lehet csak a kártya által

ismert formátumra hozni, ezután pedig fel kell tölteni a kártyára). A nehéz programozhatóságért cserébe egy kompakt, biztonságos kis számítógépet kapunk.

További sajátságos probléma, hogy a felhasználó bármikor megszakíthatja a folyamatban lévő műveletet a kártyának az olvasóból történő kihúzásával, s így annak adatai inkonzisztens állapotban maradhatnak. Ez ellen olyan terminálokkal szokás védekezni, melyek a kártyát elnyelik, és csak a művelet végén adják vissza. Azonban ez a felhasználókból gyakran bizalmatlanságot vált ki. Gondos programozással megvalósítható tranzakció kezelés a kártyán (de a hardware támogatás jelentősen megkönnyíti a munkát). Ez az adatbáziskezelők világából kölcsönzött kifejezés azt jelenti, hogy a rendszer a kritikus műveletekről garantálja, hogy vagy teljesen befejeződnek, vagy pedig (ha valamiért ez nem lehetséges) a rendszer visszajut a tranzakció előtti állapotába.

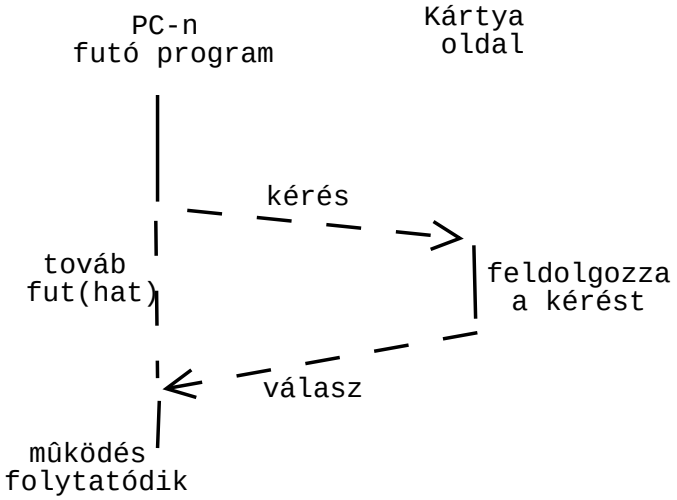
A kártyával való kommunikáció alapegysége az APDU (Application Protocol Data Unit). A kártyával történő kommunikáció tipikusan félduplex (bár a szabvány lehetőséget biztosít duplex kommunikációra is): a terminál egy APDU-val elindít egy műveletet a kártyán, amely a művelet végeztével válaszol(hat). Az ISO 7816 szabvány leegyszerűsítve az ábrán látható parancs-APDU struktúrát fekteti le. [Rankl-Effing1997]

CLA	INS	P1	P2	LC	Data field	LE

Az egyes utasítások utasításosztályokba (CLA) sorolhatók. (pl: szabványos ISO utasítások, GSM utasítások stb), az utasítás (INS) kódja pedig a csoporton belül választja ki az utasítást, tehát kódjaik együtt azonosítják a végrehajtandó műveletet. P1 és P2 opcionális paraméter, ezeket követheti az adatok hosszát jelző byte, maga az adatmező, majd a várt válasz hossza. A kártya válasza adatokat s két státusz szót tartalmaz.

SW1	SW2	Data field

A PC-kártya kommunikáció egyáltalán nem szimmetrikus. Igaz, hogy a kártya is tekinthető számítógépnek, de nem rendelkezik saját tápegységgel, s a rajta futó program futása nem folyamatos. Minden esetben a PC a kezdeményező, s a működés erősen függvényhívás-szerű [Berta-Mann1999]:



3.1.8. Objektum orientáltság?

Egy PC-chipkártya kommunikáció esetén nincs értelme írási vagy olvasási műveletekről beszélni. Mindössze „kérést" küldhetünk a kártyának, amelyet annak operációs rendszere valamint a rajta futó programok „megfontolnak", majd vagy teljesítik, vagy visszautasítják.

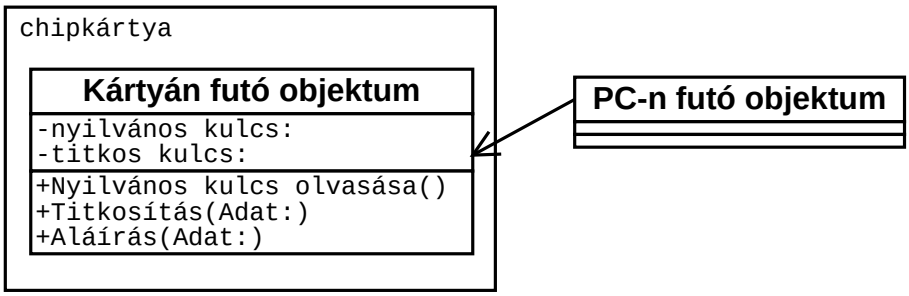
Tehát bizonyos „kapukat" definiálunk az adataink felé, s csak azokon keresztül érhetjük el őket. Ha a kapukat megfelelően definiáljuk, az ő segítségükkel biztosíthatjuk a kártyán lévő adatok mindenkori konzisztenciáját.

Például ha egy telefonkártyát vizsgálunk, a rajta lévő egységekre a következő műveleteket értelmezhetjük:

- Megmaradt egységek számának kiolvasása
- Egységek számának csökkentése n egységgel

Látható, hogy a fenti listában nem szerepel az „egységek számának növelése" vagy „egységek számának" megváltoztatása művelet. Ha a chipkártya hardvere sikeresen biztosítja, hogy a kártyán lévő egységeket csakis ezen két művelettel érhetjük el, akkor elértük azt, hogy egy csaló semmiképp nem növelheti meg az egységeinek számát.

Adatainkat elrejtjük a külvilág elől, s őket csakis a rajtuk értelmezett műveleteken keresztül érhetjük el. Ez a koncepció nagyon hasonlít az objektum orientált világból ismert encapsulation (egységbe zárás) és information hiding (információ elrejtés) fogalmának. Bizonyos szemszögből nézve a chipkártyán futó program is egy objektum.



Chipkártyák segítségével megvalósíthatjuk az objektum-orientáltságot a fizikai valóságban is. Úgy adhatunk valakinek át adatokat, hogy szigorúan definiálhatjuk azt is, hogy milyen műveleteket végezhet el vele. Például kiköthetjük azt, hogy csak egyszer olvashatja el őket. [Berta-Mann-2000]

A következő fejezetben megismerkedhetünk a programozható chipkártyák egy igen jelentős fajtájával, a Java Card-okkal.

3.2. Java Card-os rész

3.2.1. A Java Card specifikáció

A Sun Microsystems által kifejlesztett Java Card specifikáció fő célja az, hogy csökkentse a programozható kártyákba való befektetés kockázatát, s ezzel elősegítse a technológia fejlődését. Amíg a chipkártyákat pusztán assembly nyelven lehet programozni, a fejlesztő cégek kiszolgáltatottá válnak a kártyagyártóval szemben, mert minden chipkártya-típusnak különböző assembly nyelve

lehet. Ha egy alkalmazást át kell vinni egyik kártyafajtáról egy másikra - vagy ugyanannak a típusnak egy másik altípusára - a teljes alkalmazást át kell írni. Ennek orvoslására született a Java Card programozási környezet (API). [JavaCard1997]

A Java nyelv kifejlesztésének egyik célja beágyazott rendszerekben való alkalmazás, a másik pedig a weblapokon megjelenő programocskák (appletek) készítése volt, amelyek intelligens funkciókat vittek az addig javarészt statikus weblapokba. Ehhez képest gyökeresen más kihívást támasztott a Java Cardok esete. Míg egy Web-böngészőt futtató PC gyors processzorral rendelkezik, amely képes lehet appleteket hatékonyan futtatni, egy chipkártya néhány Mhz-en járó 8 bites processzora és néhány kilobyte memóriája merőben más sebességviszonyokat jelentenek. Így a Java Cardok nyelve, bár szintaktikailag Java, valójában sokkal közelebb áll a C-hez.

Egy Java Card applet forráskódját nézegetve feltűnik, hogy a kód alacsony szintű. Kevés az objektum-létrehozás, s rengeteg a bitművelet és shiftelés. Látszik a kódon, hogy a programozónak minden órajel ütemért meg kell küzdeni. A Java nyelven nem ilyen filozófiával szokás programozni.

A másik probléma az, hogy a kártyán lévő Java Virtuális Gép (JVM) nem támogatja a szemétgyűjtést (garbage collection), így ha létrehozunk egy objektumot, azt nem tudjuk elpusztítani. Az applet és az általa létrehozott objektumok élete addig tart, míg az alkalmazást le nem töröljük a kártyáról. A fentiekből - ha a program robosztusságát mindenek felettinek tartjuk, ami például egy digitális pénztárca alkalmazásnál igen fontos szempont - logikusan adódó kötöttség, hogy a programozó köteles minden objektumot a konstruktorban létrehozni, s így oldja meg azt, hogy a memória el ne fogyjon menet közben.

Minden kártyás alkalmazás együttműködő objektumok egy nem üres halmaza, melyeknek egyike a javacard.framework.Applet leszármazottja. Ezen objektum egyes metódusai szolgálnak az alkalmazás belépési pontjaként. A működés módja lényegében a következő: miután az applet feltöltésre kerül, a rendszer meghívja annak install() metódusát. Az install() általában létrehozza az applet egy példányát, s meghívja a register() metódust. Futtatáshoz ki kell választani az adott appletet a megfelelő APDU-val, s a rendszer ekkor meghívja annak select() metódusát. A Java Card appletnek nincs sokféle eseménykezelője, hiszen csak egyféle "közönséges" esemény történhet vele: a futtatás. Ilyenkor a process(APDU) metódus hívódik meg. Ez megkapja paraméterként a teljes APDU-t, s feldolgozhatja azt. Miután befejezzük a munkát az applettel, a rendszer meghívja annak deselect() metódusát.

A Java Card specifikáció támogatja a tranzakciókezelést. A programozó megadhat olyan blokkot, amely kívülről atominak minősül. Tranzakció közben a kiírandó adatok egy pufferben tárolódnak, s csak akkor kerülnek ténylegesen kiírásra, ha a tranzakció véget ér. Sajnos minden puffer mérete véges, s ha betelik, akkor a tranzakciókezelés nem működik. Ennek elkerülésére gondos programozás szükséges.

Hogy történik egy Java Card fejlesztés? Egy Java Card program byte kódja ingyenes eszközökkel (pl. JDK 1.1.7) elkészíthető. Így létrehozhatunk osztályokat, melyek egyike a javacard.framework.Applet leszármazottja. Az appletet konvertálni kell egy - a kártyagyártótól származó - konvertáló programmal. Ez kiszűri a nem Java Card konform elemeket, s az adott kártyatípus gépi kódjára konvertálja a programot. Ez egy az egyben feltölthető a Java Cardra, majd a megfelelő APDU segítségével meg kell hívni az install() metódusát. Ezután az applet működőképes.

3.2.2. A specifikáció Bull-féle megvalósítása

A Bull Odyssey 1.2 a Java Card specifikáció egy megvalósítása. Sajnos a fejlesztőkörnyezet annyiban nem tesz eleget a specifikációnak, hogy kriptográfiai funkciókat nem tartalmaz. Az általunk vizsgált kártya 8 kilobyte EEPROM-mal rendelkezik. Használhatunk emellett a lokális változók átmeneti tárolására 512 byte RAM-ot is. Megfelel az ISO 7816 szabványnak, s így elfogadja a szabványos APDU-kat.

A Java Card specifikáció az int típust opcionálisan megvalósíthatónak definiálja. Az Odyssey I támogatja a Java int típusát, de a dokumentáció megjegyzik, hogy implementációja a 8 bites processzor miatt korántsem hatékony.

A Bull fejlesztőeszköze, az OdysseyLab néhány utility mellett három programból áll. A konvertáló program platform-független byte kódról a kártya gépi kódjára fordít. A feltöltő program menedzseli a kártyán lévő appleteket. Ennek segítségével lehet letörölni a kártyát, illetve feltölteni egy új appletet. Végül a harmadik programnak már nem a fejlesztés, hanem a tesztelés fázisában van szerepe: egy egyszerű terminálfelületet biztosít a kártyával való kommunikációra. Lehetőség van scriptek, függvények írására, s a nyelve támogat egyszerű elágazási- illetve ciklusműveleteket is.

E mellett tartalmaz még a Bull CD-je egy JDK 1.1.7-et, dokumentációt, s néhány alkalmazást, például a kártyaolvasó detektálására. Nem tartalmaz viszont semmilyen programot vagy programozási könyvtárat, amely a PC oldali feljesztést lehetővé tenné.

4. A mi implementációnk

4.1. Általános megfontolások

Saját ECC implementációnkat kettős céllal hoztuk létre. Egyrészt jobban meg kívántuk ismerni az ECC technológiát, s különféle méréseket akartunk elvégezni, másrészt létre akartunk hozni egy chipkártya alapú ECC megvalósítást. Végül egy olyan programot hoztunk létre, amely egyaránt fut PC-n és Java Cardon, vagyis ugyanaz a forráskód, valamint ugyanazon Java osztályok futnak mindkét platformon.

Mivel a Java Cardot csakis Java nyelven programozhattuk, a fejlesztést Java nyelven végeztük. A Java [Java] egy magasszintű nyelv, amely nagyfokú platformfüggetlenséget tesz lehetővé. Ebből következik viszont, hogy egy Java program korántsem optimális egy adott hardverre.

A Java Card nyelv számos erős megszorítást tartalmaz a Java nyelvhez képest, s így programunkban is a nyelvnek csupán egy részhalmazát használhattuk. Célunk kifejezetten nem valamely platformon való nagy sebesség elérése volt, hanem az, hogy alkalmazásunk működjön a kártya adta szűkös erőforráskeretek között is.

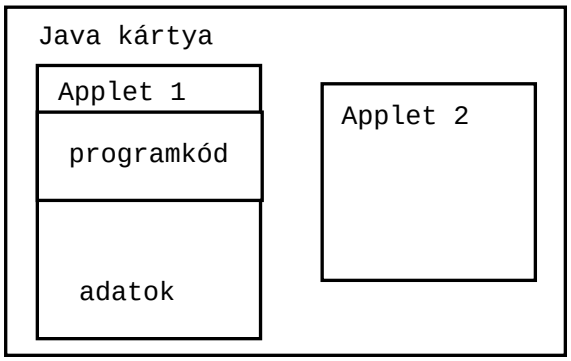
Igaz, a chipkártya mind sebesség, mind tárkapacitás terén alulmúlja az asztali számítógépeket, mindkét paraméterük jelentősen fejlődött az utóbbi időben. A fő hangsúlyt a biztonság megvalósíthatóságára és az algoritmikus hatékonyságra fektettük, nem pedig a hétköznapi értelemben vett sebességre.

Implementációnk továbbá elsősorban prototípus értékű, amely demonstrálni kívánja a Java Card technológia jelen fejlettségi szintjét, számítási erejét, nem pedig egy majd kereskedelmi forgalomba kerülő termék.

4.2. A Java Card technológiából adódó korlátok

4.2.1. A kártya hardveréből adódó korlátozások

Megvalósíthatóság terén fő korlátnak a kártya szűkös memóriája bizonyult. Nemcsak a program forráskódja kellett, hogy a kártyán elférjen, de a kártya szűkös memóriájának tartalmaznia kellett a program által használt adatokat is. Ezek közt helyet kell szorítani mindannak a nem kevés változónak is, amelyet a program futása közben használunk. Így az előtt a választási lehetőség előtt álltunk, hogy ha bonyolultabb (s ettől gyorsabb) algoritmust írunk, nem marad memóriánk megfelelő méretű testet választani a kellő biztonság eléréséhez. (Programunkban a naiv eljárásokat használtuk a műveletek megvalósítására, s appletünk ennek ellenére kitette a kártya memóriájának több, mint felét.)



Komoly korlátot jelent egy chipkártyás alkalmazás esetén a sebesség is, de ez a megvalósíthatóság tényét nem befolyásolta. Kriptográfiai alkalmazás esetén gyakorlati jelentősége csupán hardveresen támogatott műveleteknek lehetséges. Amennyiben az ECC a későbbiekben elterjed, várható, hogy a megjelenő ECC-t támogató kripto-koprocesszorok jelentősen megnövelik a számítási sebességet. (Mint az tapasztalható volt a DES és RSA chipek esetében) [Schneier1996]

4.2.2. A Java Card specifikációból adódó korlátozások

A fő korlátozást itt is a memória jelentette. A Java specifikációtól eltérően a Java Card ugyanis nem támogatja a garbage collectiont. Míg Javában azon objektumok, amelyek elérhetetlenné válnak, önmaguktól megsemmisülnek, Java Card nyelven ez nem történik meg. Nincs mód továbbá a lefoglalt memória explicit felszabadítására sem (mint ahogy ilyet Java nyelven sem tehetünk). A specifikáció szerint egyáltalán nincsen garancia arra, hogy a memória, amelyet lefoglalunk, valaha is felszabadul. [JavaCard1997]

Egy biztonságtechnikai beágyazott rendszerben, mint aminek a Java Card is tekinthető, nem megengedhető, hogy a rendszer nem elérhető állapotba kerüljön. Kerülnünk kell a legkisebb esélyét is annak, hogy az adataink inkonzisztens állapotba kerüljenek, hiszen számos adatot - éppen a kártya védelmi rendszere miatt - később nem tudunk konzisztens állapotba visszaállítani. Egy kártyán futó program igen komoly bajba kerülhet, amennyiben hirtelen elfogy a kártya memóriája.

Megjegyezzük, hogy ez egyáltalán nem biztos, hogy az alkalmazás írójának a hibája, hiszen a kártyán több egymástól függetlenül működő, és egymástól független forrásból származó, program fut egymás mellett, s amennyiben valamelyik mérete megnő, lehet, hogy a többi alkalmazás ezután nem jut kellő memóriához, s nem tud „lélegezni". Így azt biztosítani, hogy az alkalmazásunk

valamely időpillanatban biztonsággal lefoglalhasson magának további memóriaterületeket, elvileg sem lehetséges.

Így a specifikáció ajánlása [JavaCard1997], mely szerint ajánlatos az applet konstruktorában lefoglalni minden szükséges memóriát, nem csupán egy erőforrás-szegénységből adódó korlátozás, hanem egy alapvető biztonságtechnikai elv chipkártyás környezetben. Ugyanakkor ez merőben ellentmond a Java programozás filozófiájának, ahol a program „lélegzik".

4.3. Algoritmikus megoldások

4.3.1. Koncepciónk

Úgy döntöttünk, a $GF(2^m)$ testben valósítjuk meg az ECC algoritmust a kártyára. Ennek fő oka az volt, hogy számítógépekkel viszonylag könnyebb bitvektorokat feldolgozni, mint a $GF(p)$ testben számolni. Egyrészt az elvégzendő műveleteket a bináris világban kell elvégeznünk, másrészt az algoritmus is jóval tömörebb, egyszerűbb lett. Szintén a kis méretű programkód miatt döntöttünk a polinomiális reprezentáció mellett.

4.3.2. Aritmetika

A kód méretének csökkentése végett a tömör, spártai egyszerűségű algoritmusokat választottuk, akkor is, ha ez nem a leggyorsabb kódot eredményezte. A következő műveleteket valósítottuk meg a testelemek között (m az input testelem(ek) hosszát jelenti):

- összeadás: a bitenkénti kizáró vagynak felel meg (polinomiális reprezentációban ez a triviális megoldás), $O(m)$ lépésben elvégezhető.
- elforgatás (shiftelés): A balra való shiftelés a polinomok világában x-szel való szorzást jelent, a jobbra való forgatás pedig x-szel való maradékos osztást. Az implementációban ez úgy jelenik meg, hogy minden egyes bitnek értékül adjuk a jobbra/balra mellette lévőt, majd (balra shiftelés esetén) modulust képzünk. Költsége $O(m)$.
- modulusképzés: P polinomnak akarjuk a modulusát képzni M modulussal. Ez a művelet elvégezhető $GF(2^m)$ felett egyszerű összeadással is, amennyiben a P foka ($deg(P)$) megegyezik M fokával. Ha $deg(M) > deg(P)$, nincsen szükség modulusképzésre, de ha $deg(M) < deg(P)$, a modulusképzés bonyolult művelet lenne. Mi programunkban úgy intézzük, hogy ez utóbbi eset ne fordulhasson elő. Ha $deg(M) = m$, minden polinomot $m + 1$ biten reprezentálunk, ahol a testbe tartozó polinomok esetében a MSB mindig 0. Amennyiben MSB nem 0, szükség van modulusképzésre. Nem engedjük meg, hogy egy polinom fokszáma m fölé nőjön, ha elérné m-et, azonnal képezzük a modulusát. Ezzel a módszerrel azt értük el, hogy egy P polinomról, $O(1)$ lépésben eldönthetjük, hogy esetében szükség van-e modulusképzésre. Ezek után $O(m)$ lépésben képezhetjük a modulusát (ha szükséges). Ez nem biztos, hogy a leghatékonyabb megoldás, de a memóriával nem bánhattunk pazarlóan. A modulusképzés költsége összességében $O(m)$.
- szorzás: Az írásbeli szorzás algoritmusának megfelelően valósítottuk meg. A szorzás eredményét egy akkumulátorban tároljuk, melynek kezdeti értéke 0. Ha P és Q polinomokat kívánjuk összeszorozni, végigiterálunk Q bitjein. Ha az aktuális bit 1, akkor P-t hozzáadjuk

az akkumlátorhoz. Ha a bit 0, nem csinálunk semmit. Minden iteráció végén megszorozzuk P polinomot x-szel. (Azaz balra shifteljük eggyel.) Ha szükség van rá, modulust képzünk. Összeadásokat ($O(m)$) végzünk el m-szer. Az összköltség $O(m^2)$. (Ez persze még jelentősen gyorsítható.)

- maradékos osztás: Az írásbeli osztás műveletét hajtjuk végre. Tegyük fel, hogy P polinomot akarjuk Q polinommal elosztani, az osztás eredménye kerül majd E-be, a maradék pedig R-be. Első lépésként megszorozzuk Q-t x^k-nal ($Q' = Q * x^k$), úgy, hogy $deg(P) = deg(Q')$ teljesüljön. Így Q' utolsó k helyiértéke 0 lesz. Ezután a következőket végezzük el k-szor: Ha $deg(P') = deg(Q')$, akkor $P' := P + Q$ és $E := E + x^k$, de a feltételtől függetlenül elforgatjuk Q polinomot jobbra és k-t csökkentjük eggyel. Amikor k nulla lesz, akkor P'-ben keletkezett a maradék. Költsége: shifteléseket $O(m)$ és összeadásokat ($O(m)$) végzünk el legfeljebb m-szer. Így az összköltsége $O(m^2)$.

- osztás, azaz inverzzel való szorzás: Két részből áll. Először inverzet képzünk az Euklideszi algoritmus [hiv] segítségével. Ezután szorzunk $O(m^2)$. Az euklideszi algoritmusban $O(m)$ darab maradékos osztást ($O(m^2)$) végzünk. Így az összköltség $O(m^3)$.

- negálás: minden P bináris polinom negáltja önmaga.

A következő műveleteket definiáltuk a görbe pontjai alkotta csoportban:

- P pont összeadása egy tőle különböző Q ponttal: Legköltségesebb művelete a testelemek közt elvégzett osztás, így költsége $O(m^3)$.

- P pont összeadása saját magával: Itt is az osztás a legköltségesebb művelet. Költsége: $O(m^3)$.

- P pont szorzása k egész számmal: Mivel erre közvetlen képlet nem létezik, összeadásokkal kell megvalósítani az előbbi két művelet segítségével. Mivel a naiv megoldás (k darab összeadás) lépésszáma k hosszával exponenciálisan arányos lépésszámú algoritmust eredményezne, kicsit ravaszabbnak kell lennünk. Az általunk választott megoldásban kiszámítjuk a $P = l(i)$ értékeket, ahol $l(i) = 2^i$, majd k bináris reprezentációját felhasználva összeadásokkal állítjuk elő $k * P$-t: $O(log(k))$ összeadás. Hogyha egy akkumulátort használunk, s k MSB-jétől az LSB-je felé haladunk, s egyes bit esetén hozzáadjuk P értékét az akkumulátorhoz, s továbblépéskor megduplázzuk az akkumulátort, akkor mindez megoldható konstans méretű tárban is, ami a chipkártya szűkös erőforráskészletét figyelembevéve nem utolsó. A művelet algoritmikus komplexitása $O(m^3 log(k))$. Figyelembe véve a Hasse-tétel azon következményét, miszerint egy $GF(q)$ feletti görbe pontjainak száma jól közelíthető q-val, feltehetjük, hogy $k \leq q$, ugyanis a görbe pontjainak számánál nagyobb k-nak nem lenne értelme. Így $log(k)$-t közelíthetjük m-mel, tehát a kapott költség: $O(m^4)$.

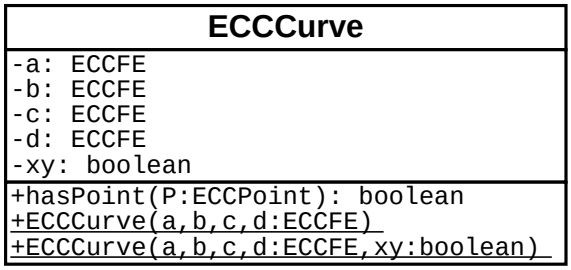
4.4. A használt osztályok bemutatása

4.4.1. Technológia

A programunk megtervezéséhez az UML módszertant [UML] használtuk, működési elvét, szerkezetét is ennek segítségével mutatjuk be. Az elkövetkezőkben olyan részleteket közlünk UML

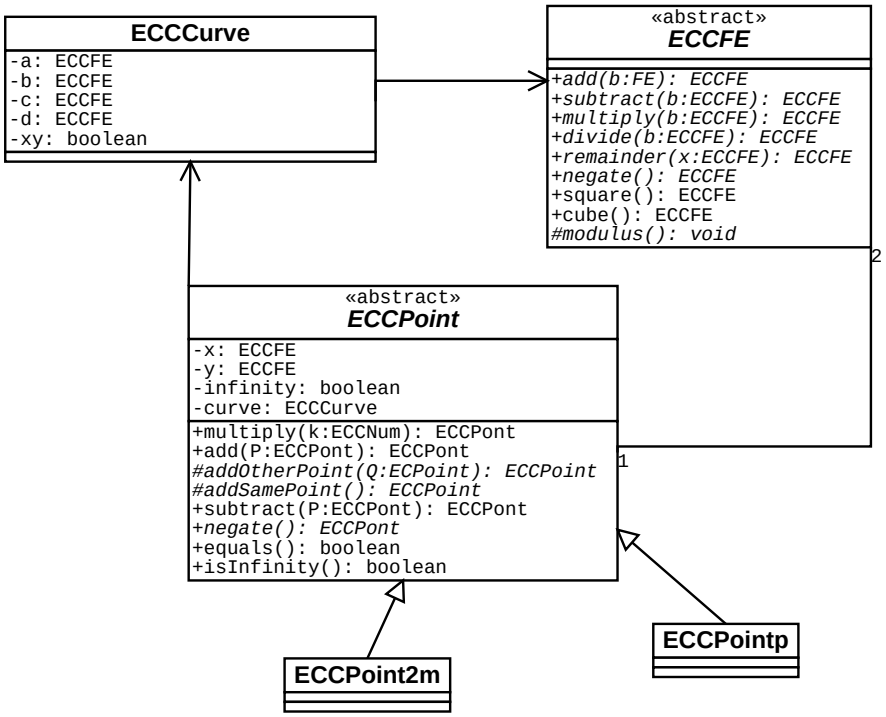
diagrammunkból, melyeken bizonyos - a lényeghez nem tartozó - attribútumokat, objektumokat nem tüntetünk fel.

4.4.2. A görbe (ECCCurve)



Az ECCCurve osztály egy Galois test felett értelmezett görbét jellemez. A lehető legáltalánosabb modell elérése érdekében x minden hatványa rendelkezhet együtthatóval, továbbá az egyenlet rendelkezhet xy-os taggal. A görbe egyenlete így: $y^2 + xy = ax^3 + bx^2 + cx + d$ lehet, de belőle az xy tag el is hagyható. Egy göbe objektum nem tudja, pontosan milyen test felett használjuk, ettől függetlenül működik, hiszen ő csak az ECCFE (Field Element) általános osztályt látja.

4.4.3. A görbe egy pontja (ECCPoint)



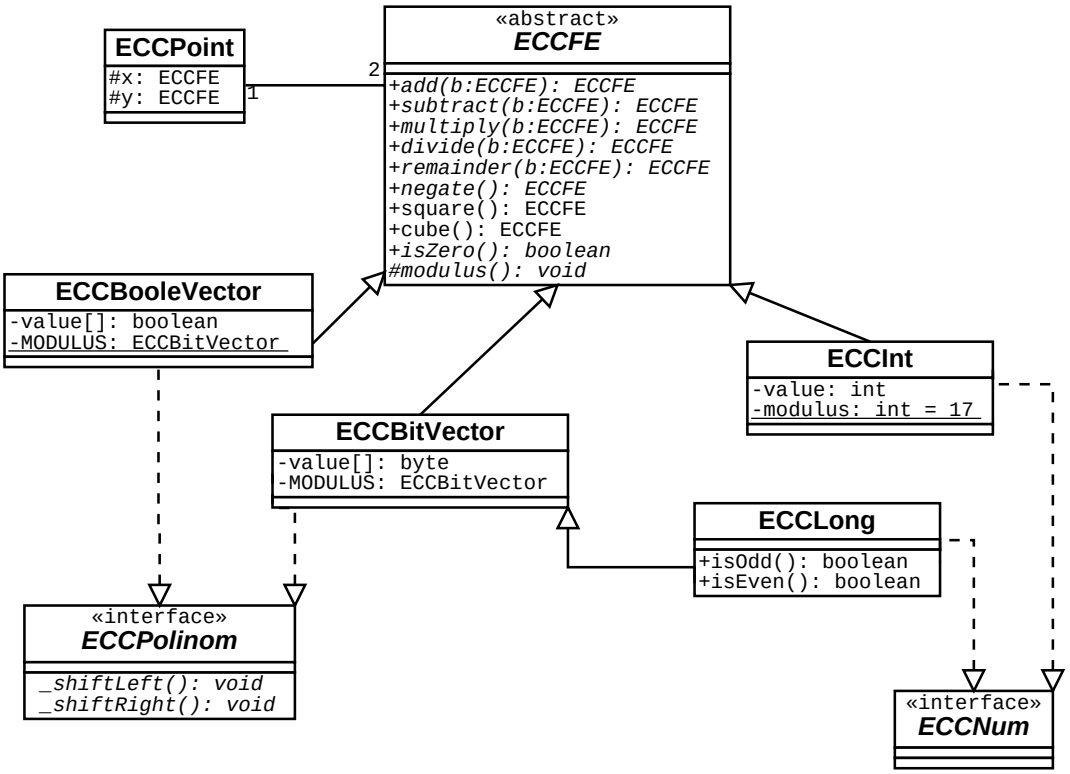
Az ECCPoint is absztrakt objektum, a gyakorlatban csak belőle származtatott ECCPoint2m és ECCPointp pontokat hozunk létre. Eredetileg tervünk az volt, hogy egyetlen ECCPoint osztály lesz, amely teljesen független a testtől, amelyben a számításainkat végezzük, de később hatékonysági okok miatt választottuk szét a $GF(p)$ és $GF(2^m)$ feletti pontokat. Az ECCPoint közös

interface-t biztosít ezek számára, s elrejti a testet az ECC rendszert használó osztályok elől.

Minden pont rendelkezik egy x és egy y koordinátával. Ezek a koordináták a test elemei, tehát ECCFE osztályú objektumok. Komoly problémát jelent ECC rendszerekben az O pont, a görbe pontjaiból álló csoport neutrális eleme. Mi kis méretű és egyszerű kód végett a naiv megoldást választottuk: egy boolean értéket vettünk fel, ez tárolja, hogy az adott pont az O pont-e. Amennyiben ez a boolean attribútum igaz értéket vesz fel, az x és y koordináták nem rendelkeznek értelemmel. (Sajnos a helyüket a memóriában felszabadítani nem tudjuk.)

4.4.4. Egy Galois test eleme (ECCFE)

Rendszerünk kulcsfontosságú osztálya az ECCFE absztrakt osztály, amely egy Galois test aritmetikáját írja le.



ECCFE példányokat természetesen soha nem hozunk létre, helyette belőle leszármaztatott osztályokat használunk, amelyek egy konkrét Galois testet írnak le. Ezeket két csoportra oszthatjuk: Egész számokra és polinomokra.

Az ECCInt és az ECCLong egész számok, s a $GF(p)$-beli aritmetikának megfelelően működnek. Az előbbi a java int típusa segítségével dolgozik, így a test méretét az int felülről korlátozza 32 bitben. Ennek az osztálynak kriptográfiai jelentősége természetesen nincsen, prototípusknak hoztuk létre, s a tesztelést segítette. Az ECCLong osztály tetszőleges hosszúságú pozitív egész ábrázolására képes. Mivel a $GF(2^m)$-béli aritmetika mellett döntöttünk, mivel ezt ítéltük hatékonyabbnak, ezt az osztályt teljesen nem valósítottuk meg. Néhány egyszerűbb műveletére szükség volt viszont az ECCPoint.multiply művelet implementálásakor. Ilyen volt a kettővel való osztás (jobbra shiftelés) és a páratlanság vizsgálata. Ezen okok miatt származtattuk végül az ECCBit-

Vector osztályból.

Az ECCPolinom interface-t implementáló két osztály, az ECCBooleVector és az ECCBitVector mindössze a polinom tárolásának és elérésének módjában különbözik egymástól.

4.4.5. Polinomok és azok bináris reprezentációi

Az ECCBooleVector és az ECCBitVector egyaránt tetszőleges hosszúságú polinomokkal képesek dolgozni. A különbség köztük az, hogy míg az ECCBooleVector osztály a Java nyelv beépített boolean típusát használja a polinom bitjeinek tárolására, az ECCBitVector számokat használ e célra.

Először csupán az ECCBooleVectort valósítottuk meg, mivel ezt ítéltük hatékonyabbnak. Ezzel a módszerrel a polinom egyes bitjei közvetlenül megcímezhetőek, s a boolean típussal gyors műveletvégzés lehetséges. Ugyanakkor hibája, hogy mivel a Java a boolean változókat 1 byte-on tárolja, 1 bit információval 8 bitet foglalunk el. Ez azt is jelentheti, hogy sok bit mozgatásakor (pl.: átmásolás, összeadás) jelentős adatterületet kell megmozgatnunk.

Az ECCBitVector esetén a Java byte típusát használtuk, s minden byte minden bitjét képesek voltunk kihasználni adataink számára. Egy polinom mozgatásakor sokkal kisebb adatterületet kellett írnunk-olvasnunk, így a chipkártya processzora sokkal kevesebb memóriahozzáférést kellett, hogy végrehajtson. Ugyanakkor az egyes bitek elérése sokkal drágább lett, hiszen a byte-ról le kellett választanunk a felesleges biteket. A Java nyelv támogatja ugyan a bitműveleteket, de támogatása az alacsony szintű nyelvekéhez képest igen szegényes. Nem rendelkezik a C-éhez hasonló unsigned típusokkal, továbbá bitműveleteket csakis int változók között lehet elvégezni. A Java Card specifikáció rosszallóan nyilatkozik az int típusról, s megvalósítását opcionálisnak tartja. Szerencsére az általunk használt Odyssey I kártya támogatja az int típust, de felhívja a figyelmet arra, hogy megvalósítása korántsem hatékony, hiszen a kártya csupán 8 bites processzorral rendelkezik.

A Java Card specifikáció mindennek ellenére tartalmaz bitműveleteket, de hogy megvalósításuk hogyan történik, arról nincsen pontos képünk. ECCBitVectort használva a mozgatás és az összeadás sebessége jelentősen csökken, viszont az egyes bitek tesztelésének ideje, valamint a szorzás és osztás ideje jelentősen megnő.

4.5. Tesztadatok ismertetése

Az általunk kifejlesztett ECC implementációt össze kívántuk vetni mások eredményeivel. Ezért olyan adatokat választottunk, amelyek nyilvánosan ismertek, s sokan dolgoztak már velük. Megfeleltek ezen szempontoknak a Certicom honlapján [ref] ECC Challenge néven kibocsájtott kihívások. Ezen feladványokkal a Certicom - mint egy elliptikus görbéken alapuló kriptográfiai technológiát kereskedelmi termékekbe integráló cég - demonstrálni kívánta az ECC erejét. Különböző nehézségű (kulcsméretű) kihívásokat támasztott a közönségnek, s megfelelő pénzdíjat ajánlott fel a kódok feltörőinek. Az ECC töréseinek helyzetét, állapotát egy későbbi fejezetben ismertetjük.

A mi választásunk az ECC2-109-es görbére esett a következő okokból:

- Gyakorlati szempontból is megfelelő erősségűnek tartjuk, hiszen ilyen méretű kihívást a mai napig senki nem oldott meg.

- Elméleti szepontból az erőssége megfelel a 1024-bites RSA-nak. [Certicom]
- $GF(2^m)$ feletti feladványról van szó, tehát az általunk hatékonyabbnak ítélt aritmetikával dolgozik.
- Kulcsmérete elég kicsi ahhoz, hogy az általunk hozzáférhető chipkártya memóriája elegendő a kódolás elvégzéséhez.

4.6. Java Card implementáció

4.6.1. Kitűzött célunk

Lévén a Java Card erőforráskészlete egyértelműen szerényebb a PC-énél, a chipkártyán való futás volt a szűk keresztmetszet. Tudtuk jól, hogy szoftveresen megvalósított algoritmusunk, amelyet ráadásul magasszintű nyelven írtunk, nem lehet versenyképes semmilyen hardver segítségével megvalósított algoritmussal. Így célunk az volt, hogy:

1. Programunk algoritmikus szempontból hatékony legyen (tehát a test méretének legfeljebb polinomjával legyen arányos a végrehajtási idő). Ez azt jelenti, hogy elméletileg bármekkora probléma esetén kivárható, amíg a kártya elvégzi a kódolást.
2. A kártya fizikailag képes legyen végrehajtani a programot.

4.6.2. Memóriamenedzsment

Ezek után fő problémánk a memóriagazdálkodás megszervezése maradt. Két gondot kellett orvosolnunk:

1. A kártya memóriája kicsi (7040 byte)
2. A Java Card specifikáció nem teszi lehetővég a Java-hoz hasonló memóriakezelést. Ez azt jelenti, hogy nem lehetséges dinamikusan objektuomokat létrehozni s megsemmisíteni. A programozó köteles minden memóriát az applet konstruktorában lefoglalni.
3. A Java nyelv nem teszi láthatóvá a programozó számára közvetlenül a memóriát. Így nem vagyunk képesek közvetlenül egyes memóriacímekre írni. Tehát az assemblyben szokásos memóriakezelés sem lehetséges.

Ki kellett dolgoznunk egy módszert, amely segítségével az applet élete elején lefoglaljuk a memóriát, majd rendelkezésére bocsátjuk a metódusoknak, akik használják azt, majd annak befejeztével visszanyerjük a memóriát.

Egyik megoldás erre a célra, hogy szoftveresen létrehozunk egy heap-et, amelyből mi gazdálkodunk. Metódusaink igényelhetnek belőle területet, majd miután nem kell nekik, visszaadják azt. Mivel programunk során ki kívántuk használni a Java nyelv magasszintű típusait, a heap nem egy véletlen elérésű byte tömb lenne, hanem cellákból állna, amelyek megfelelnének a Galois test elemeinek. Tehát ECCFE-leszármazott objektumokból állna. A heap használatát végül elvetettük.

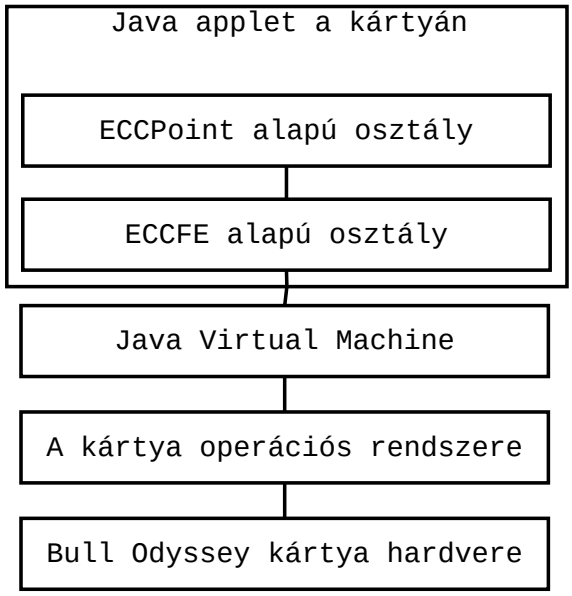
4.6.3. Regiszterkészlet

A másik megoldás, amelyet végül választottunk, egy regiszterkészlet definiálását jelentette. Osztályaink létrejöttükkor lefoglalnak maguknak objektumokat, s a későbbiek során csakis ezen objektumokat használjuk a számítások elvégzésére. Később nem foglalunk le magunknak új adatterületet, s ezen regiszter-objektumokat sem pusztítjuk el, hanem tartalmukat írjuk felül. (Itt utalni kívánunk a 3. fejezetben azon megjegyzésünkre, miszerint a Java nyelvű fejlesztés ellenére nem lehetséges a teljes objektum-orientáltság megvalósítása) Természetesen a regisztereknek nincsen semmi közük a kártya processzorának regisztereihez, hiszen egyrészt Java nyelven programozva nem is láthatjuk azokat, másrészt méretük akár többszáz byte is lehet.

Azért választottuk ezt a megoldást, mert ez logikailag is biztosítja, hogy a kártya memóriája nem fogyhat el. Az ellen is véd például, hogyha az első megoldásban az egyik művelet végén elfelejtenénk felszabadítani valamelyik segédváltozó által használt memóriát. Ilyen hibát teszteléssel lehetetlen lenne kimutatni, hiszen ha n futást megvizsgáltunk, lehetséges, hogy csak az $n + 1$. futtatás után fogyna el a kártya memóriája.

Voltak általános célú pszeudo-regisztereink is, de voltak olyanok is, amelyek bizonyos speciális feladatokat láttak el. A GF-eredményregiszter például arra szolgált, hogy a test műveletei ezen regiszterbe írják be eredményüket, s később innen lehet majd azt kiolvasni. Nem egyezett meg a regiszterek típusai sem. Többségük a test elemei közt való műveletek elvégzésére szolgált, s itt töltötte be segédváltozó szerepét. De voltak olyan regiszterek is, amelyek nem testelem-regiszterek, hanem pont-regiszterek voltak. (nem ECCFE-leszármazottak, hanem ECCPoint leszármazottak)

4.6.4. Rétegszerkezet



Érdekes helyzet állt elő: Magukat a műveletek a programot futtató gép (vagy kártya) processzora végzi el a processzor regiszterei közt. A JVM ezen regisztereket elfedi a programozó elől, s byteokat, objektumokat tesz láthatóvá. Mi ezen byte-ok felett hoztunk létre ECCFE-alapú pszeudoregisztereket, s a Galois-test műveleteit ezek között végeztük el. A felsőbb szint felől pedig

elfedtük a Galois-test műveleteinek implementációját, s létrehoztunk ECCPoint-alapú pontregisztereket. Az ECC implementációnkat használó program elől próbáltunk elfedni minél többet a pontregiszterekből. Mindössze egyetlen egy, a pontok eredményregisztere látható kívülről, s az eredményt belőle kell kiolvasni.

4.6.5. Futási eredmények

Kártyánkra feltelepítettük az ECC2-109 paraméterű ECC Challenge-t [Certicom]. Méréseink eredménye értelmében egy elemi műveletet (vagyis két pont összeadását) a rendszer 7 perc 8 másodperc alatt végezte el. Hasse tétele értelmében körülbelül 100 összeadás elvégzésénél többre nem lesz szükség egy kódoláshoz. Ez összesen 11 órát jelentene. (hangsúlyozzuk, hogy implementációnk a technológia jelen szintjét demonstráló prototípusnak készült, nem pedig kereskedelmi forgalomba kerülő terméknek)

4.6.6. Levont következtetés

ECC rendszerünk egy Java Card appletbe beágyazva képes elvégezni egy ECC kódolást, de sebessége olyan kicsi, hogy a gyakorlatban csakis akkor alkalmazható, ha feltétlenül szükséges a chipkártya által nyújtott biztonság kihasználása.

4.7. PC implementáció

4.7.1. PC kompatibilis Java Card program

Igaz, programunkat azon fő szempont szerint fejlesztettük, hogy működjön Java Cardon, ugyanaz a Java kód sikeresen lefut PC-n is. Egy közönséges JDK segítségével lefordítható, s bármilyen Java programba beágyazható. PC-n így nem jelentkeznek a Java Card sebesség és méret-korlátozásai, tehát tetszőleges méretű test felett dolgozhatunk. A 32-bites processzor viszont az int típust kezeli hatékonyan, s mivel mi a Java Card miatt a byte és short típusokat használtuk, így a program PC-s Java kódnak nem optimális.

Hogyha programunkat PC-re fejlesztettük volna, legalább a sebesség szempontjából kritikus részeket valamely alacsony szintű nyelven írtuk volna meg.

4.7.2. Futási eredmények

PCs környezetben is az ECC2-109-es kihívással teszteltünk. Itt a mért eredmény természetesen sokkal jobb: A rendszer egy kódolást, vagyis egy pontnak konstanssal való szorzását (kb 100 összeadás) 28 másodperc alatt végzett el.

4.7.3. Levont következtetés

PC-s környezetben a program alkalmas gyakorlati haszálatra is, bár elsősorban nem PC-re készült, így közel nem a leggyorsabb implementáció.

4.8. Tanulság

Programunkat elsősorban demonstratív céllal írtuk meg. Egyrészt fel kívántuk hívni vele a figyelmet a kis méretű kulcsokkal nagy biztonságot nyújtó ECC algoritmusok chipkártyán való használatára. Programunk illeszkedik az ECC matematikai struktúrájához, tehát a Galois-test aritmetikája, és a pontok csoportjának aritmetikája elválasztható egymástól, s bármikor kicserélhető. Demonstrálni kívántuk továbbá, hogy a mai (illetve néhány évvel ezelőtti) chipkártyák már képesek ECC műveletek végrehajtására is, bár a sebességük jelenleg igen csekély.

Meg kell jegyeznünk, hogy mi magasszintű, tehát nem hardverközeli, nyelven dolgoztunk, mindennemű hardvertámogatás nélkül. Nem tartjuk kizártnak, hogy chipkártyán célhardverrel való (vagy legalábbis assembly nyelven írt) megvalósítás sebessége igen sokszorosa lehet a miénknek, még a mai technológia adta lehetőségek közt is.

Figyelembevéve a hardverelemek és a chipkártyák rohamos fejlődését (pl amelyet a Moore törvény is demonstrál) igen valószínűnek tartjuk, hogy a néhány éven belül megjelenő jelentősen gyorsabb kártyákon sokkal gyorsabban fog futni a mi imlementációnk is.[hiv a hiradcikkre] Memória szempontjából pedig már ma is léteznek olyan Java Card-ot támogató chipkártyák, amelyek nagy méretű memóriájukból (32k-64k) adódóan számos gyorsítási lehetőség előtt nyitnak teret.

5. Elemzés

5.1. Teljesítmény szempontjából kritikus pontok

5.1.1. Kritikus sebességű pontok megnevezése

Két olyan pontot találtunk, amelyet a teljesítmény szempontjából igen kritikusnak ítéltünk. Az egyik ilyen a testben való osztás (inverzzel való szorzás), a másik pedig a pont konstanssal való szorzása.

Ráadásul e két művelet egymásba ágyazódik, tehát algoritmikus komplexitásuk összeadódik.

5.1.2. Gyorsítási lehetőségek

A testelemek közt végzett osztás két részből áll: inverzképzésből és szorzásból. Az inverzképzést euklideszi algoritmussal végezzük, amely igen hatékony. Ugyanakkor az euklideszi algoritmus is szorzásokból, összeadásokból és maradékos osztásokból áll. Amennyiben ezen műveleteket felgyorsítjuk, maga az inverzképzés is gyorsul.

A szorzást és maradékos osztást sokkal gyorsabban végezhetjük el, ha az egybefüggő 0 területeket nem egyesével, hanem egyben shifteljük végig. Azt is megtehetnénk, hogy nem végzünk minden egyes elforgatás után modulusképzést. Elvégezzük a szorzást, s kapunk egy legfeljebb $2m$ fokszámú polinomot. Ebből a modulust maradékos osztás elvégzésével kapnánk.

A szorzást ráadásul sokkal hatékonyabb lenne elvégezni optimális normál bázis reprezentációval. Hogy ezt nem így tesszük, annak egyik oka, hogy ilyenkor az összeadás lassul le, másik pedig, hogy az algoritmus biztonsága is megszenvedheti. (2. fejezet)

Másik gyorsítási lehetőség az lenne, ha ismernénk az alaptest multiplikatív csoportjának egy generátorelemét, hiszen így a szorzást tulajdonképpen összeadásként tudnánk értelmezni.

Sokkal nehezebb probléma a pont egész számmal való szorzásának gyorsítása. Figyelembe véve, hogy nem ismeretes közvetlen képlet kP kiszámítására, a szorzás csakis összeadásokkal valósítható meg. Ezt figyelembe véve jelen algoritmusunk igen hatékonynak mondható. Ugyanakkor konstansszorosára gyorsíthatjuk például azzal, hogyha a 2^iP értékeket előre kiszámoljuk, és egy táblázatban tároljuk. Ekkor is $O(log(k))$ műveletet kell elvégeznünk, de a műveletek száma durván felére csökken, ugyanis jelentős mennyiségű pontduplázást nem kell elvégeznünk. Ezzel a módszerrel csak akkor érünk el eredményt, ha egy P pontot viszonylag sokszor használunk, tehát sok k számmal kell megszoroznunk. Figyelembe véve, hogy P igen gyakran nyilvános kulcs, amellyel viszonylag sok kódolást kell elvégezni, ezen gyorsítási lehetőség egyáltalán nem irreális. Nehezen megvalósítható viszont a mai memóriaméretek mellett chipkártyán, hiszen $m*log(k)$, tehát $O(m^2)$ méretű táblázatot vagyunk kénytelenek tárolni.

5.2. Törési próbálkozások

Mivel az ECC biztonságára vonatkozóan nincsen elméleti bizonyítékunk, így erről csak a gyakorlati tapasztalatok nyomán alkothatunk képet. E célból nagyon hasznos, ha áttekintjük, milyen törési módszerekkel próbálkoztak eddig, és ezek milyen sikerrel jártak. Először röviden összefoglaljuk az ismert törő algoritmusokat, majd a törési eredményeket.

5.2.1. Törési algoritmusok

A legegyszerűbb megoldási algoritmus az ECDLP-re a naív kimerítő keresés. Ennek során sorban kiszámítjuk a P, $2P$, $3P$,... pontokat, amíg el nem érjük $Q = lP$-t. Ez legrosszabb esetben n lépést vesz igénybe, ahol n a P pont rendje.

A „gyerek-lépés-óriás-lépés" (baby-step-giant-step) algoritmus egy sebesség-memória kompromisszumnak tekinthető. Ennek során a lépések száma csupán $\sqrt{n}$, de körülbelül $\sqrt{n}$ pont egyidejű tárolására is szükség van.

A Pollard-féle rho algoritmus a gyerek-lépés-óriás-lépés algoritmus randomizált változatának tekinthető. Lépésszáma hasonló ($\sqrt{\pi n/2}$), de memória-igénye konstans. Ráadásul Gallant, Lambert és Vanstone illetve Wiener és Zuccherato módszerével egy $\sqrt{2}$-es faktorral gyorsítható az algoritmus.

Ennél azonban lényegesen fontosabb Van Oorschot és Wiener eredménye, mely szerint a rho algoritmus hatékonyan párhuzamosítható. Ez azt jelenti, hogy m processzor párhuzamos használata esetén valóban m-szeres gyorsítás érhető el. Mai tudásunk szerint a párhuzamos rho algoritmus a leghatékonyabb általános eljárás az ECDLP megoldására.

A Pohlig-Hellman algoritmus akkor működik jól, ha n prímfaktorizációjában viszonylag kis prímek (p_i) szerepelnek. Ilyenkor ugyanis l meghatározása redukálható l-nek modulo p_i történő meghatározására, amikből aztán l a kínai maradéktétel segítségével kapható meg. Annak érdekében, hogy ez az eljárás ne legyen használható, gyakran n-et prímszámnak választják, vagy pedig egy nagy prím néhányszorosára.

Pollard lambda algoritmusa is, csakúgy, mint a rho algoritmus, jól párhuzamosítható, azonban általánosságban kicsivel lassabb. Gyorsabb azonban olyan esetekben, amikor tudjuk, hogy a keresett logaritmus kisebb, mint $0.39n$.

Silverman és Stapleton megmutatták, hogy egy adott ECDLP megoldása során nyert információkat fel lehet használni ugyanazon görbére és alappontra vonatkozó további ECDLP-k megoldására. Pontosabban, k ECDLP megoldása kb. $\sqrt{k}$-szor annyi idő alatt elvégezhető, mint egyetlen ECDLP. Ehhez azonban igen sok adat tárolására van szükség. Annak érdekében, hogy ezt a módszert ne lehessen sikeresen alkalmazni, célszerű már az első ECDLP-t is olyan nagyra választani, hogy annak megoldására ne legyen esély.

5.2.2. Törési eredmények

A törési kísérletek fő mozgatórugója a kanadai Certicom által szponzorált „ECC Challenge". A vállalat 500.000 dollár összdíjazású feladatsort tett közzé honlapján. A feladatokat növekvő számításigény alapján három kategóriába sorolták. Minden kategórián belül vannak $GF(p)$ és $GF(2^r)$ feletti görbék, illetve véletlenszerűen generált és speciális, ún. Koblitz típusú görbék. (A Koblitz-görbéket olyan egyenlet definiálja, melynek $GF(2)$ felett pontosan két megoldása van.) A legelső kategória neve „Excercises". Ebben 79-97 bites problémák vannak, melyeket a Certicom becslése szerint egyetlen számítógép néhány év alatt tudna megoldani, nagy hálózatok azonban néhány nap alatt. A második kategória neve „Level I Challenges". Ebben 108-131 bites feladatok vannak, a megoldásukhoz szükséges becsült idő egyetlen gép számára 10^3-10^7 év. A harmadik kategóriában („Level II Challenges") pedig 163-358 bites feladatok vannak, ezek megoldása egyetlen géppel 10^{12}-10^{41} évig tartana.

Ezek voltak tehát a Certicom becslései 1997-ben, a Challenge indításakor. Azóta az első kategóriában lévő valamennyi feladatot megoldották, és a második kategóriából is egyet. A legnagyobb, eddig megoldott probléma 108 bites, ennek feltörésére 2000. áprilisában került sor, miután 9500 számítógép négy hónapon keresztül dolgozott a problémán. A befektetett számítási kapacitás 50-szerese volt az 512 bites RSA feltöréséhez felhasználtnak.

Egy kivételével valamennyi törési eredményt a párhuzamosított rho algoritmussal érték el, ráadásul ugyanaz a Robert Harley által vezetett team. Az eddigi eredmények azt mutatják, hogy a Certicomnak a szükséges számítási kapacitásra vonatkozó becslései lényegében helyesek voltak. Ebből fakadóan ma úgy látszik, hogy egy kb. 150 bites ECDLP lényegében megoldhatatlan.

5.3. Kritikus pontok a biztonság szempontjából

Annak érdekében, hogy kellően biztonságos ECDLP-hez jussunk, nem elég csupán arra figyelni, hogy az alaptest elég nagy legyen. Legalább ilyen fontos az is, hogy a görbének sok pontja legyen, valamint hogy a választott alappont rendje nagy legyen, hiszen, mint láttuk, az ismert törő algoritmusok lépésszáma alapvetően ettől függ. Ezen kívül néhány speciális esetet, amire van hatékony törés, el kell kerülni. Ilyen esetek:

- szuperszinguláris görbék
- anomalikus görbék: ezek olyan, $GF(q)$ feletti görbék, melyeknek pontosan q pontja van
- $GF(2^r)$, r összetett

A Koblitz-görbékre vonatkozóan nincsen sem olyan eredmény, amely azt mutatná, hogy ezek lényegesen biztonságosabbak lennének a véletlenszerűen választott görbéknél, sem olyan, ami ennek az ellenkezőjét sejtetné.

Egy konkrét kriptográfiai rendszer tervezésénél azt is mérlegelni kell, hogy az egész rendszert egyetlen görbére és alappontra érdemes építeni, vagy pedig minden szereplő választhasson magának saját görbét és alappontot. Mivel rengeteg elliptikus görbe van, ezért az utóbbi módszernek nincs akadálya, és megvan az az előnye, hogy a rendszer biztonsága nem lényegében egyetlen ECDLP nehézségére van alapozva. Hátránya viszont, hogy az egyes felhasználók esetleg nem elég nehéz problémákat választanak, így az egész rendszer biztonságát kockáztatva.

6. Összefoglalás

Az elliptikus görbéken alapuló nyilvános kulcsú kriptográfia fiatal, kevéssé ismert elméletnek számít. Igaz, alapelvei lassan húsz éve ismertek, de - főként bonyolult matematikája miatt - sokkal kisebb elterjedtségnek örvend, mint az RSA.

Ugyanakkor, mivel számos előnnyel rendelkezik nagy vetélytársával szemben, szerepe jelentősen megnövekedett, s ma már igen sok szabványban szerepel az ECC is az RSA mellett. (2.2.2)

Reméljük, kutatásainkkal mi is hozzájárulunk az ECC terjedéséhez.

Chipkártyás implementációnkkal demonstrálni szeretnénk az ECC jelentőségét, továbbá a Java Card technológia számítási erejét, amely ma már szoftveresen is képes elvégezni bizonyos kriptográfiai műveleteket.

Célunk nem egy kereskedelmi forgalomba kerülő termék megalkotása volt. Munkánk inkább prototípusértékűnek tekinthető, amely később alapul szolgálhat egy célhardverrel megvalósított chipkártyás ECC számára.

7. Irodalomjegyzék

7.1. Matematikai és algoritmuselméleti referenciák

[Katona1997] Katona Gy. Y., Recski A., Szabó Cs.: Gráfelmélet, algoritmuselmélet és algebra.BME, 1997.

[Mullin1988] R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone, R. M. Wilson: Optimal Normal Bases in GF(p^n). in: Discrete Applied Math, vol 22. (Elsevier, 1988, p. 149-161)

[Ronyai1999] Rónyai L., Ivanyos G., Szabó R.: Algoritmusok. Typotex, Budapest, 1999.

7.2. Elliptikus görbékre vonatkozó referenciák

[Husemoller1987] D. Husemöller: Elliptic Curves. Springer, 1987.

[Koblitz1987] N. Koblitz: Elliptic Curve Cryptosystems. in: Mathematics of Computation (1987), p. 203-209.

[Miller1985] V. S. Miller: Use of Elliptic Curves in Cryptography. in: CRYPTO '85 (Springer, 1986: p. 417-426.)

[Ronyai1995] Rónyai L.: Elliptikus görbék és a Fermat-sejtés. in: Matematikai Lapok 1992/3-4. Budapest, 1995.

[Rosing1995] M. Rosing: Implementing Elliptic Curve Cryptography. Manning, 1999.

[Wiener1998] M. J. Wiener, R. J. Zuccherato: Faster Attacks on Elliptic Curve Cryptosystems. Entrust, 1998.

[Win1999] E. D. Win, B. Preneel: Elliptic Curve Public Key Cryptosystems - an introduction. ftp://ftp.esat.kuleuven.ac.be/pub/cosic/dewin/coursetext.ps.gz

[Crutchley1999] D. A. Crutchley: Cryptography and Elliptic Curves. Masters in Mathematics thesis, University of Southampton, 1999.

7.3. Kriptográfiai referenciák

B. Schneier: Applied Cryptography, 2nd ed. John Wiley & Sons, 1996.

Györfi - Vajda: A hibajavító kódolás és a nyilvános kulcsú titkosítás elemei, 1991

D. W. Davies - W. L. Price: Security for Computer Networks. John Wiley & Sons, 1992

[Hornak] Hornák Zoltán honlapja: http://www.mit.bme.hu/~hornak

[Crack1999] www.certicom.com/news/99/sep2899.html

[Crack2000] www.certicom.com/news/00/apr1700.html

7.4. Chipkártyákkal kapcsolatos referenciák

Berta István Zsolt - Mann Zoltám Ádám: A hitelesség biztosításának lehetőségei intelligens smart card segítségével (TDK dolgozat, 1999 nov.)

Berta István Zsolt - Mann Zoltán Ádám: Programozható chipkártyák - elmélet és gyakorlati tapasztalatok (Magyar Távközlés, 2000. április)

W. Rankl - W. Effing: Smart Card Handbook. John Wiley & Sons, 1997

Bruce Schneier - Adam Shostack: Breaking Up Is Heard To Do: Modelling Security Threats for Smart Cards

J. L. Zoreda - J. M. Oton: Smart Cards: Artech House, 1994

Java Card honlap: http://java.sun.com/products/javacard/htmldoc

[JavaCard1997] Java Card 2.0 Language Subset and Virtual Machine Specification (Sun Microsystems) 1997.

[Java] Java nyelv: http://java.sun.com

7.5. Egyéb referenciák

Az UML nyelv definíciója: http://www.becrc.org/ec/stds/uml.html

[Certicom] Certicom honlap: www.certicom.com

8. Rövidítések jegyzéke

ANSI American National Standard Institute

APDU Application Protocol Data Unit (3.2.7)

API Application Programming Interface

DH Diffie-Hellman

DLP Discrete Logarithm Problem

DSS Digital Signature Standard

ECC Elliptic Curve Cryptography

ECDLP Elliptic Curve Discrete Logarithm Problem

GF Galois Field

IEEE Institute of Electrical and Electronics Engineers

IFP Integer Factorisation Problem

ISO International Standard Organisation

JVM Java Virtual Machine

LSB Least Significant Bit

MIPS Million Instructions per Second

MSB Most Significant Bit

PK Public Key

PKI Public Key Infrastructure

RSA Rivest-Shamir-Adleman: három felfedezőjéről elnevezett nyilvános kulcsú titkosítási rendszer

SSL Secure Socket Layer

UML Unified Modelling Language